Device Installation: Windows DDK

Device Installation Overview

Before reading this section, see Providing an Installation Package for an introduction to the concepts and requirements you'll need to know concerning device installation.

Setup works with other system-supplied components and with vendor-supplied components to install devices. Setup installs devices when the system boots and at any time after boot when a user plugs in a Plug and Play (PnP) device (or manually installs a non-PnP device).

In support of PnP, Setup proceeds with device installation based on the devices in the system, rather than structuring installation around the drivers. For example, rather than loading a set of drivers and having those drivers detect the devices that they support, Setup determines the devices that are present in the system and loads and calls the drivers for each device. Drivers such as the ACPI driver and other PnP bus drivers help Setup determine which devices are present.

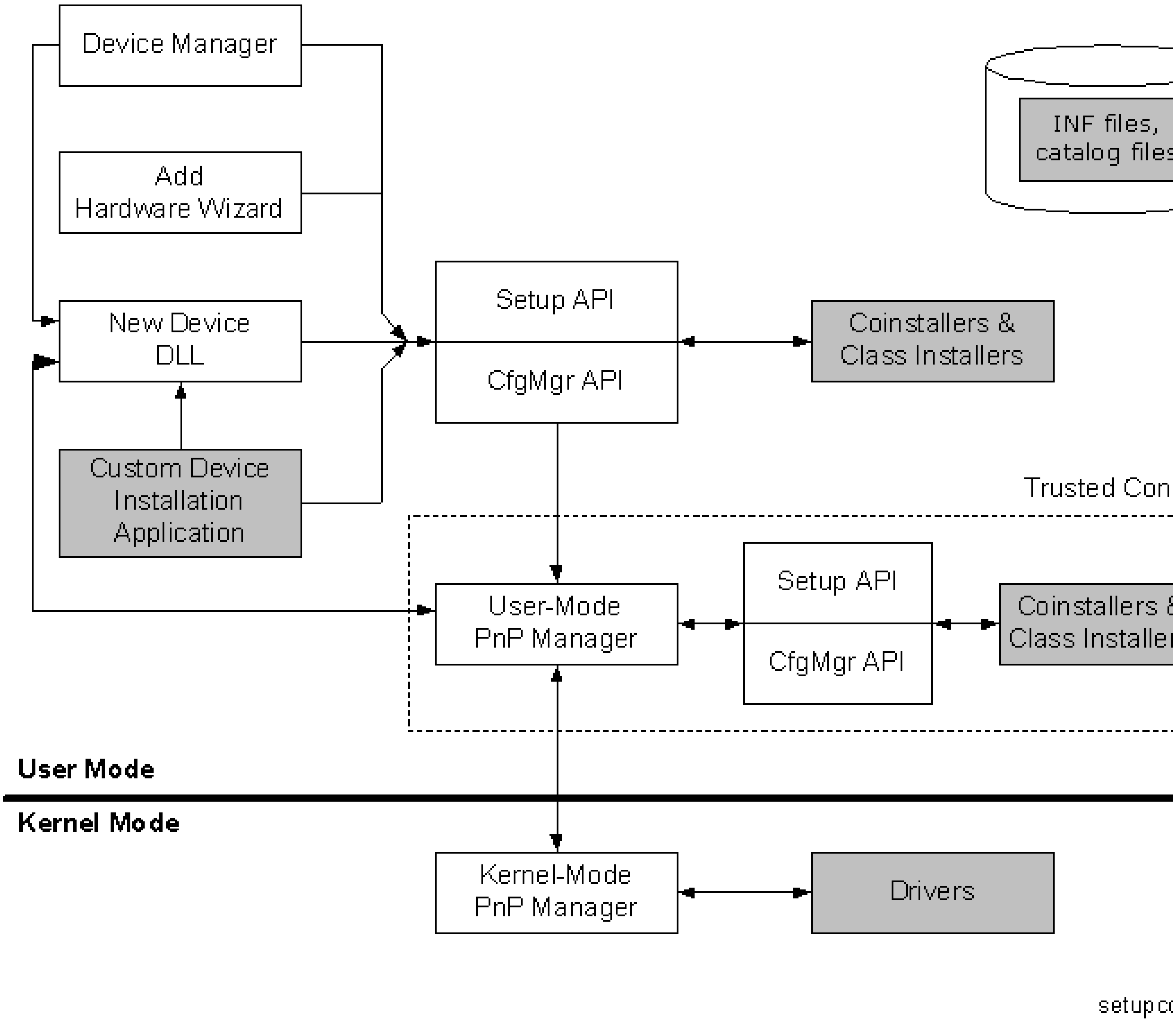
This section includes:

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Device Installation Components

The following figure shows the software components involved in installing a device.



Device Installation Setup Components

The shaded boxes in the figure represent components that can be provided by independent hardware vendors (IHV) and original equipment manufacturers (OEM). The other device installation components are supplied by Microsoft.

The following list describes the roles that the various components play in device installation:

Kernel-Mode PnP Manager

> The kernel-mode PnP Manager notifies the user-mode PnP Manager that a new device is present on the system and needs to be installed. The kernel-mode PnP Manager also calls the **DriverEntry** and *AddDevice* routines of a device's drivers and sends the IRP_MN_START_DEVICE request to start the device.
>
> The PnP Manager has two parts: one runs in user-mode and the other runs in kernel mode. The kernel-mode PnP Manager reports PnP events to the user-mode PnP Manager, and the user-mode PnP Manager sends control requests to the kernel-mode PnP Manager.

Drivers

> PnP drivers perform device-installation operations as directed by the PnP Manager. For example:
>
> - When the PnP Manager sends the driver an IRP_MN_QUERY_DEVICE_RELATIONS request for **BusRelations**, a PnP bus driver returns a list of its child devices.

- The PnP Manager calls each PnP driver's **DriverEntry** and *AddDevice* routines.
- The PnP Manager also send such additional requests as IRP_MN_QUERY_CAPABILITIES, IRP_MN_START_DEVICE, and IRP_MN_REMOVE_DEVICE.

PnP drivers can also detect nonPnP devices and report them to the PnP Manager using **IoReportDetectedDevice**.

User-Mode PnP Manager

The user-mode PnP Manager receives device installation requests from the kernel-mode PnP Manager, calls other user-mode Setup components to initiate device-installation tasks, and sends control requests (such as "start the device") to the kernel-mode PnP Manager.

The user-mode PnP Manager and the kernel-mode PnP Manager together maintain the device tree.

The user-mode PnP Manager tries to install a device in a trusted process context without requiring user responses to dialog boxes (a server-side installation). This automatic installation provides a better user experience. If the PnP Manager is unable to complete a trusted installation because, for example, an installer supplies custom Finish pages, the PnP Manager terminates the trusted installation. In this case, when a user with administrative privileges logs in, the PnP Manager restarts the device installation by launching the Found New Hardware wizard in the New Device DLL (a client-side installation).

Setup API

The Setup API includes the general Setup functions (**Setup***Xxx*) and the device installation functions (**SetupDi***Xxx*). These functions perform many device installation tasks such as searching for INF files, building a potential list of drivers for a device, copying driver files, writing information to the registry, registering device co-installers, and so forth. Most other Setup components call these functions.

The **SetupDi***Xxx* functions are also sometimes referred to as the *device installer*. This term can be confusing, however, because more than one component is involved in installing a device.

For more information, see General Setup Functions and Device Installation Functions.

CfgMgr API

The Configuration Manager API provides basic installation and configuration operations that are not provided by Setup API. The Configuration Manager Functions perform low-level tasks such as getting the status of a devnode and managing resource descriptors. These functions are primarily called by Setup API but can also be called by other Setup components.

Co-installers and Class Installers

A class installer performs installation operations that apply to devices in a particular device setup class. For example, the ports class installer is responsible for assigning a COM port name to a device in the ports setup class. If the devices in a particular setup class do not require any special installation operations, a class installer is not required for that setup class.

Microsoft provides class installers for most of the system-defined device setup classes. For more information, see Writing a Class Installer. IHVs and OEMs can provide class installers, but they typically provide co-installers.

A co-installer performs installation operations that are specific to a particular device or to a setup class of devices. For more information, see Writing a Co-installer.

INF Files and Catalog files

INF files and catalog files provide information about devices and drivers to be installed. For more information, see Creating an INF File.

Device Manager

The Device Manager allows a user to view and manage the devices on a system. For example, a user can view device status and set device properties. If a user requests to update a driver, the Device Manager calls the Hardware Update wizard in the New Device DLL.

For more information, see Using the Device Manager. Also see the Device Manager's online help.

Add Hardware Wizard
The Add Hardware Wizard allows a user to add, remove, unplug, and troubleshoot devices.

New Device DLL
The New Device DLL contains the Found New Hardware wizard, the Hardware Update wizard, and the **UpdateDriverForPlugAndPlayDevices** function.

The user-mode PnP Manager calls the Found New Hardware wizard to initiate a client-side installation of a new device in the context of a user with the appropriate privileges. The Device Manager calls the Hardware Update wizard when a user clicks the **Update Driver** button on a device's Driver property page. The Found New Hardware and Hardware Update wizards call Setup APIs and Configuration Manager APIs to accomplish their tasks.

Custom Device Installation Application
An IHV or OEM can provide a custom device installation application. For more information, see Writing a Device Installation Application.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Device Installation Files

The software required to support a particular device depends on the kind of device and the ways in which the device is used. Typically, a vendor provides the following software in a driver package to support a device:

- A device setup information file (INF file).

  An INF file contains information that the system Setup components use to install support for the device. Setup copies this file to the *%windir%\inf* directory when it installs the driver. This file is required.

  For more information, see Creating an INF file.

- One or more drivers for the device.

  A *.sys* file is the driver's image file. Setup copies this file to the *%windir%\system32\drivers* directory when the driver is installed. Drivers are required for most devices. For more information, see Determining Device-Specific Driver Requirements.

- Digital signatures for the driver package (a driver catalog file).

  A driver catalog file contains digital signatures. All driver packages should be signed.

  A vendor gets digital signatures by submitting its driver package to the Windows Hardware Quality Lab (WHQL) for testing and signing. WHQL returns the package with a catalog file (*.cat* file). For more information, see Driver Signing.

- One or more co-installers.

  A co-installer is a Win32® DLL that assists in device installation. For example, an IHV might provide a co-installer to write device-specific information to the registry that cannot be handled by the INF. Co-installers are optional. For more information, see Writing a Co-installer.

- Other files.

A driver package can contain other files, such as a Custom Device Installation Application, a device icon, a driver library file (for example, for video drivers), and so forth. For more information, see:

Writing a Device Installation Application
Providing Device Property Pages
Drivers with Special Installation Requirements

Also, see the device type-specific documentation in this DDK.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Device Installation Types

Windows defines two types of device installations: those that occur on the "server side" and those that occur on the "client side". Because they do not require user intervention by an administrator, server-side installations are more desirable than client-side installations.

Server-side Installations

A server-side installation is a device installation that can be handled entirely by the system's user-mode and kernel-mode PnP Manager components. A server-side installation can occur only if:

- The device can be detected by a bus driver, which then must notify the PnP Manager.
- The PnP Manager can find an INF file containing a hardware ID that matches that of the device.
- The device's INF file and drivers are signed (see Driver Signing).
- SetupAPI can locate all of the device's drivers without prompting the user for media locations. (This means the drivers are included "in-box" with the operating system, a vendor-supplied driver was previously installed, or vendor-supplied driver files have been "pre-installed" so SetupAPI can locate them during the actual installation (see Pre-installing Driver Files).
- The device's class installer and co-installers do not display property pages at the end of the installation operation.
- The device's INF file has not marked the device as requiring an interactive installation (by specifying **InteractiveInstall** in an INF **ControlFlags** section).
- **RunOnce** registry entries consist only of calls to *rundll32.exe*.

Server-side installations do not display any user interface and do not require an administrator to be logged on. (The term "server-side" is used because installation can be accomplished by the system's PnP Manager without a user-mode "client" making calls into the PnP Manager.)

Client-side Installations

A client-side installation is a device installation that requires communication with a user. A client-side installation occurs if any of the following are true:

- The PnP Manager cannot find an INF file containing a hardware ID that matches that of the device.
- The PnP Manager cannot locate all the required driver files.
- The device's INF or driver files are not signed.
- SetupAPI has to display any form of user interface, such as prompting the user for media locations.
- A class installer or co-installer displays property pages at the end of the installation operation.
- The device, bus, or bus driver does not support Plug and Play.

Client-side installations require use of installation "client" software, such as the Add Hardware wizard, the New Device DLL, or a vendor-supplied device installation application. This client software installs the device by making calls to SetupAPI, which in turn calls the PnP Manager when necessary. (The PnP Manager is considered to be the installation operation's "server side.")

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

System Setup Phases

For most devices, you should design your device and supporting files so that Setup can install the device at an appropriate time during system startup, or dynamically at any time after startup while the system is running. Certain devices, however, must be installed during a particular phase of system startup. To assist vendors of such devices, this section provides some background information on the system setup phases as they relate to device installation.

Installations During Text-Mode Setup

When Windows® boots on a system, the initial phase of the Setup program installs only the minimum number of devices needed for Windows to run, such as the keyboard, mouse, video adapter, SCSI/Disk, and Machine/HAL. This phase of the Setup program is known as *text-mode setup*.

For a user to install your device during text-mode setup, your distribution disk must include a text file named *txtsetup.oem*. For more information, see Installing a Boot Driver.

Installations During GUI-Mode Setup

After text-mode setup has completed, the Setup program boots Windows and proceeds with the *GUI-mode setup* phase of the installation. Setup installs most devices during GUI-mode setup.

Devices that are not installed during GUI-mode setup include those that require user interaction to be installed. For example, if this is the first time a device is being configured on the system and a co-installer for the device supplies one or more custom Finish pages, Setup must display the pages to the user. Setup performs such a device installation when a user with administrative privileges logs on.

Installations After the System is Running

Once a system is up and running, a user can install new devices:

- To install a new PnP device, plug the device into the system.

  If the device and the bus support hot-plug notification, device installation is initiated automatically.

  If the user needs to power down the system and open the system box to plug in the device, the device will be recognized and configured when the system boots.

  If the device can be plugged in while the system is running and the device and the bus do not support hot-plug notification, use the Add Hardware Wizard to initiate device configuration. (Select Install.)

- To install a nonPnP device, use the Add Hardware Wizard.

If Setup can install the device in the trusted context of a system process, the user does not need administrative privileges to install the device. If Setup needs to prompt the user for information, it requires the process context of an administrative user to install the device.

To update a driver for an installed device, use the Device Manager or the Hardware tab of the appropriate control panel applet. (Clicking the **Properties** button in the Hardware tab of a Control Panel applet brings up the Device Manager properties for the device.)

Built on Thursday, July 19, 2001

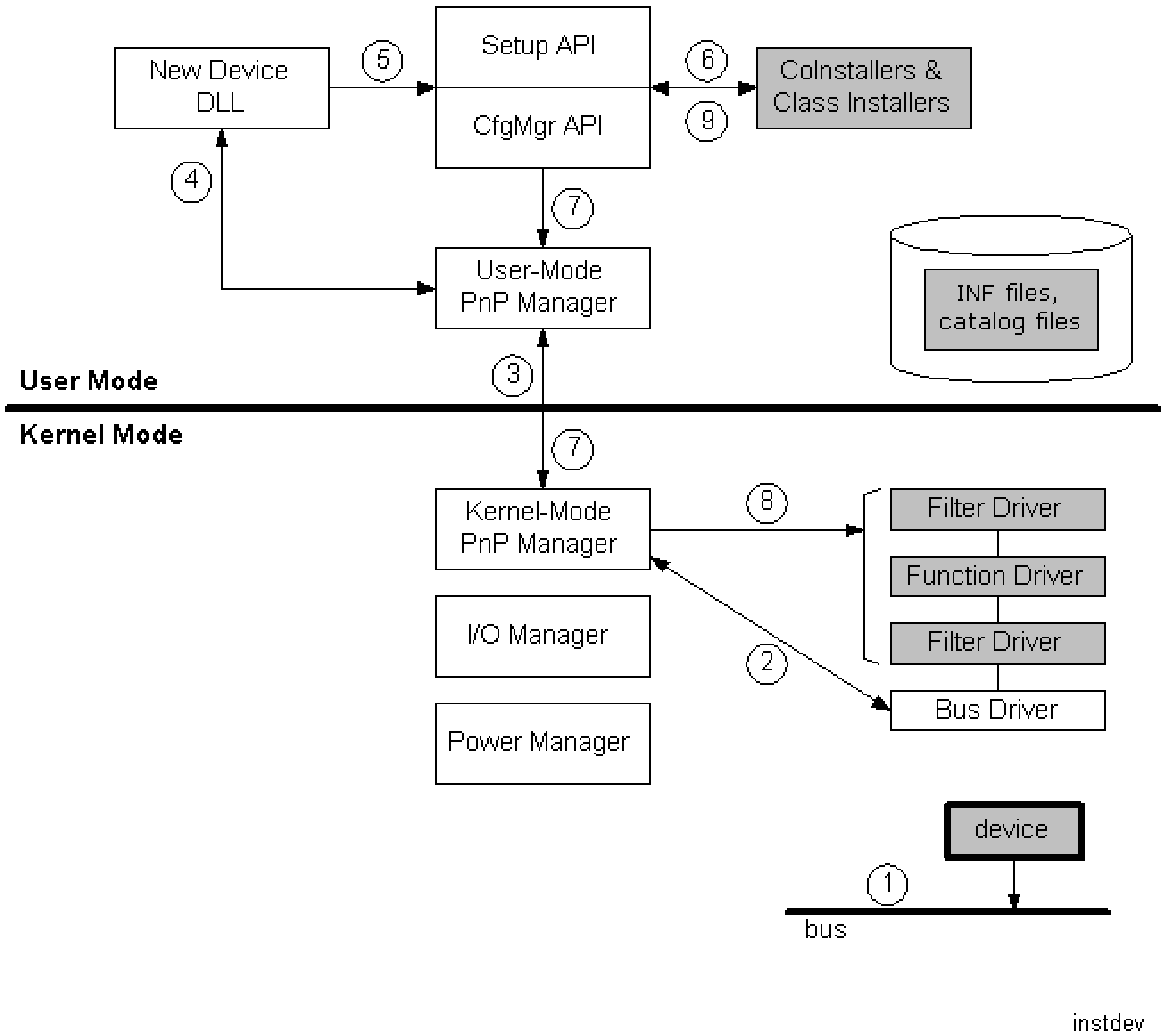
Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Sample PnP Device Installation

To illustrate the interaction of the components involved in device installation, this section steps through the installation of a hot-plug PnP device with an accompanying CD of supporting files.

The following figure shows the software components that participate in installing the PnP device. The shaded components are items that a device vendor might supply.



Sample PnP Device Installation

The following notes correspond to the circled numbers in the figure:

1. The user plugs the device into the computer.

   If the device and the bus support hot-plug notification, the user can plug in a new device while the system is running.

2. The device is enumerated.

   The bus driver, with support from the bus, receives hot-plug notification of the new device. The bus driver notifies the kernel-mode PnP Manager that the list of devices on the bus has changed by calling **IoInvalidateDeviceRelations**. In this case, the change is a new device on the bus.

   The kernel-mode PnP Manager queries the bus driver for a list of devices present on the bus by sending an IRP_MJ_PNP with IRP_MN_QUERY_DEVICE_RELATIONS for **BusRelations**. The bus driver responds to the IRP with current list of devices on the bus. The kernel-mode PnP Manager compares the new list against the previous list and, in this case, determines that there is one new device on the bus.

The kernel-mode PnP Manager sends IRPs to the bus driver to gather information about the new device, such as the device's hardware IDs, compatible IDs, and device capabilities. The IRPs include IRP_MN_QUERY_ID and IRP_MN_QUERY_CAPABILITIES. For more information on the enumeration activities of the kernel-mode PnP Manager, see Adding a PnP Device to a Running System.

3. The kernel-mode PnP Manager notifies the user-mode PnP Manager that there is a device to be installed.

   The user-mode PnP Manager tries to perform a trusted installation (a server-side installation), but it cannot in this example because it needs to display a UI to ask the user for an installation disk for the device.

4. The user-mode PnP Manager creates a new process using *rundll32.exe* and launches *newdev.dll* to install the device.
5. The New Device DLL calls device installation functions (Setup API)and PnP Configuration Manager functions (CfgMgr API) to carry out its installation tasks.

   The New Device DLL calls **SetupDiBuildDriverInfoList** to build the list of possible drivers for the device. In this example, none of the INFs on the system matches the new device, so the returned driver list is empty. The New Device DLL displays the Found New Device wizard. The user supplies the location of the drivers (floppy disk, CD, or Windows® Update). In this example, the driver files are on a CD, so the user loads the CD and clicks **Next** in the wizard. The New Device DLL calls **SetupDiBuildDriverInfoList** again with the new location and receives a list that contains the drivers from the CD.

   For more information on driver selection, see How Setup Selects Drivers.

   (Note that for this example, in which the device's drivers are not included with the operating system, the sequence described is valid only if the user plugs in the new device *after* the operating system has been installed. If a device without "in box" drivers is plugged in before the operating system is installed, the Device Manager will show the device as "unknown".)

6. The class installer and co-installers, if any, can participate in the installation by handling DIF requests.

   For example, the New Device DLL calls **SetupDiCallClassInstaller** to send the DIF_SELECTBESTCOMPATDRV installation request. **SetupDiCallClassInstaller** sends the DIF request to any class co-installers that are registered for the device and to the class installer for the device, if there is one.

   The New Device DLL sends a series of DIF codes, including DIF_SELECTBESTCOMPATDRV, DIF_ALLOW_INSTALL, DIF_INSTALLDEVICEFILES, DIF_REGISTER_COINSTALLERS, DIF_INSTALLINTERFACES, and DIF_INSTALLDEVICE. The class installer and co-installers can participate in servicing these DIF codes.

   Setup uses the **Class** and **ClassGUID** entries in the device's INF **Version** section to determine the device setup class. The setup class determines the class installer and the class co-installers for the device, if any. Device-specific co-installers are listed in the INF *DDInstall*.**Coinstallers** section. For more information on installers, see Writing a Co-installer and Writing a Class Installer.

7. Setup transfers control to kernel mode to load drivers and start the device.

   Once Setup has selected the best driver for the device, copied the appropriate driver files, registered any device-specific co-installers, registered any device interfaces, and so forth, it transfers control to kernel mode to load the drivers and start the device. The appropriate CfgMgr function sends a request to the user-mode PnP Manager, which passes it to the kernel-mode PnP Manager.

8. The PnP Manager loads the appropriate function driver and any optional filter drivers for the device.

   The PnP Manager calls the **DriverEntry** routine for any required driver that is not yet loaded. The PnP Manager then calls the *AddDevice* routine for each driver, starting with lower filter drivers, then the function driver, and, lastly, any upper filter drivers. The PnP Manager assigns resources to the device, if required, and sends an IRP_MN_START_DEVICE to the device's drivers.

9. Installers can supply wizard pages to change device settings.

   Setup sends a DIF_NEWDEVICEWIZARD_FINISHINSTALL request before it displays the standard Finish page. This is an opportunity for installers to add custom pages to change device settings, if necessary.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

How Setup Selects Drivers

Setup selects a driver for each device, unless the device is being used in raw mode. This section describes how Setup makes that selection.

Setup's concept of a driver is really a driver node, which includes all the support for a device, such as any services, device-specific co-installers, registry entries, and so forth. The services for a device can include a set of PnP drivers (a function driver and any upper and lower filter drivers).

Some devices require a vendor-supplied driver that is designed specifically for that device. Other devices can be driven by a vendor-supplied driver that is designed to support a family of devices. Still other devices can be driven by a system-supplied driver that supports all the devices of a given device setup class. To choose a driver for a device, Setup prefers a driver that is a specific match for the device. If it does not find such a driver, it chooses from successively more general drivers.

To locate a driver match, Setup compares the device's hardware IDs and compatible IDs, as reported by the device's parent bus driver, to the hardware IDs and compatible IDs listed in the INF files on the system. Setup creates a list of possible drivers for the device; that is, drivers whose INF *Models* section entries contain an ID that matches one of the device's IDs. (For more information about hardware IDs and compatible IDs, see Device Identification Strings.)

If Setup does not find a match in any of the installed INF files it starts up the Found New Hardware wizard, which prompts the user for a driver.

Setup assigns a "rank" to each possible driver. The rank indicates how well the driver matches the device. The lower the rank number, the better a match the driver is for the device. A rank of zero represents the best match.

For Windows 2000 (and for Windows Me), if more than one driver has the same lowest rank, Setup chooses the driver with the most recent date. A driver's date is specified by an INF **DriverVer** directive. If two drivers have the same rank and the same date, Setup can choose either one, with the following exceptions:

- For Windows 2000 only, Setup sets the date to 00/00/0000 if the driver is not digitally signed (see Driver Signing).
- For some device setup classes, Windows Me might block installation of an unsigned driver if a signed driver is already installed for the device.

For Windows XP and later, some driver packages are considered to be "trusted" while others are "untrusted". All signed driver packages are trusted, and all unsigned driver packages are untrusted.

Note that a driver that is signed for a particular operating system version might not be considered to be signed for another operating system version.

For NT®-based systems, Setup assigns ranks to trusted drivers as follows:

Trusted Driver Rank	Description
0x0-0xFFF	Matched a device's hardware ID with the hardware ID in an INF *Models* section. A rank *N*

	match means that the device's *N*+1 hardware ID matched the INF hardware ID.
0x1000-0x1FFF	Matched one of the device's hardware IDs with a compatible ID in an INF *Models* section.
0x2000-0x2FFF	Matched one of the device's compatible IDs with the hardware ID in an INF *Models* section.
0x3000-0x3FFF	Matched one of the device's compatible IDs with a compatible ID in an INF *Models* section.

Driver ranks in the range 0x0-0xFFF are called "hardware ID matches" because they match hardware ID to hardware ID. These are the best matches. All other ranks are called "compatible ID matches" because the match involves at least one compatible ID. If Setup determines that a driver package for an NT-based system is untrusted, it raises the driver package's rank to a value within the appropriate untrusted rank range.

Setup assigns ranks to untrusted drivers as follows:

Untrusted Driver Rank	**Description**
0x8000-0x8FFF	Hardware IDs that match unsigned drivers through decorated INF sections.
0x9000-0xBFFF	Compatible IDs that match unsigned drivers through decorated INF sections.
0xC000-0xCFFF	Hardware IDs that match unsigned drivers through undecorated INF sections. Probably a Windows 9x/Me-only driver.
0xD000-0xFFFE	Compatible IDs that match unsigned drivers through undecorated INF sections. Probably a Windows 9x/Me-only driver.
0xFFFF	Worst-case match. This is a special value used by components such as co-installers.

When Setup installs a device in the context of an administrative user process (a client-side installation), it starts the Found New Hardware wizard if the driver rank is only a compatible ID match or if its rank is within the untrusted ranges. The wizard asks the user whether he or she has a better driver for the device.

For example, consider a device with two hardware IDs and two compatible IDs. The syntax of the IDs is defined in the glossary. This discussion refers to the IDs using the following shorthand names:

```
HwID_1, HwID_2, CID_1, and CID_2
```

The first hardware ID in a list of hardware IDs is the most precise identification for the device. In this example, that is HwID_1.

For this example, assume there is a signed INF file with a *Models* section that has an entry like the following:

```
DeviceDesc1 = InstallSection1, INF_HWID_1, INF_CID_1
```

Again, the INF_*XXX*_*N* names are shorthand for the hardware IDs and compatible IDs listed in the INF *Models* section.

The following table lists the ranks for the possible matches that Setup can make when comparing the sample device's IDs against the IDs in the INF files:

Device's IDs	**IDs in INF *Models* Section**	
	INF_HwID_1	**INF_CID_1**
HwID_1	Rank 0	Rank 1000-1FFF
HwID_2	Rank 1	Rank 1000-1FFF
CID_1	Rank 2000-2FFF	Rank 3000-3FFF
CID_2	Rank 2000-2FFF	Rank 3000-3FFF

In this example, Setup starts by comparing the device's first hardware ID, HwID_1, with the IDs in the INF files on the system. If Setup matches the HwID_1 string with the first hardware ID listed in an INF *Models* section, it adds that driver node to its list of potential drivers and ranks that match as a "rank 0" match. If Setup matches any other INF *Models* entries with HwID_1, it ranks those matches and adds them to its list of potential drivers.

Next, Setup compares the device's additional hardware IDs, if any, with the IDs in the INF files. If it finds any matches, it ranks them and adds them to its list of potential drivers for this device. Then, it follows the same process for the device's compatible IDs, if any.

After Setup builds its complete list of potential drivers, it selects the best driver for the device (the driver with the lowest rank). If more than one driver has the lowest rank, Setup uses the driver's date, contained in the INF file's **DriverVer** directive, to choose among those drivers.

- For Windows® XP and later (and Windows Me), Setup chooses the driver with the most recent date.
- For Windows 2000, Setup chooses the driver with the most recent date only if the driver packages are signed. Otherwise, Setup ignores **DriverVer** entries and uses a default date of 00/00/0000.

If two drivers have the same rank and the same date, Setup can choose either one.

Class installers and co-installers can participate in driver selection by handling the DIF_SELECTBESTCOMPATDRV request (or the DIF_SELECTDEVICE request for a manually installed device). Setup sends this DIF codes after it builds its list of potential drivers but before it selects the best driver. When handling these DIF codes, a class installer or co-installer can set the DNF_BAD_DRIVER flag on a potential driver if the installer is able to determine that the driver does not support the device. If DNF_BAD_DRIVER is set on a potential driver node, Setup ignores the driver when making its selection.

An installer can also participate in driver selection by changing the rank of a potential driver when the installer handles the DIF_SELECTBESTCOMPATDRV (or DIF_SELECTDEVICE) request. However, the only way an installer should modify driver rank is by setting the rank to 0xFFFF. This rank value indicates that the driver can be used for the device, but only as a last choice option. Installers must not set intermediate values for driver rank because they might interfere with proper driver selection and because the rank ranges are subject to change.

Instead of modifying the driver selection process, a class installer or co-installer can override Setup's entire selection process and directly choose the driver for a device. (See DIF_SELECTBESTCOMPATDRV and DIF_SELECTDEVICE for more information.) However, this is not recommended because Setup's selection algorithms might be enhanced in a future release. When an installer directly selects a driver, Setup's selection operations are terminated.

To illustrate driver selection, consider a sample video device that matches three driver nodes in the INF files on the system. The device's parent bus driver (*pci.sys*) reports the following device IDs:

The device's hardware IDs:
    PCI\VEN_FFFF&DEV_493D&SUBSYS_001C105D&REV_00
    PCI\VEN_FFFF&DEV_493D&SUBSYS_001C105D
    PCI\VEN_FFFF&DEV_493D&CC_030000
    PCI\VEN_FFFF&DEV_493D&CC_0300
The device's compatible IDs:
    PCI\VEN_FFFF&DEV_493D&REV_00
    PCI\VEN_FFFF&DEV_493D
    PCI\VEN_FFFF&CC_030000
    PCI\VEN_FFFF&CC_0300
    PCI\VEN_FFFF
    PCI\CC_030000
    PCI\CC_0300

In this example, Setup finds three driver nodes in the INF files that match the device's IDs. The first driver node in the *Models* section of an INF that matches is the following:

```
%Sample1%  = Sample1.DDInstall, PCI\VEN_FFFF&DEV_493D&CC_0300
```

The device's fourth hardware ID matches the hardware ID in this INF entry, so this is a "rank 3" driver match. Setup adds this driver to its list of potential drivers for the device. This driver supports any video device of this type (a video device with any subsystem vendor ID and any hardware revision number).

The second INF entry that Setup matches for the device has the following driver node in its *Models* section:

```
%Sample2%  = Sample2.DDInstall, \
    PCI\VEN_FFFF&DEV_493D&SUBSYS_001C105D
```

The device's second hardware ID matches the hardware ID in this INF entry, so this is a "rank 1" driver match. Setup adds this driver to its list of potential drivers for the device. This driver supports devices with this specific subsystem vendor ID and with any hardware revision number.

The third INF entry that Setup matches for the device has the following driver node in its *Models* section:

```
%Sample3%  = vga, PCI\CC_0300
```

The device's last compatible ID matches the hardware ID in this INF entry, so this driver match has a rank in the range 2000-2FFF. Setup adds this driver to its list of potential drivers for the device. This driver is a generic driver (the VGA driver) that supports any PCI video device.

For this example, if the co-installers and class installer do not modify the ranks that Setup assigned to the driver matches, Setup chooses the rank 1 driver, %Sample2%.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Device Installations Requiring a Reboot

Device installations should not force the user to reboot the system unless absolutely necessary. The following circumstances are the only ones for which a system reboot should be necessary:

- Installing or removing a non-Plug and Play device.

  For these legacy devices, a user typically must shut down the system before physically adding or removing the device. After the power is turned back on, the system reboots. Note that the device's setup files should not initiate a reboot, regardless of whether the user installs the drivers before or after plugging in the hardware.

- Updating a driver for a system boot device.

  If a device can potentially hold the system's paging, hibernation, or crash dump file, its drivers must service IRP_MN_DEVICE_USAGE_NOTIFICATION requests. The system sends this request before placing one of these files on the disk. If the drivers succeed the request, they must fail any subsequent IRP_MN_QUERY_REMOVE_DEVICE requests. When a driver for the device fails an IRP_MN_QUERY_REMOVE_DEVICE request, the system prompts the user to reboot. Note that the device's setup files should not initiate a reboot.

- Installing a non-WDM filter driver.

  If a filter driver is added to a non-WDM driver stack, the system must be rebooted. In this case, the driver's installer or co-installer should request a reboot (see Initiating System Reboots During Device Installations). Note that adding a filter driver to a WDM driver stack does not require a reboot, unless an underlying device is a system boot device.

Avoiding System Reboots During Device Installations

To avoid system reboots during device installations, use the following rules:

- Never use **Reboot** or **Restart** entries in INF *DDInstall* sections, except for installations specific to Windows 9x/Me.
- Do not use COPYFLG_FORCE_FILE_IN_USE, or COPYFLG_REPLACE_BOOT_FILE flags with INF **CopyFiles** directives, unless absolutely necessary.
- Assign a new file name to each new version of a class installer or co-installer, or a service DLL. This avoids the necessity of a reboot if an older version is in use. (In fact, if a new file name is not used for an updated class installer or class co-installer, these new files will not be used for the installation.)
- To update a device's drivers, follow the rules listed under Updating Driver Files.

Initiating System Reboots During Device Installations

In the rare cases in which it is necessary for the system to be rebooted to complete a device installation, use the following rules:

- During initial installations, a device's installer or co-installer can request a system reboot by setting DI_NEEDREBOOT in the SP_DEVINSTALL_PARAMS structure, which is received along with device installation function codes. (This should not be done unless absolutely necessary.)
- During update installations, a device's installation application can call **UpdateDriverForPlugandPlayDevices**, which determines whether a reboot is necessary.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Installing Devices on 64-Bit Systems

If your device will be installed on both 32-bit platforms and 64-bit platforms, you must use the following steps when creating an installation package:

- Provide both 32-bit and 64-bit compilations of all kernel-mode drivers, device installation applications, device installers, class installers, and co-installers. For more information, see Porting Your Driver to 64-Bit Windows.
- Provide one or more cross-platform INF files that use decorated INF sections to control platform-specific installation behavior.

If you are writing a device installation application, the 32-bit version must be the default version. That is, the 32-bit version should be invoked by Autorun (described in Platform SDK documentation), so that it starts automatically when a user inserts your distribution disk.

The 32-bit version of the application must check the value returned by **UpdateDriverForPlugAndPlayDevices**. If the return value is ERROR_IN_WOW64, the 32-bit application is executing on a 64-bit platform and cannot update "in-box" drivers. Instead, it must call **CreateProcess** (described in Platform SDK documentation) to start the 64-bit version of the application. The 64-bit version can then call **UpdateDriverForPlugAndPlayDevices**, specifying a *FullInfPath* parameter that identifies the location of the 64-bit versions of all files.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Driver Information in the Registry

The operating system and the drivers store information about drivers and devices in the registry. In general, drivers should use the registry to store data that must be maintained across reboots of the system. In addition, drivers can access the registry to obtain information stored by the system or by other programs or drivers. For more information about the registry in general, see the Platform SDK documentation.

The following trees in the registry are of particular interest to driver writers (where **HKLM** represents **HKEY_LOCAL_MACHINE**):

- **HKLM\SYSTEM\CurrentControlSet\Services**
- **HKLM\SYSTEM\CurrentControlSet\Control**
- **HKLM\SYSTEM\CurrentControlSet\Enum**
- **HKLM\SYSTEM\CurrentControlSet\HardwareProfiles**

Drivers and user-mode setup components must access Plug and Play (PnP) keys in the registry using system routines such as **IoGetDeviceProperty** and **SetupDiGetDeviceRegistryProperty**. Drivers must not access these keys directly. The following discussion of registry information is solely for debugging a device installation or configuration problem.

The keys under **HKLM\SYSTEM\CurrentControlSet** are a safe place to preserve data that is vital to your driver because the data is stored in the system hive. The system takes extra precautions to protect the system hive (for example, keeping multiple copies).

The HKLM\SYSTEM\CurrentControlSet\Services Tree

The **HKLM\SYSTEM\CurrentControlSet\Services** registry tree stores information about each service on the system. Each driver has a key of the form **HKLM\SYSTEM\CurrentControlSet\Services\***DriverName*. The PnP Manager passes this path to a driver in the *RegistryPath* parameter when it calls the driver's **DriverEntry** routine. A driver can store global driver-defined data under its key in the **Services** tree. Information stored under this key is available to the driver during its initialization.

The following keys and value entries are of particular interest:

ImagePath

A value entry that specifies the fully qualified path of the driver's image file. Setup creates this value using the required **ServiceBinary** entry in the driver's INF file. This entry is in the *service-install-section* referenced by the driver's INF **AddService** directive. A typical value for this path is *%windir%\system32\Drivers\DriverName*.sys, where *DriverName* is the name of the driver's **Services** key.

Parameters

A key used to store driver-specific data. For some types of drivers, the system expects to find specific value entries. You can add value entries to this subkey using **AddReg** entries in the driver's INF file.

Performance

A key that specifies information for optional performance monitoring. The values under this key specify the name of the driver's performance DLL and the names of certain exported functions in that DLL. You can add value entries to this subkey using **AddReg** entries in the driver's INF file.

The HKLM\SYSTEM\CurrentControlSet\Control Tree

The **HKLM\SYSTEM\CurrentControlSet\Control** registry tree contains information for controlling system startup and some aspects of device configuration. The following subkeys are of particular interest:

Class

Contains information about the device setup classes on the system. There is a subkey for each class, named using the GUID of the setup class. Each subkey contains information about a setup class, such as the class installer (if there is one), registered class upper-filter drivers, registered class lower-filter drivers, and so forth. Each class subkey contains other subkeys known as *software keys* (or, *driver keys*) for each device instance of that class installed in the system.

CoDeviceInstallers

Contains information about the class-specific co-installers that are registered on the system.

DeviceClasses

Contains information about the device interfaces on the system. There is a subkey for each device interface class and entries under those subkeys for each instance of an interface that is registered for the device interface class.

The HKLM\SYSTEM\CurrentControlSet\Enum Tree

The **HKLM\SYSTEM\CurrentControlSet\Enum** registry tree contains information about the devices on the system. The PnP Manager creates a subkey for each device, with a name in the form of **HKLM\SYSTEM\CurrentControlSet\Enum***enumerator**deviceID*. Under each of these keys is a subkey for each device instance present on the system. This subkey, which is known as the device's *hardware key*, has information such as the device description, hardware IDs, compatible IDs, resource requirements, and so forth.

The **Enum** tree is reserved for use by operating system components, and its layout is subject to change. Drivers and user-mode Setup components must use system-supplied functions, such as **IoGetDeviceProperty** and **SetupDiGetDeviceRegistryProperty**, to extract information from this tree. *Drivers and Setup applications must not access the **Enum** tree directly.* You can view the **Enum** tree directly using the registry editor when debugging drivers.

The HKLM\SYSTEM\CurrentControlSet\HardwareProfiles Tree

The **HKLM\SYSTEM\CurrentControlSet\HardwareProfiles** registry tree contains information about the hardware profiles on the system.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

RunOnce Registry Entries

The system supports a registry key, **RunOnce**, that can be used to place commands that the system will execute one time and then delete. Each time the system boots, it executes any commands stored under the **RunOnce** key, then removes them. However, this is not the only time **RunOnce** commands can be executed, so if you place a command under the **RunOnce** key, you cannot predict when it will execute.

For device installations, **RunOnce** registry entries can be created using *add-registry-sections*, which are referenced by INF **AddReg** directives. The registry root and subkey values are as follows:

HKLM,"Software\Microsoft\Windows\CurrentVersion\RunOnce"

Following is an example *add-registry-section* entry that stores a command under the **RunOnce** key:

```
;; WDMAud swenum install

HKLM,%RunOnce%,"WDM_WDMAUD",,\
"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_WDMAUD.DeviceId%,%KSNAME_Filter%,%KSCATE

[Strings]
RunOnce = "SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"
WDM_WDMAUD.DeviceId = "{CD171DE3-69E5-11D2-B56D-0000F8754380}"
KSNAME_Filter = "{9B365890-165F-11D0-A195-0020AFD156E4}"
KSCATEGORY_WDMAUD = "{3E227E76-690D-11D2-8161-0000F8775BF1}"
```

Immediately after a device has been installed, Setup executes all commands stored under the **RunOnce** registry key.

The following rules apply to the use of **RunOnce** entries for device installations:

- **RunOnce** registry entries should only be used for installations of software-only devices that are enumerated by SWENUM, the software device enumerator.
- **RunOnce** entries must consist only of calls to *rundll32.exe*, or else Microsoft will not digitally sign the driver package.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Creating an INF File

Microsoft® Windows drivers must have an INF file in order to be installed. An INF file is a text file that contains all the necessary information about the device(s) and file(s) to be installed, such as driver images, registry information, version information, and so on, to be used by the Setup components.

Windows INF files do not contain installation scripts. The installation procedures are part of a Microsoft Win32® installer application, such as the New Device Wizard and the Add Hardware Wizard, with each INF file acting as a resource.

This chapter contains the following information:

General Guidelines for INF Files

Specifying the Source and Target Locations for Device Files

Creating INF Files for Multiple Platforms and Operating Systems

Creating International INF Files

Specifying Driver Load Order

Tightening File-Open Security in a Device INF File

Accessing INF Files From a Setup Application

See INF File Sections and Directives for a complete description of INF file format.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

General Guidelines for INF Files

INF files have some common parts and follow a single set of syntax rules, but they are also as different as the variety of devices that are supported on a Microsoft® Windows® machine. When writing an INF file, you should consult the following sources of information:

- This chapter and the INF Files Sections and Directives reference material.

- The documentation for your class of device.

  For example, if your device is a printer, see Installing and Configuring Printer Drivers.

- DDK tools for INF files.

  The DDK provides some tools to assist with writing INF files. See the DDK's *tools* subdirectory for information on tools such as GenINF and ChkINF.

- Sample INF files and INF files for similar devices.

  The DDK includes INF files for its sample drivers. Peruse the sample drivers to see if there are INF files for devices similar to your device.

You can create or modify an INF file using any text editor in which you can control the insertion of line breaks. If your INF contains non-ASCII characters, save the file as a Unicode file.

An INF file that ships with the operating system must have a file name of *xxxxxxxx***.inf**, where "*xxxxxxxx*" is no longer than eight characters. The name of an INF file that ships separately from the operating system is not limited to eight characters.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Specifying the Source and Target Locations for Device Files

When Setup processes copy, rename, and delete file statements in an INF file, it determines the source and target locations for the files. To determine these locations, it assesses whether the driver ships with the operating system or separately and examines various INF file sections and entries, including **SourceDisksNames**, **SourceDisksFiles**, **LayoutFile**, **Include**, **Needs**, and **DestinationDirs**.

This section describes how Setup determines source and target locations, and provides guidelines to help you correctly specify these locations, and describes how to copy INF files from one location to another. It contains the following topics:

Source Media for INFs

Target Media for INFs

Copying INFs

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Source Media for INFs

The methods you should use to specify source media for device files depend on whether your INFs ship separately from the operating system or are included with the operating system.

Source Media for INFs that Ship Separately from the Operating System

INFs for drivers that are delivered separately from the operating system specify where the files are located using **SourceDisksNames** and **SourceDisksFiles** sections. If such an INF contains **Include** and **Needs** entries in the *DDInstall* section, those entries specify additional possible source locations.

If an INF has **SourceDisksNames** and **SourceDisksFiles** sections and no **Include** entries, the **SourceDisksNames** and **SourceDisksFiles** sections must list all the source media and source files in the kit except for the catalog and INF files. (Microsoft® Windows® 98 requires that catalog files be listed in the **SourceDisksFiles** section; Windows 2000 and later versions ignore catalog file entries.)

Catalog files must be in the same location as the INF file. Catalog files must not be compressed. If the installation media includes multiple disks, then *a separate copy of the INF and catalog files must be included on every disk*. This is because the INF and catalog files must continue to be accessible throughout the entire installation.

Source Media for INFs that Ship with the Operating System

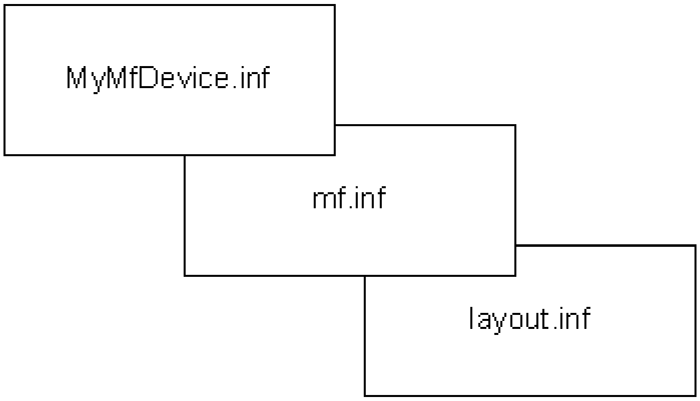
If the files to support the device are included with the operating system, the INF must specify a **LayoutFile** entry in the **Version** section of the file. Such an entry specifies where the files reside on the operating system media. An INF that specifies a **LayoutFile** entry must not include **SourceDisksNames** and **SourceDisksFiles** sections.

Only system-supplied INFs should reference *layout.inf* directly. OEM and IHV INF files that ship with the operating system can reference a common install section in a system-supplied INF (such as *mf.inf*), and the system-supplied INF references the *layout.inf* file.

Source Media and INFs that Contain Include and Needs Entries

If an INF has **SourceDisksNames** and **SourceDisksFiles** sections and **Include** and **Needs** entries, Setup uses the main INF file plus any of the included INF files to locate source media. It is especially important with included files to be as precise as possible when specifying source media and source file locations.

Consider the hierarchy of included INFs shown in the following figure.



inf-hier

Sample Hierarchy of Included INF Files

The figure shows an INF for a multifunction device, named *MyMfDevice.inf*, which includes the system-supplied *mf.inf* file. In turn, the system-supplied *mf.inf* includes the system-supplied *layout.inf* file. When Setup searches for source media from which to copy a file referenced in *MyMfDevice.inf*, it looks for a **SourceDisksFiles** section in *MyMfDevice.inf* and in any included INF files that reference the file to be copied. Setup searches *MyMfDevice.inf* first, but it does not guarantee the order in which it searches the included INFs.

Decorated **SourceDisksFiles** sections take precedence over undecorated sections, even if the decorated INF section is in an included file. For example, for the INFs shown in the previous figure, if *layout.inf* contains a **[SourceDisksFiles.x86]** section and *MyMfDevice.inf* contains only an undecorated **[SourceDisksFiles]** section, Setup uses the decorated section from *layout.inf* first when installing on an x86 machine. Therefore, an INF that includes other INFs should contain section names that use platform extensions.

Typically, a vendor-supplied INF should specify the location of the files in its driver package and should not cause Setup to search included INFs for file locations. In other words, a vendor INF that copies files should specify both a **SourceDisksNames** and a **SourceDisksFiles** section, those sections should be decorated with platform extensions, and those sections should contain information for all the files directly copied by the INF.

It is helpful if the vendor file names are as vendor-specific as possible, unless the installation is explicitly replacing a vendor-supplied file that shipped with the operating system.

Note the following rules when using **Includes** and **Needs** entries:

- **Includes** and **Needs** entries can only be used to specify system-supplied INF files.
- **Needs** entries cannot be nested. That is, a section identified by a **Needs** entry cannot itself contain a **Needs** entry.

An INF file that makes use of a section in another INF file by means of the **Include** and **Needs** entries might have to make use of an accompanying section to maintain consistency. For instance, if an INF file references the installation section (*DDInstall*) of another INF file in order to install the driver, it must reference an INF *DDInstall***.Services** section to install the accompanying service. Such an INF file might have the following sections:

```
[DDInstall]
Include = AnotherINFFile.inf
Needs = AnotherINFFileDDInstall

[DDInstall.Services]
Include = AnotherINFFile.inf
Needs = AnotherINFFileDDInstall.Services
```

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Target Media for INFs

An INF file specifies the target location for device files with a **DestinationDirs** section. This section should always be specified in the same INF file as the section with the copy, rename, or delete statements.

A **DestinationDirs** section should include a **DefaultDestDir** entry.

If an INF has copy, rename, or delete sections but no **DestinationDirs** section and the INF includes other INFs, Setup searches the included INFs for target location information. However, the order in which Setup searches the included files is not predictable. Therefore, there is a risk that Setup will, for example, copy files to a location not intended by the INF writer. To avoid such confusion, always specify a **DestinationDirs** section in an INF that contains copy, rename, or delete sections. The **DestinationDirs** section should include at least a **DefaultDestDir** entry and can list sections in the INF that copy, rename, or delete files.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Copying INFs

It is sometimes necessary to copy INF files during device installation, so that Setup can find them without repetitively displaying user prompts. For example, the base INF file for a multifunction device might copy the INF files for the device's individual functions, so that Setup can find these INF files without prompting the user each time it installs one of the device's functions.

To copy INF files, you can use either of the following techniques:

- A co-installer or setup application can call **SetupCopyOEMInf** (described in Platform SDK documentation). This DDK includes source code for a co-installer, *cocpyinf.dll*, which uses this technique. The source code resides under the DDK's */src* directory.
- (*Windows XP and later.*) An INF file can include the INF **CopyINF** directive.

  (Note that this DDK also includes a redistributable, compiled version of *cocpyinf.dll*. If you use this co-installer for your device, the **CopyINF** directive can then be used on Windows 2000. The redistributable version of *copyinf.dll* resides under the DDK's *\tools* directory.)

Both of these methods will:

- Install the appropriate catalog file, if it exists, along with the INF file.
- Give the INF file a unique name so that it does not overwrite any other INF files, and will not be overwritten by other INF files.
- Retain the path to the source medium from which device files are to be copied.
- Ensure compatibility with future versions of Windows.

On Windows 2000 and later, installation software must never copy INF files directly into a system's *%windir%/inf* directory. *Copying INF files by using techniques not described in this section will invalidate a driver's digital signature.*

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Creating INF Files for Multiple Platforms and Operating Systems

By using system-defined extensions to INF file sections and directives, you can create a single INF file for cross-platform installations and/or for dual-OS installations. The extensions enable you to create *decorated* section names. For INF *DDInstall* sections and related directives, these *decorations* (**.ntx86**, **.ntia64**, or **.nt**) specify which sections and directives are relevant to each platform and operating system. You can create an INF file that installs a device only on x86 platforms, only on ia64 platforms, on all platforms supported by Windows NT4.0 and later, on all platforms supported by Windows 9x/Me, or on all platforms for all operating systems.

For example, if your device supports Windows 2000 and later platforms, and Windows 9x/Me platforms, you would create an INF file with two or more "parallel" *DDInstall* sections that have the following extensions to control what is installed on each platform and/or for each operating system:

- The **[***install-section-name***.ntx86]** extension contains instructions for installing the device or set of device-compatible models on all Windows NT4.0 and later x86-based platforms.
- The **[***install-section-name***.ntia64]** extension contains instructions for installing the device or set of device-compatible models on all ia64-based platforms.
- The **[***install-section-name***.nt]** extension contains instructions for installing the device or set of device-compatible models on all Windows NT4.0 and later platforms.
- The **[***install-section-name***]** extension contains Windows 9x/Me-specific instructions.

  Undecorated sections are also fallback sections for Windows NT4.0 and later platforms. Setup uses undecorated sections on Windows NT4.0 and later platforms if it cannot locate the appropriate decorated sections. However, to prevent these sections from being used on Windows 9x/Me by mistake, you should decorate any Windows NT4.0-specific or later *DDInstall* and related sections with the **.ntx86**, **.ntia64**, or **.nt** extensions.

Windows Setup processes these *DDInstall* sections as follows:

- On Windows 9x/Me, Setup processes the undecorated section and ignores any sections with **.nt*** platform extensions.
- On Windows NT4.0 and later:
  1. Setup checks for install sections with the platform-specific extensions, such as **.ntx86**. If one is present, Setup processes the section appropriate for the current platform. Setup checks for platform-specific install sections in the INF being processed and in any included INF files (that is, any INFs included with **Include** entries).
  2. If there are no specific platform extensions, Setup checks for a *install-section-name***.nt** section in the INF or any included INFs. If one is present, Setup processes the *install-section-name***.nt** section.
  3. If there is no *install-section-name***.nt** section, Setup processes the undecorated section.

An INF file containing INF *DDInstall* sections with platform extensions can also contain additional per-device sections,such as the required *DDInstall***.Services** and optional *DDInstall***.HW**, *DDInstall***.CoInstallers**, *DDInstall***.LogConfigOverride**, and/or *DDInstall***.Interfaces** sections. In dual-OS and/or cross-platform INF files, you should specify the appropriate extension immediately following the INF-writer-defined section name as, for example, *install-section-name***.ntx86.HW**.

If a dual-OS INF file contains decorated *install-section-name***.nt**, *install-section-name***.ntx86**, or *install-section-name***.ntia64** sections, it must also have additional parallel decorated and undecorated per-device sections. That is, if the INF file has both *install-section-name* and *install-section-name***.nt** sections and it has a *DDInstall***.HW** section, it also must have a parallel *install-section-name***.nt.HW** section (if the device or driver requires a **.HW** section for Windows NT4.0 and later).

The INF File Sections and Directives documentation shows these extensions as part of the formal syntax statements in the appropriate section references as, for example:

[*install-section-name***.HW]** |
[*install-section-name***.nt.HW]** |
[*install-section-name***.ntx86.HW]** |
[*install-section-name***.ntia64.HW]**

Such a formal syntax statement indicates that these extensions are valid alternatives to the basic section. It does *not* indicate that any INF with a *install-section-name***.nt.HW** section must also have every other platform-specific *install-section-name***.nt***xxx***.HW** section. You can use any subset of these extensions to specify the decorated sections required for a particular cross-platform or dual-OS installation.

INF files that contain *install-section-name* platform extensions can also include platform extensions with their INF **SourceDisksNames** section and INF **SourceDisksFiles** section entries, to specify installation file locations in a platform-specific manner.

For Windows XP and later, you can supply INF *Models* sections that are specific to various versions of the operating system. These version-specific *Models* sections are identified using the *TargetOSVersion* decoration in an INF **Manufacturer** section.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Creating International INF Files

Creating installations for international markets requires providing locale-specific INF files and, possibly, locale-specific driver files.

An INF file that will be used in an international market should use **%***strkey***%** tokens for all user-viewable text. The strings are defined in an INF **Strings** section, which is typically at the end of the INF file.

Locale-Specific INF Files

You can create a single INF file that supports several locales, or you can create a separate INF file for each locale.

- To create a single international INF file, you should include a set of locale-specific **Strings.***LanguageID* sections, as described in the reference page for the INF **Strings** section. Use this technique if you intend to supply a single installation medium for all international markets.

  For installations on NT-based operating systems, this is the recommended method for supporting international markets.

- To create a separate INF file for each locale, start with a main INF file containing all the necessary sections and entries, except for the **Strings** section. Then create a second set of files, where each file contains just the **Strings** section for a supported locale. Concatenate the main file with each strings file to generate the locale-specific INF files.

  For installations on Windows 9x/Me, this is the recommended method for supporting international markets, because Windows 9x/Me does not support **Strings**.*LanguageID* sections.

  For installations on NT-based operating systems, use this technique *only* if you intend to supply a separate installation medium for each international market. You cannot provide multiple versions of an INF file, for a particular NT-based OS version, on a single installation medium because Setup cannot determine which INF file to use.

Locale-Specific Versions of Driver Files

If you need to provide locale-specific versions of driver files for NT-based operating systems, place the files in separate, locale-specific subdirectories of the distribution medium, such as /*English* and /*German*. In your INF file, do the following:

- Within the INF **SourceDisksFiles** section, specify locale-specific subdirectories using a string key token such as **%LocaleSubDir%**.
- Provide separate INF **Strings** sections for each language, and define the appropriate subdirectory name string in each section, as follows:

```
[Strings]              ; No language ID implies English
LocaleSubDir="English"
[Strings.0407]         ; 0407 is the language ID for German
LocaleSubDir="German"
```

Creating Unicode INF Files

If an INF contains characters outside the ASCII range (that is, outside the range of 0-127), the INF should be in Unicode format. One way to create a Unicode INF file is to save it in Unicode format from an application such as Notepad. If the INF is not in Unicode format, Setup uses the current locale to translate characters. If the INF is in Unicode format, Setup uses the full Unicode character set. (Windows 9x/Me Setup ignores Unicode-formatted INF files.)

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Specifying Driver Load Order

For most devices, the physical hierarchy of the devices on a machine determines the order in which Setup and the PnP Manager load drivers. Setup and the PnP Manager configure devices starting with the system root device, and then they configure the child devices of the root device (for example, a PCI adapter), the children of those devices, and so on. The PnP Manager loads the drivers for each device as the device is configured, if the drivers were not previously loaded for another device.

Settings in the INF file can influence driver load order. This topic describes the relevant values that vendors should specify in the *service-install-section* referenced by a driver's INF **AddService** directive. Specifically, this topic discusses the **StartType**, **LoadOrderGroup**, and **Dependencies** entries.

Drivers should follow these rules for specifying **StartType**:

- PnP driver

  A PnP driver should have a start type of SERVICE_DEMAND_START (0x3), specifying that the PnP Manager can load the driver whenever the PnP Manager finds a device that the driver services.

- Driver for a device required to boot the machine

  If a device is required to boot the machine, the drivers for the device should have a start type of SERVICE_BOOT_START (0x0).

- Non-boot driver that detects device(s) that are not PnP-enumerable

  For a device that is not PnP-enumerable, a driver reports the device to the PnP Manager by calling **IoReportDetectedDevice**. Such a driver should have the start type SERVICE_SYSTEM_START (0x01) so Setup will load the driver during system initialization.

  Only drivers that report nonPnP hardware should set this start type. If a driver services both PnP and nonPnP devices, it should set this start type.

- NonPnP driver that must be started by the Service Control Manager

  Such a driver should have the start type SERVICE_AUTO_START (0x02). PnP drivers must not set this start type.

A PnP driver should be written so that it can be loaded whenever Setup configures a device that the driver services. Conversely, a driver should be able to be unloaded any time the PnP Manager determines that there are no longer devices present that the driver services. The only driver load orderings that PnP drivers should depend on are the following:

1. The drivers for a child device can depend on the fact that the drivers for the parent device have been loaded.
2. A driver in the device stack can depend on the fact that any drivers below it have been loaded. For example, the

function driver can be certain that any lower-filter drivers have been loaded.

Note, however, that a driver in the device stack cannot depend on being loaded sequentially after a device's lower drivers, because the driver might have been loaded previously when another device was configured.

Filter drivers within a filter group cannot predict their load ordering. For example, if a device has three registered upper-filter drivers, those three drivers will all be loaded after the function driver but could be loaded in any order within their upper-filter group.

If a driver has an explicit load-order dependency on another driver, that dependency should be implemented through a parent/child relationship. A driver for a child device can depend on the drivers for the parent device being loaded before the child drivers are loaded.

To reinforce the importance of setting the correct **StartType** value, the following list describes how Setup and the PnP Manager use the **StartType** entries in INF files:

1. On system boot, the operating system loader loads drivers of type SERVICE_BOOT_START before it transfers control to the kernel. These drivers are in memory when the kernel gets control.

   Boot-start drivers can use INF **LoadOrderGroup** entries to order their loading. (Boot-start drivers are loaded before most of the devices are configured, so their load order cannot be determined by device hierarchy.) The operating system ignores INF **Dependencies** entries for boot-start drivers.

2. The PnP Manager calls the **DriverEntry** routines of the SERVICE_BOOT_START drivers so the drivers can service the boot devices.

   If a boot device has child devices, those devices are enumerated. The child devices are configured and started if their drivers are also boot-start drivers. If a device's drivers are not all boot-start drivers, the PnP Manager creates a devnode for the device but does not start the device yet.

3. After all the boot drivers have loaded and the boot devices are started, the PnP Manager configures the rest of the PnP devices and loads their drivers.

   The PnP Manager walks the device tree and loads the drivers for the devnodes that are not yet started (that is, any nonstarted devnodes from the previous step). As each device starts, the PnP Manager enumerates the children of the device, if any.

   As it configures these devices, the PnP Manager loads the drivers for the devices, *regardless* of the drivers' **StartType** values, and starts the devices. Many of these drivers are SERVICE_DEMAND_START drivers, but they can have any **StartType**.

   The PnP Manager ignores registry entries that were created as a result of INF **Dependencies** entries and **LoadOrderGroup** entries for drivers that it loads in this step. The load ordering is based on the physical device hierarchy.

   At the end of this step, all of the devices have been configured, except devices that are not PnP-enumerable and the descendants of those devices. (The descendants might or might not be PnP-enumerable.)

4. The PnP Manager loads drivers of **StartType** SERVICE_SYSTEM_START that are not yet loaded.

   These drivers detect and report their nonPnP devices. The PnP Manager processes registry entries that are the result of INF **LoadOrderGroup** entries for these drivers. It ignores registry entries that were created as a result of INF **Dependencies** entries for these drivers.

5. The Service Control Manager loads drivers of **StartType** SERVICE_AUTO_START that are not yet loaded.

The Service Control Manager processes the service database information with respect to the services' **DependOnGroup** and **DependOnServices**. This information is from **Dependencies** entries in INF **AddService** entries. Note that the **Dependencies** information is only processed for nonPnP drivers because any necessary PnP drivers were loaded in an earlier step of system startup. The Service Control Manager ignores INF **LoadOrderGroup** information.

See the Platform SDK documentation for more information on the Service Control Manager.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Tightening File-Open Security in a Device INF File

For Microsoft® Windows® 2000 and later, Microsoft tightened file-open security in the class installer INFs for certain device classes, including CDROM, DiskDrive, FDC, FloppyDisk, HDC, and SCSIAdapter.

If you are unsure whether the class installer for your device has tightened security on file opens, you should tighten security by using the device's INF file to assign a value to the **DeviceCharacteristics** value name in the registry. Do this within an *add-registry-section*, which is specified using the INF **AddReg** directive. The *add-registry-section* is referenced in an INF *DDInstall* section and has the following syntax:

```
[DDInstall]
...
AddReg = Xxx_AddReg
...

[Xxx_AddReg]
...
HKR,,DeviceCharacteristics,0x10001,characteristics
```

The *characteristics* value is the numeric value of the characteristics to assign to the device. This value is the result of ORing one or more FILE_* file characteristics values as defined in *wdm.h* or *ntddk.h*. Setting the FILE_DEVICE_SECURE_OPEN characteristic, which has a numeric value of 0x100, directs the I/O Manager to perform security checks on all open requests for the device. The FILE_DEVICE_SECURE_OPEN characteristic is supported on Windows 2000 and later platforms but not on Win9x/Me platforms.

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.

During device installation, the Plug and Play Manager determines the device characteristics for a device object as follows:

1. It retrieves the characteristics settings for the device's class, if any, as specified in the class-installer INF.
2. It retrieves the characteristics for the device, if any, as specified in the device INF. Device-specific settings override class settings.
3. It ORs the resulting INF-specified settings with those set in the device object by the driver, either directly or through **IoCreateDevice**.
4. It ORs the device characteristics settings of the FDO and any filter DOs and propagates these settings to the FDO and filter DOs. If the device is in raw mode, and therefore has no FDO, the Plug and Play Manager propagates the device characteristics across the PDO and any filter DOs.

The above steps ensure, for example, that if one driver for a device sets FILE_DEVICE_SECURE_OPEN, that characteristic is propagated to all the device objects for drivers servicing the device.

The Plug and Play Manager ORs and propagates these characteristics after it has called the AddDevice routines for the device's drivers and after the drivers have created their device objects and attached them to the device stack, but before the Plug and Play Manager sends the IRP_MN_START_DEVICE request.

The Plug and Play Manager also propagates the following device characteristics when it propagates FILE_DEVICE_SECURE_OPEN: FILE_REMOVABLE_MEDIA, FILE_READ_ONLY_DEVICE, and FILE_FLOPPY_DISKETTE.

An INF file that has a **ClassInstall32** section can set the FILE_DEVICE_SECURE_OPEN characteristic for a setup class of devices. See Tightening File-Open Security in a Class Installer INF File.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Accessing INF Files From a Setup Application

Most INF files are used by system class installers and other setup applications. This section describes common operations that vendor-supplied setup applications can be carry out on INF files, using the general Setup functions. For complete information on how to use these functions, see the Platform SDK documentation.

This section contains the following topics:

Opening and Closing an INF File

Retrieving Information from an INF File

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Opening and Closing an INF File

Before an application can access the information in an INF file, it must open the file by calling **SetupOpenInfFile**. This function returns a handle to the INF file.

If you do not know the name of the INF file that you need to open, use **SetupGetInfFileList** to obtain a list of all the INF files in a directory.

Once an application opens an INF file, it can append additional INF files to the opened file using **SetupOpenAppendInfFile**. When subsequent Setup functions reference an open INF file, they will also be able to access any information stored in any appended files.

If no INF file is specified when calling **SetupOpenAppendInfFile**, this function appends the file specified by the **LayoutFile** entry in the INF **Version** section of the INF file opened during the call to **SetupOpenInfFile**.

When the information in the INF file is no longer needed, the application should call **SetupCloseInfFile** to release resources allocated during the call to **SetupOpenInfFile**.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Retrieving Information from an INF File

Once you have a handle to an INF file, you can retrieve information from it in a variety of ways. Functions such as **SetupGetInfInformation**, **SetupQueryInfFileInformation**, and **SetupQueryInfVersionInformation** retrieve information about the specified INF file.

Other functions, such as **SetupGetSourceInfo** and **SetupGetTargetPath**, acquire information about the source files and target directories.

Functions such as **SetupGetLineText** and **SetupGetStringField** enable you to directly access information stored in a line or field of an INF file. These functions are used internally by the higher-level Setup functions but are available if you need to directly access information at the line or field level.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Writing a Co-installer

A co-installer is a Microsoft® Win32® DLL that assists in device installation. Co-installers are called by Setup API as "helpers" for Class Installers. For example, a vendor can provide a co-installer to write device-specific information to the registry that cannot be handled by the INF file.

This section describes:

Co-installer Operation

Co-installer Interface

Co-installer Functionality

Handling DIF Codes

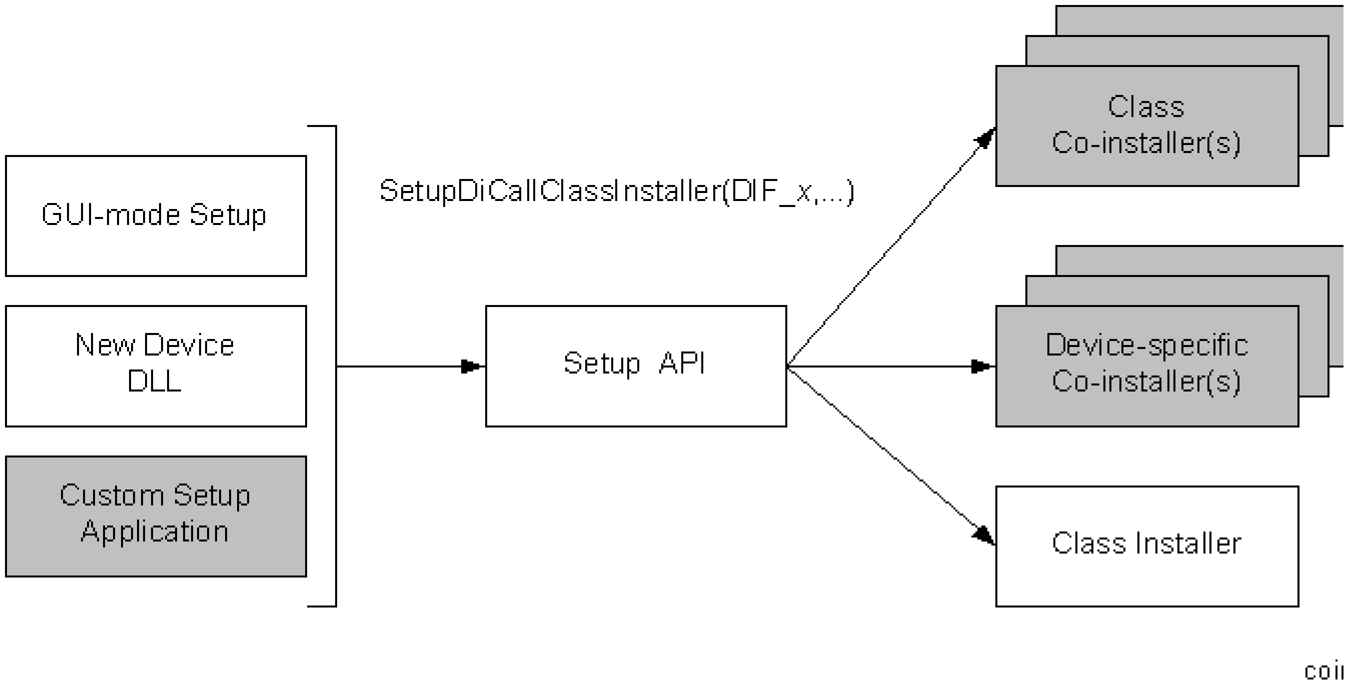
Registering a Co-installer

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Co-installer Operation

Co-installers are called by Setup API as shown in the following figure.



Co-installers Participate in Device Installation

The shaded boxes represent components that IHVs and OEMs can provide. The other components are supplied by the OS. See Device Installation Overview for more information on Setup components.

Co-installers can be device-specific or class-specific. Setup API calls a *device-specific co-installer* only when installing the device for which the co-installer is registered. The OS and vendors can register zero or more device-specific co-installers for a device. Setup API calls a *class co-installer* when installing any device of the device setup class for which the co-installer is registered. The OS and vendors can register one or more *class co-installers* for a device setup class. In addition, a class co-installer can be registered for one or more setup classes.

GUI-mode Setup, the New Device DLL, and Custom Device Installation Applications install devices by calling **SetupDiCallClassInstaller** with device installation function codes (DIF codes). For each DIF request, **SetupDiCallClassInstaller** calls any class co-installers registered for the device's setup class, any device co-installers registered for the specific device, and then the Class Installer supplied by the system for the device's setup class, if there is one.

Custom device installation applications must call **SetupDiCallClassInstaller** rather than calling a co-installer or class installer directly. This function ensures that all registered co-installers are called appropriately.

Class co-installers are typically registered prior to device installation, and device-specific co-installers are registered as part of the device's installation. Class co-installers are therefore typically added to the co-installer list when it is first built and are called for all DIF requests during device installation.

Device-specific co-installers are added to the co-installer list after a DIF_REGISTER_COINSTALLERS request has been completed for the device (or **SetupDiRegisterCoDeviceInstallers** has been called). Device-specific co-installers do not participate in the following DIF requests:

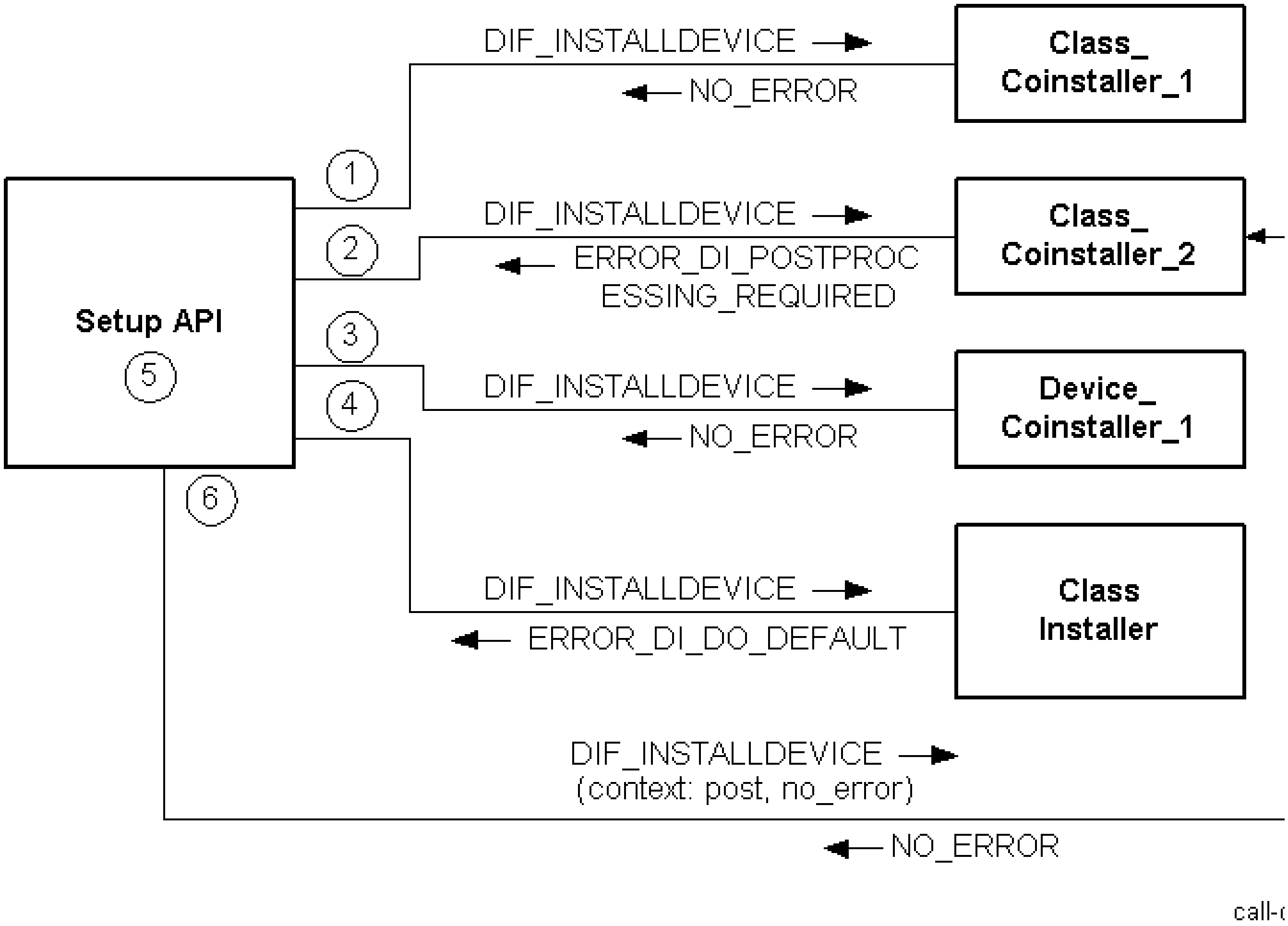
DIF_ALLOW_INSTALL
DIF_INSTALLDEVICEFILES
DIF_SELECTBESTCOMPATDRV

Only a class co-installer (not a device-specific co-installer) can respond to the following DIF requests:

DIF_DETECT
DIF_FIRSTTIMESETUP
DIF_NEWDEVICEWIZARD_PRESELECT
DIF_NEWDEVICEWIZARD_SELECT
DIF_NEWDEVICEWIZARD_PREANALYZE
DIF_NEWDEVICEWIZARD_POSTANALYZE

A device co-installer is not appropriate in these contexts, either because a particular device has not yet been identified or at this early stage of installation the device co-installers have not yet been registered.

The following figure shows the order in which **SetupDiCallClassInstaller** calls co-installers and a Class Installer after any device-specific co-installers have been registered.



Example of Calling Co-installers for DIF Request Processing and PostProcessing

In the example illustrated by the previous figure, two class co-installers are registered for this device's setup class and one device-specific co-installer is registered for the device. The following steps correspond to the circled numbers in the previous figure:

1. **SetupDiCallClassInstaller** calls the first class co-installer, specifying a DIF code that indicates the install request being processed (DIF_INSTALLDEVICE, in this example). The co-installer has the option of participating in the install request. In this example, the first registered class co-installer returns NO_ERROR.
2. **SetupDiCallClassInstaller**, in turn, calls any additional registered class co-installers. In this example, the second

class co-installer returns ERROR_DI_POSTPROCESSING_REQUIRED, which requests that the co-installer be called later for postprocessing.

3. **SetupDiCallClassInstaller** calls any registered device-specific co-installers.
4. After all registered co-installers have been called, **SetupDiCallClassInstaller** calls the system-supplied Class Installer, if there is one for the device's setup class. In this example, the class installer returns ERROR_DI_DO_DEFAULT, which is a typical return value for class installers.
5. **SetupDiCallClassInstaller** calls the default handler for the installation request, if there is one. DIF_INSTALLDEVICE has a default handler, **SetupDiInstallDevice**, which is part of Setup API.
6. **SetupDiCallClassInstaller** calls any co-installers that requested postprocessing. In this example, the second class co-installer requested postprocessing.

Co-installer post-processing is similar to driver *IoCompletion* routines, except that the co-installer is called a second time at its single entry point. When **SetupDiCallClassInstaller** calls a co-installer for post-processing, it sets *PostProcessing* to TRUE and *InstallResult* to the appropriate value in the *Context* parameter. In this example, *Context.InstallResult* is NO_ERROR because the default handler executed successfully.

For postprocessing, **SetupDiCallClassInstaller** calls co-installers in reverse order. If all the co-installers in the previous figure had returned ERROR_DI_POSTPROCESSING_REQUIRED, **SetupDiCallClassInstaller** would call Device_Coinstaller_1 first for postprocessing, followed by Class_Coinstaller_2, and then Class_Coinstaller_1. Class Installers do not request postprocessing; only co-installers do.

A co-installer that requests postprocessing is called even if a previous co-installer failed the install request.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Co-installer Interface

A co-installer's interface consists of an exported entry point function and an associated data structure.

Co-installer Entry Point

A co-installer must export an entry point function that has the following prototype:

```
typedef
DWORD
(CALLBACK* COINSTALLER_PROC) (
    IN DI_FUNCTION  InstallFunction,
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL,
    IN OUT PCOINSTALLER_CONTEXT_DATA  Context
    );
```

InstallFunction specifies the device installation request being processed, in which the co-installer has the option of participating. For example, DIF_INSTALLDEVICE. See Device Installation Function Codes for documentation on DIF codes.

DeviceInfoSet supplies a handle to a device information set.

DeviceInfoData optionally identifies a device that is the target of the device installation request. If this parameter is non-NULL, it identifies a device information element in the device information set. *DeviceInfoData* is non-NULL when **SetupDiCallClassInstaller** calls a device-specific co-installer. A class-specific co-installer can be called with a DIF request that has a NULL *DeviceInfoData*, such as DIF_DETECT or DIF_FIRSTTIMESETUP.

Context points to a COINSTALLER_CONTEXT_DATA structure, described below.

A device co-installer returns one of the following values:

NO_ERROR
: The co-installer performed the appropriate actions for the specified *InstallFunction*, or the co-installer determined that it didn't need to perform any actions for the request.

ERROR_DI_POSTPROCESSING_REQUIRED
: The co-installer performed any appropriate actions for the specified *InstallFunction* and is requesting to be called again after the Class Installer has processed the request.

A Win32 error
: The co-installer encountered an error.

A co-installer does not set a return status of ERROR_DI_DO_DEFAULT. This status can be used only by a Class Installer. If a co-installer returns this status, **SetupDiCallClassInstaller** will not process the DIF_*Xxx* request properly. A co-installer might *propagate* a return status of ERROR_DI_DO_DEFAULT in its postprocessing pass, but it never *sets* this value.

COINSTALLER_CONTEXT_DATA

COINSTALLER_CONTEXT_DATA is a co-installer-specific context structure describing an installation request. The format of the structure is:

```
typedef struct
_COINSTALLER_CONTEXT_DATA {
    BOOLEAN  PostProcessing;
    DWORD  InstallResult;
    PVOID  PrivateData;
} COINSTALLER_CONTEXT_DATA, *PCOINSTALLER_CONTEXT_DATA;
```

PostProcessing is TRUE when a co-installer is called after the appropriate Class Installer, if any, has processed the *InstallFunction*. *PostProcessing* is read-only to the co-installer.

If *PostProcessing* is FALSE, *InstallResult* is not relevant. If *PostProcessing* is TRUE, *InstallResult* is the current status of the install request. This value is NO_ERROR or an error status returned by the previous component called for this install request. A co-installer can propagate the status by returning this value for its function return, or it can return another status. *InstallResult* is read-only to the co-installer.

PrivateData points to a co-installer-allocated buffer. If a co-installer sets this pointer and requests postprocessing, **SetupDiCallClassInstaller** passes the pointer to the co-installer when it calls the co-installer for postprocessing.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Co-installer Functionality

A co-installer is a user-mode Win32 DLL that typically writes additional configuration information to the registry, or performs other installation tasks requiring information that is not available when an INF is written.

A co-installer might do some or all of the following:

- Handle one or more of the device installation function codes (DIF codes) received by the co-installer entry point function.

- Perform operations before the associated class or device installer is called, after the class or device installer is called, or both, as described in Co-installer Operation.
- Provide "finish install" pages (in response to DIF_NEWDEVICEWIZARD_FINISHINSTALL) to obtain device parameters that are required so the device can operate, or to ask if device-specific applications should be installed.
- Provide device property pages, which are displayed by the Device Manager so users can modify device parameters.

When called for post-processing, a co-installer must check the **InstallResult** member of the COINSTALLER_CONTEXT_DATA structure. If its value is not NO_ERROR, the co-installer must do any necessary clean up operations and return an appropriate value for **InstallResult**.

Co-installers are sometimes used to obtain information from the user. Such information might include additional device parameters (such as a modem's dialing properties), or whether the user wants device-specific applications installed. Co-installers can create user interfaces by providing "finish install" pages and by providing device property pages. No other form of user interface is allowed. Setup displays "finish install" pages at the end of the installation (within the Found New Hardware, Hardware Update, or Add Hardware Wizard, which require administrator privilege). The Device Manager displays property pages, and allows users with administrator privilege to modify parameters displayed on these pages.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Handling DIF Codes

Setup applications send device installation function codes (DIF codes) to installers by calling **SetupDiCallClassInstaller**. This function, in turn, calls the installer's entry point function. For a description of installer entry points, see:

Co-installer Interface

Class Installer Interface

The reference page for each DIF code contains the following sections:

When Sent
Describes the typical times when, and reasons why, a Setup application sends this DIF request.

Who Handles
Specifies which installers are allowed to handle this request. The installers include class installers, class co-installers (setup-class-wide co-installers), and device co-installers (device-specific co-installers).

Installer Input
Besides the DIF code, **SetupDiCallClassInstaller** supplies additional information relevant to the particular request. See the reference page for each DIF code for details on the information that is supplied with each request. The following list contains a general description of the additional input parameters, and lists the device installation functions (**SetupDi***Xxx* functions) that installers can call to handle the parameters:

DeviceInfoSet
Supplies a handle to the device information set.

The handle is opaque. Use the handle, for example, to identify the device information set in calls to **SetupDi***Xxx* functions.

The *DeviceInfoSet* might have an associated device setup class. If so, call **SetupDiGetDeviceInfoListClass** to get the class GUID.

DeviceInfoData
Optionally supplies a pointer to an SP_DEVINFO_DATA structure that identifies a device in the device

information set.

Device Installation Parameters

These indirect parameters supply information for the device installation in an SP_DEVINSTALL_PARAMS structure. If *DeviceInfoData* is not NULL, there are device installation parameters associated with the *DeviceInfoData*. If *DeviceInfoData* is NULL, the device installation parameters are associated with the *DeviceInfoSet*.

Call **SetupDiGetDeviceInstallParams** to get the device installation parameters.

Class Installation Parameters

The optional indirect parameters are specific to the particular DIF request. These are essentially "DIF request parameters." For example, the class installation parameters for a DIF_REMOVE installation request are contained in an SP_REMOVEDEVICE_PARAMS structure.

Each SP_*XXX*_PARAMS structure starts with a fixed-sized SP_CLASSINSTALL_HEADER.

Call **SetupDiGetClassInstallParams** to get the class installation parameters.

If a DIF request has class installation parameters, there is a set of parameters associated with the *DeviceInfoSet* and another set of parameters associated with the *DeviceInfoData* (if the DIF request specifies *DeviceInfoData*). **SetupDiGetClassInstallParams** returns the most specific parameters available.

Context

Co-installers have an optional context parameter.

Installer Output

Describes the output expected for this DIF code.

If an installer modifies the device installation parameters, the installer must call **SetupDiSetDeviceInstallParams** to apply the changes before returning. Similarly, if an installer modifies the class installation parameters for the DIF code, the installer must call **SetupDiSetClassInstallParams**.

Installer Return Value

Specifies the appropriate return value(s) for the DIF code. See the following figure for more information about return values.

Default DIF Code Handler

Specifies the **SetupDi** function that carries out system-defined operations for the DIF code. Not all DIF codes have a default handler. Unless a co-installer or class installer takes steps to prevent the default handler from being called, **SetupDiCallClassInstaller** calls the default handler for a DIF code after it calls the class installer (but before it calls any co-installers that registered for post-processing).

Co-installers should *not* call default DIF code handlers.

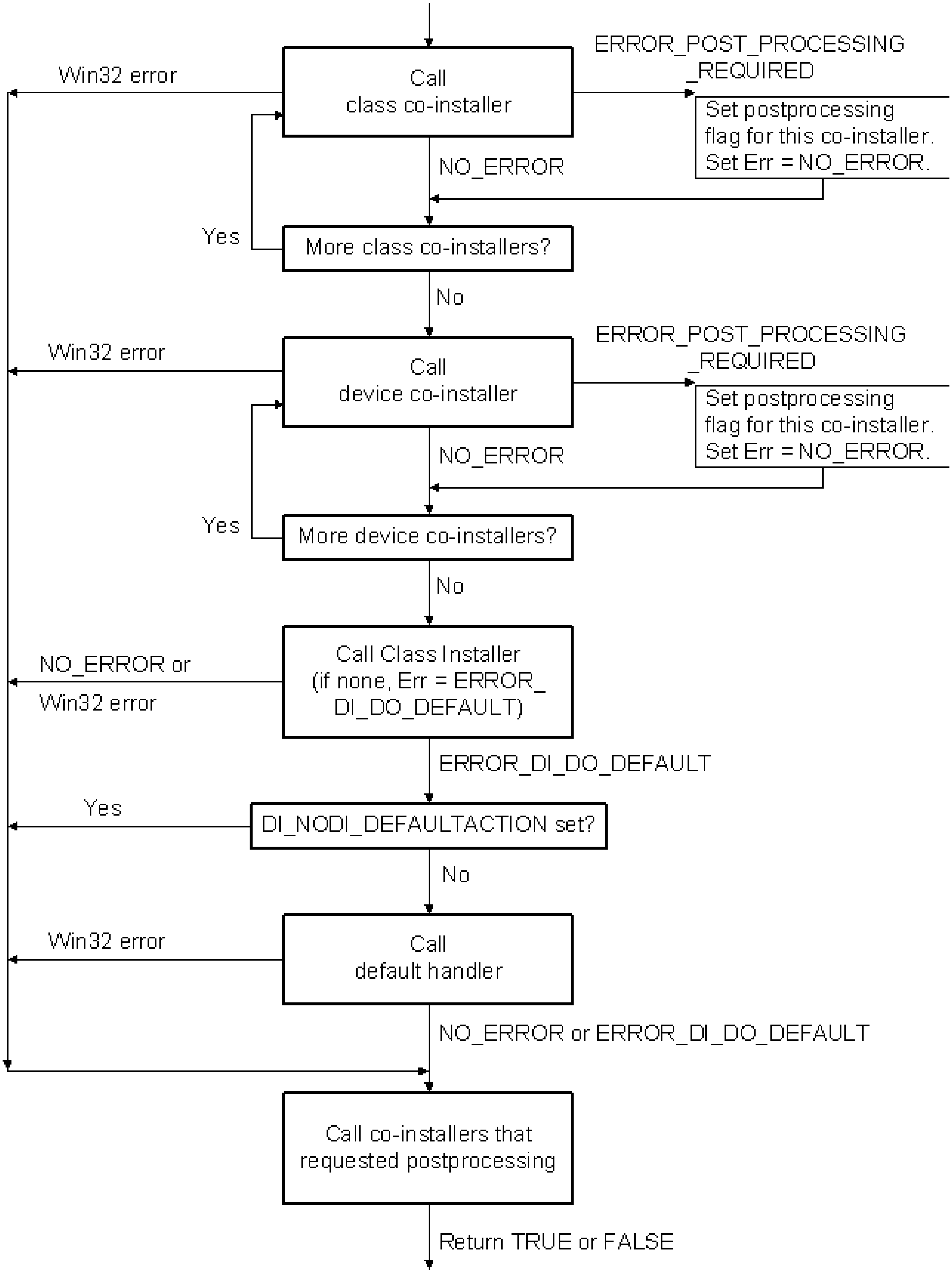
Installer Operation

Describes typical tasks that an installer might do to handle the DIF request.

See Also

Lists sources of related information.

The following figure shows the sequence of events in **SetupDiCallClassInstaller** for processing a DIF code.


Win32 error
Call
class co-installer
ERROR_POST_PROCESSING
_REQUIRED
Set postprocessing
flag for this co-installer.
Set Err = NO_ERROR.
NO_ERROR
Yes
More class co-installers?
No
Win32 error
Call
device co-installer
ERROR_POST_PROCESSING
_REQUIRED
Set postprocessing
flag for this co-installer.
Set Err = NO_ERROR.
NO_ERROR
Yes
More device co-installers?
No
NO_ERROR or
Win32 error
Call Class Installer
(if none, Err = ERROR_
DI_DO_DEFAULT)
ERROR_DI_DO_DEFAULT
Yes
DI_NODI_DEFAULTACTION set?
No
Win32 error
Call
default handler
NO_ERROR or ERROR_DI_DO_DEFAULT
Call co-installers that
requested postprocessing
Return TRUE or FALSE

dif-flow

Flow of DIF Code Processing in SetupDiCallClassInstaller

The OS performs some operations for each DIF code. Vendor-supplied co-installers and class installers can participate in the installation activities. Note that **SetupDiCallClassInstaller** calls co-installers that registered for postprocessing even if the DIF code fails.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Registering a Co-installer

A co-installer can be registered for a single device or for all the devices in a particular setup class. A device-specific co-installer is registered dynamically through the INF file when one of its devices is installed. Class co-installers are registered manually or by a Custom Device Installation Application and an INF.

For more information, see

Registering a Device-Specific Co-installer

Registering a Class Co-installer

To update a co-installer, each new version of the DLL should have a new filename because the DLL is typically in use when the user clicks the Update Driver button on the device property page.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Registering a Device-Specific Co-installer

To register a device-specific co-installer, add the following sections to the device's INF file:

```
;   :
;   :
[DestinationDirs]
XxxCopyFilesSection = 11                \\DIRID_SYSTEM
                                        \\ Xxx = driver or dev. prefix
;   :
;   :
[XxxInstall.OS-platform.CoInstallers]   \\ OS-platform is optional
CopyFiles = XxxCopyFilesSection
AddReg = Xxx.OS-platform.CoInstallers_AddReg

[XxxCopyFilesSection]
XxxCoInstall.dll

[Xxx.OS-platform.CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,"XxxCoInstall.dll, \
    XxxCoInstallEntryPoint"
```

The entry in the **DestinationDirs** section specifies that files listed in the *Xxx*CopyFilesSection will be copied to the system directory. The *Xxx* prefix identifies the driver, the device, or a group of devices (for example, cdrom_CopyFilesSection). The *Xxx* prefix should be unique.

The co-installer install section name can be decorated with an optional OS/architecture extension (for example, cdrom_install.NTx86.CoInstallers).

The entry in the *Xxx*_**AddReg** section creates a **CoInstallers32** value entry in the device's driver key. The entry contains the co-installer DLL and, optionally, a specific entry point. If you omit the entry point, the default is CoDeviceInstall. The hexadecimal flags parameter (0x00010000) specifies that this is a REG_MULTI_SZ value entry.

To register more than one device-specific co-installer for a device, copy the files for each co-installer and include more than one string in the registry entry. For example, to register two co-installers, create INF sections like the following:

```
;   :
;   :
[DestinationDirs]
XxxCopyFilesSection = 11                 \\DIRID_SYSTEM
                                         \\ Xxx = driver or dev. prefix
;   :
;   :
[XxxInstall.OS-platform.CoInstallers]    \\ OS-platform is optional
CopyFiles = XxxCopyFilesSection
AddReg = Xxx.OS-platform.CoInstallers_AddReg

[XxxCopyFilesSection]
XxxCoInstall.dll                          \\ copy 1st coinst. file
YyyCoInstall.dll                          \\ copy 2nd coinst. file

[Xxx.OS-platform.CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,                  \
    "XxxCoInstall.dll, XxxCoInstallEntryPoint", \
    "YyyCoInstall.dll, YyyCoInstallEntryPoint"
                                          \\ add both to registry
```

Device-specific co-installers are registered during the process of installing a device, when the Coinstallers INF section is processed. Setup API then calls the co-installers at each subsequent step of the installation process. If more than one co-installer is registered for a device, Setup API calls them in the order in which they are listed in the registry.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Registering a Class Co-installer

To register a co-installer for every device of a particular setup class, create a registry entry like the following under the **HKLM\System\CurrentControlSet\Control\CoDeviceInstallers** subkey:

```
{setup-class-GUID}: REG_MULTI_SZ : "XyzCoInstall.dll,XyzCoInstallEntryPoint\0\0"
```

The system creates the **CoDeviceInstallers** key. *Setup-class-GUID* specifies the GUID for the device setup class. If the co-installer applies to more than one class of devices, create a separate value entry for each setup class.

You must not overwrite other co-installers that have been previously written to the *setup-class-GUID* key. Read the key, append your co-installer string to the REG_MULTI_SZ list, and write the key back to the registry.

If you omit the *CoInstallEntryPoint*, the default is CoDeviceInstall.

The co-installer DLL must also be copied to the system directory.

The class co-installer is available to be called for relevant devices and services once the file has been copied and the registry entry is made.

Rather than manually creating the registry entry to register a class co-installer, you can register it using an INF file like the following:

```
[version]
signature = "$Windows NT$"

[DestinationDirs]
DefaultDestDir = 11    // DIRID_SYSTEM

[DefaultInstall]
CopyFiles = @classXcoinst.dll
AddReg = CoInstaller_AddReg

[CoInstaller_AddReg]
HKLM,System\CurrentControlSet\Control\CoDeviceInstallers, \
 {setup-class-GUID},0x00010008, "classXcoinst.dll,classXCoInstaller"
; above line uses the line continuation character (\)
```

This sample INF copies the file *classXcoinst.dll* to the system directory and makes an entry for the *setup-class-GUID* class under the **CoDeviceInstallers** key. The entry in the *Xxx*_AddReg section specifies two flags: the "00010000" flag specifies that the entry is a REG_MULTI_SZ, and the "00000008" flag specifies that the new value is to be appended to any existing value (if the new value is not already present in the string).

Such an INF that registers a class co-installer can be activated by a right-click install or through an application that calls **SetupInstallFromInfSection**.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Writing a Class Installer

A class installer DLL performs installation operations that apply to devices in a particular device setup class. For example, the ports class installer is responsible for assigning a COM port name to a device in the ports setup class. If the devices in a particular setup class do not require any special installation operations, a class installer is not required for that setup class.

Microsoft provides class installers for the system-defined device setup classes that require a class installer. An IHV or OEM can provide a co-installer to perform any necessary setup operations. (See Writing a Co-installer.)

If you define a new device setup class for your device, you might need to write a class installer. However, you rarely need to define a new device setup class because almost all devices can be associated with one of the system-supplied device setup classes. For example, the maker of a digital camera might think that it needs a new setup class. However, cameras fall under the image class.

This chapter contains the following information:

Class Installer Interface

Device Installation Functions Summary

Registering a Class Installer

Tightening File-Open Security in a Class Installer INF File

See also the chapter on Writing a Co-installer for general information on processing DIF codes.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Class Installer Interface

Every class installer exports a class installer function. The default name for the function is **ClassInstall**. Setup API calls this function whenever it needs the class installer to carry out a specific installation task for a device of its setup class.

The prototype for the installer function is as follows:

```
typedef DWORD  (CALLBACK* CLASS_INSTALL_PROC) (
    IN DI_FUNCTION       InstallFunction,
    IN HDEVINFO          DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData  /* optional */
    );
```

The class installer function receives two or three arguments:

- a device installation function code, which identifies the installation request to perform
- a handle to a device information set
- optionally, a pointer to an SP_DEVINFO_DATA structure that contains information about the device to install

A class installer returns one of the following values:

ERROR_DI_DO_DEFAULT

A class installer typically returns ERROR_DI_DO_DEFAULT, which instructs Setup to perform the default action for the DIF request.

This return status is not really an error, just a valid return value. ERROR_DI_DO_DEFAULT indicates that the class installer possibly performed some actions that filter or augment the installation action, but the class installer's actions do not replace the default action.

For example, a class installer might set some flags in response to a DIF_INSTALLDEVICE request and return ERROR_DI_DO_DEFAULT. In response to this return value, Setup performs the default action for DIF_INSTALLDEVICE (calls **SetupDiInstallDevice**) and then calls any co-installers that requested post processing.

If you know that a given DIF request has no default action, you should still return ERROR_DI_DO_DEFAULT. The DIF request might have a default action in a future release of the operating system.

A Win32 error

A class installer can return an appropriate Win32® error.

NO_ERROR

A class installer only returns NO_ERROR if it determines that it has fully completed the device installation task, including calling any default handler or superseding the default operation.

A class installer might call the default handler if, for example, the installer needs to perform certain actions after the default handler has run. If a class installer calls the default handler, the installer must not return ERROR_DI_DO_DEFAULT; it must return NO_ERROR or a Win32 error.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Device Installation Functions Summary

Installers can use the device installation functions provided by Setup API to carry out installation operations. The device installation functions allow installers to search INF files for compatible drivers, display driver choices to the user through selection dialog boxes, and perform the actual driver installation.

Most of the device installation functions rely on information in the SP_DEVINFO_DATA structure to carry out installation tasks. Each device has an SP_DEVINFO_DATA structure associated with it. You can retrieve a handle (HDEVINFO) to a device information set that contains all installed devices in a particular class by using the **SetupDiGetClassDevs** function. You can use the **SetupDiCreateDeviceInfo** function to add a new device to a device information set. You can free all SP_DEVINFO_DATA structures in a device information set by using the **SetupDiDestroyDeviceInfoList** function. This function also frees any compatible device and class device lists that might have been added to the structure.

By using the **SetupDiBuildDriverInfoList** function, you can generate a list from which the installer or the user can choose the driver or device to install. **SetupDiBuildDriverInfoList** creates a list of compatible drivers or a list of all devices of a particular class.

Once you have a list of compatible drivers, you can prompt the user to select from the list by using the **SetupDiSelectDevice** function. This function displays a dialog box that contains information about each device in the device information set. You can install a selected driver by using the **SetupDiInstallDevice** function. This function uses information in the driver's INF file to create to any new registry entries required, set the configuration of the device hardware, and copy the driver files to the appropriate directories.

An installer might need to examine and set values under the registry key for a device that is about to be installed. You can open the hardware or driver key for a device by using the **SetupDiCreateDevRegKey** or **SetupDiOpenDevRegKey** function.

You can install a new class of device by using the **SetupDiInstallClass** function. This function installs the new class of device from an INF file that contains a **ClassInstall32** section.

You can remove a device from the system by using the **SetupDiRemoveDevice** function. This function deletes the device's registry entries and stops the device if possible. If the device cannot be dynamically stopped, the function sets flags that eventually cause the user to be prompted to shut down the system.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Registering a Class Installer

To register a class installer, add **Installer32** and **Icon** entries to the registry for the class. Microsoft® Windows® 2000 and later maintains a **System\CurrentControlSet\Control\Class** branch under **HKEY_LOCAL_MACHINE** for information about each class. (Windows 9x/Me maintains a **System\CurrentControlSet\Services\Class** branch under **HKEY_LOCAL_MACHINE** for information about each class.) The value of the key contains the localized description of the class.

You can specify the **Installer32** and **Icon** registry entries in an INF file, by including an **AddReg** section referenced by an INF ClassInstall32 section.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Tightening File-Open Security in a Class Installer INF File

For Windows 2000 and later, you can tighten the file-open security for a class of devices, by assigning a value to the **DeviceCharacteristics** value name in the registry.

Do this using the INF file that installs the class installer, within an *add-registry-section*, which is specified using the INF **AddReg** directive. The *add-registry-section* is referenced in an INF **ClassInstall32** section and has the following syntax:

```
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64]
...
AddReg = Xxx_AddReg
...

[Xxx_AddReg]
...
HKR,,DeviceCharacteristics,0x10001,characteristics      ; new
```

The *characteristics* value is the numeric value of the characteristics to assign to the device. This value is the result of ORing one or more FILE_* file characteristics values as defined in *wdm.h* or *ntddk.h*. Setting the FILE_DEVICE_SECURE_OPEN characteristic, which has a numeric value of 0x100, directs the I/O Manager to perform security checks on all open requests for devices of the setup class. The FILE_DEVICE_SECURE_OPEN characteristic is supported on Windows 2000 and later platforms but not on Win9x/Me platforms.

An individual device in the class can override the characteristics set by the class installer, if appropriate, by specifying a **DeviceCharacteristics** entry in its device INF file. See Tightening File-Open Security in a Device INF File for more information.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Writing a Device Installation Application

If your installation package includes drivers and INF files that replace "in-box" drivers and INF files, or if your package includes device-*specific* applications, it should include a device *installation* application that installs those components. The device installation application should be invoked by Autorun (described in Platform SDK documentation), so that it starts automatically when a user inserts your distribution disk.

Your installation package must handle two situations:

1. The user plugs in your hardware before inserting your distribution medium.
2. The user inserts your distribution medium before plugging in your hardware.

User Plugs in Hardware Before Inserting Distribution Medium

If the user plugs in new hardware before inserting your distribution medium into a drive, there are two scenarios to deal with:

1. System-supplied in-box drivers support the device.

   In this case, Setup automatically installs the device based on INF files and drivers stored on the system disk. The system will not ask the user to insert vendor-supplied distribution media, so if you have provided a distribution medium containing device-specific applications, the applications are not installed at this time.

2. System-supplied in-box drivers do not support the device.

   In this case the Found New Hardware wizard starts executing and asks the user for a distribution medium. When the user inserts the medium, the Autorun-invoked device installation application checks for in-progress installations. Because the Found New Hardware wizard is active, the device installation application should stop itself. The wizard can then search the medium for an appropriate INF file. You can provide a co-installer that installs device-specific applications, as described in Installing Device-Specific Applications.

In either of these cases, the system will also attempt to access the Windows Update Web site to see if newer drivers and INF files are available. If they are, these drivers and INFs will be used instead of those already on the system or on the distribution medium. If the Web site is inaccessible or if no new drivers are available there, the system uses the drivers and INFs available in-box or on the distribution medium (whichever are newer).

Your distribution medium can contain an Autorun-invoked device installation application. When the user inserts your distribution medium, this application will start automatically. Assuming the user inserts the medium after the device has been plugged in and installed, the device installation application can do either or both of the following:

- If the system installed the device using in-box drivers, the device installation application can re-install the device using drivers and INF files supplied on the distribution medium, using **UpdateDriverForPlugAndPlayDevices**. (This function can be called only if the device has already been installed.)
- If the distribution medium contains device-specific applications, and a co-installer didn't install those applications in response to a DIF_NEWDEVICEWIZARD_FINISHINSTALL function code, the device installation application can use Microsoft® Installer to install them. (Microsoft Installer is described in Platform SDK documentation.)

User Inserts Distribution Medium Before Plugging in Hardware

If the user inserts your distribution medium before plugging in the hardware, an Autorun-invoked device installation application on the medium can:

- Check for in-progress installations, and stop executing if other installation activities are in progress.
- "Pre-install" driver files and INFs (see Pre-installing Driver Files).
- Use Microsoft Installer to install device-specific applications.
- If the device is "hot-pluggable," tell the user to plug it in. (If the bus doesn't provide hot-plug notification, initiate re-enumeration by calling **CM_Reenumerate_DevNode**.)
- If the device is not hot-pluggable, tell the user to turn the system off, plug in the device, and turn the system back on.

Determining if the Device Is Plugged In

Note that the behavior of an Autorun-invoked device installation application must depend on whether the user plugs in the hardware first or inserts the distribution medium first. Since vendors typically provide one distribution disk, and a disk can only have one Autorun-invoked application, your Autorun-invoked device installation application must determine if your device has been plugged in.

To determine if a device is plugged in, the application must call device installation functions to retrieve a list of devices and to examine each device, to determine if it is one the application can install.

Pre-installing Driver Files

To pre-install driver files, your device installation application should follow these steps:

1. On the target system, create a directory for the driver files. If your device installation application installs an application, the driver files should be stored in a subdirectory of the application directory.
2. Copy all files in the driver package from the distribution media to the directory created in Step 1. The driver package includes the driver or drivers, the INF file, the catalog file, and so forth.
3. Call **SetupCopyOEMInf** (described in Platform SDK documentation), specifying the INF file in the directory created in Step 1. Specify SPOST_PATH for the *OEMSourceMediaType* parameter and specify NULL for the *OEMSourceMediaLocation* parameter. **SetupCopyOEMInf** copies the INF file for the driver package into the *%windir%\Inf* directory on the target system and directs SetupAPI to store the source location of the INF file in its list of preprocessed INF files. **SetupCopyOEMInf** also processes the catalog file, so the PnP Manager will install the driver the next time it recognizes a device listed in the INF file.

When the user plugs in the device, the PnP Manager recognizes the device, finds the INF file copied by **SetupCopyOEMInf**, and installs the drivers copied in Step 2. (For more information about copying INF files, see Copying INFs.)

Installing Device-Specific Applications

If your distribution medium includes device-specific applications, the medium should include two methods for installing those applications:

1. A device-specific co-installer that installs the device-specific applications when it receives the DIF_NEWDEVICEWIZARD_FINISHINSTALL function code.

   If the user plugs in the device before inserting the distribution medium, and the device is not supported by in-box drivers, Setup calls the co-installer during the installation process. The co-installer should determine if the device-specific applications have already been installed and if they have not, it should install them. For more information, see Writing a Co-installer.

2. A device installation application that uses Windows Installer to install the device-specific applications.

   If the user inserts the distribution medium before plugging in the device, the medium's Autorun-invoked device installation application should determine if the device-specific applications have already been installed and if they have not, it should install them using Windows Installer. For more information, see the Platform SDK documentation.

Checking for In-Progress Installations

If your device installation application will be invoked by Autorun when the installation CD is inserted, there is another consideration. If your application is meant for use on Windows 2000 and later (but not Windows 95/98/Me), it should determine if other installation activities are in progress before performing its installations. To make this determination, the application should call **CMP_WaitNoPendingInstallEvents**, typically with a zero time-out value. If the return value from this function indicates other installation activities are pending (for instance, the Found New Hardware wizard might be active), the device installation application should exit.

To make your device installation application compatible with platforms that don't support **CMP_WaitNoPendingInstallEvents**, the application should include the following code:

```
BOOL
IsDeviceInstallInProgress (VOID)
```

```
{
    HMODULE hModule;
    CMP_WAITNOPENDINGINSTALLEVENTS_PROC pCMP_WaitNoPendingInstallEvents;

    hModule = GetModuleHandle(TEXT("setupapi.dll"));
    if(!hModule)
    {
        // Should never happen since we're linked to SetupAPI, but...
        return FALSE;
    }

    pCMP_WaitNoPendingInstallEvents =
        (CMP_WAITNOPENDINGINSTALLEVENTS_PROC)GetProcAddress(hModule,
                                             "CMP_WaitNoPendingInstallEvents");
    if(!pCMP_WaitNoPendingInstallEvents)
    {
        // We're running on a release of the operating system that doesn't supply this functi
        // Trust the operating system to suppress Autorun when appropriate.
        return FALSE;
    }
}

int
__cdecl
_tmain(IN int argc, IN PTCHAR argv[])
{
    if(IsDeviceInstallInProgress()) {
        //
        // We don't want to start right now.  Instead, our
        // device co-installer will invoke this application
        // (if necessary) during finish-install processing.
        //
        return -1;
    }
    .
    .
}
```

Use of this code is based on the premise that if a platform doesn't support **CMP_WaitNoPendingInstallEvents**, the platform does not invoke Autorun if installation activities are in progress.

Guidelines for Writing Device Installation Applications

Device installation applications *must* do the following:

- Support removal ("uninstall") of all device-specific applications that it installs. As part of that uninstall process, the device installation application should check to see if any associated devices are still present on the system and, if so, warn the user.
- Follow the guidelines for installing devices on 64-bit systems.
- List all installed applications in **Add/Remove Programs**, using Microsoft® Windows® Installer (MSI), so the user can uninstall them if desired.
- Follow the guidelines for Microsoft® Windows® applications. See the MSDN® web site for more information (http://msdn.microsoft.com).

(Submit device-specific applications to the appropriate Windows Logo program for software. See the MSDN web site for more information.)

Device installation applications *must not* do the following:

- Do not instruct the user to copy or overwrite any files, especially *.inf* and *.sys* files.
- Do not delete the installed driver files from the system during the uninstall operation, even if the hardware has been removed.

- Do not force any unnecessary system reboots. Reboots are generally not required for installing PnP devices or software applications. The **UpdateDriverForPlugAndPlayDevices** function's *bRebootRequired* parameter indicates the need for a reboot.
- Do not use "RunOnce" registry entries to spawn device installation applications, because this requires a system reboot.
- Do not use a device or class co-installer, or a class installer, to spawn a device installation application, because the state of the system during device installation cannot be guaranteed to be safe for installing software applications. Specifically, if the device installation application runs on the server side, the system will hang. (However, device installation applications can be spawned from within finish-install wizard pages supplied by class installers or co-installers.)
- Do not use the Startup Group to spawn device installation applications.
- Do not use *win.ini* entries to spawn device installation applications.
- Do not force the user to install any device-specific applications, unless the device will not operate without the application. Examples might include utilities for setting configurable keyboard keys or for setting a modem's country/region code, if an in-box application does not support such a capability.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Providing Device Property Pages

A co-installer or class installer should supply a custom device property page if its device or class has any individual properties that a user can set. Such properties might include default playback volume for a CD-ROM drive or speaker volume for a modem. The Device Manager displays the property page any time a user clicks on Properties for the device. For versions of Microsoft® Windows NT® prior to Windows 2000, users set such information through Control Panel applets. Driver software written for Windows® 2000 and later should provide property pages instead.

Although you can write a class installer that provides custom property pages, it's generally preferable to provide this functionality in a co-installer, along with other device- or device-class-specific features.

The Platform SDK documentation provides comprehensive documentation of property pages and the Win32® functions that manipulate them. This chapter supplements the Platform SDK documentation with the following information specific to device installation:

Required Support for Device Property Pages

Handling DIF_ADDPROPERTYPAGE_ADVANCED Requests

Property Page Callback Function

Handling Windows Messages for Property Pages

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

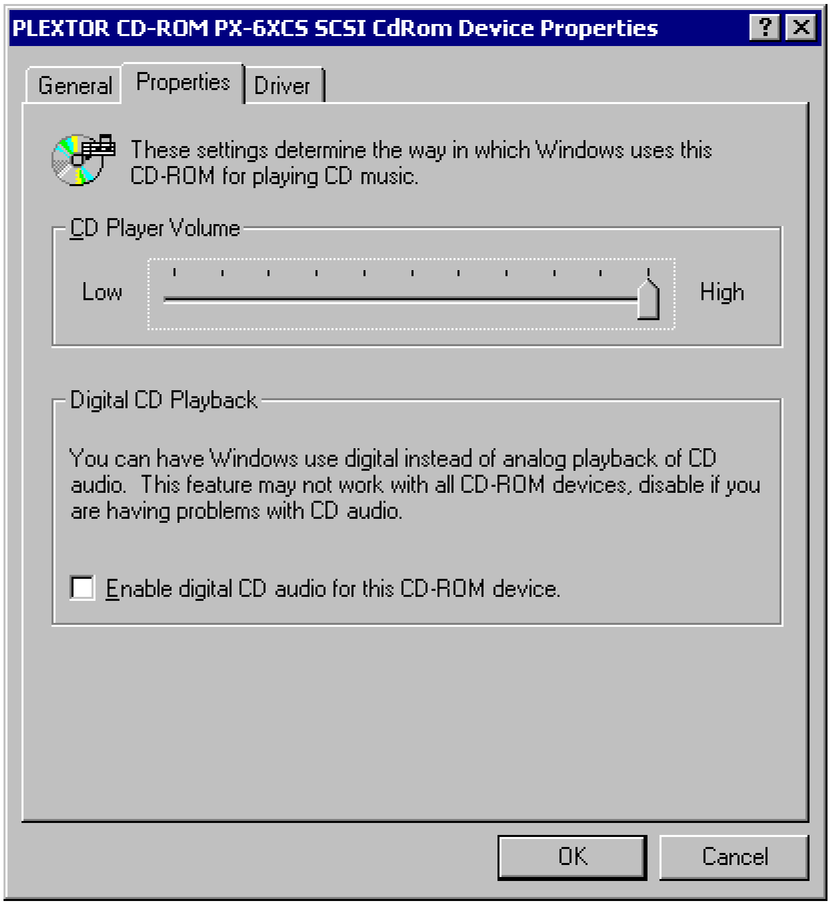
Device Installation: Windows DDK

Required Support for Device Property Pages

A provider of custom property pages must do the following:

- Handle DIF_ADDPROPERTYPAGE_ADVANCED requests.
- Supply a callback that handles PSPCB_CREATE and PSPCB_RELEASE messages if additional storage must be allocated and released for a property page.
- Supply a dialog box procedure that handles Windows messages for each custom property page.

The following figure shows a sample custom properties page.



These settings determine the way in which Windows uses this CD-ROM for playing CD music.

You can have Windows use digital instead of analog playback of CD audio. This feature may not work with all CD-ROM devices, disable if you are having problems with CD audio.

☐ Enable digital CD audio for this CD-ROM device.

A Sample Custom Properties Page

In the previous figure, an installer supplies a property page named Properties. By default, the system supplies the General and Driver tabs shown in the figure. When appropriate, the system also provides Resources and Power tabs.

A setup component can define more than one property page for its device or class. Each tab should have a name that succinctly describes the purpose of the page. The system does not check to ensure that the names are unique.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Handling DIF_ADDPROPERTYPAGE_ADVANCED

Requests

An installer that supplies one or more custom property pages must handle the DIF_ADDPROPERTYPAGE_ADVANCED device installation function code. The Device Manager, through Setupapi.dll, sends this request when a user clicks on Properties for a device in the Device Manager or in a Control Panel applet.

In response to this request, the installer provides information about each of its custom property pages, creates the pages, and adds the created pages to the list of dynamic property pages for the device.

The installer should take the following steps:

1. Call **SetupDiGetClassInstallParams** to get the current class install parameters for the device. For example:

   ```
   SP_ADDPROPERTYPAGE_DATA AddPropertyPageData;
   :
   ZeroMemory(&AddPropertyPageData, sizeof(SP_ADDPROPERTYPAGE_DATA));
   AddPropertyPageData.ClassInstallHeader.cbSize =
        sizeof(SP_CLASSINSTALL_HEADER);
   if (SetupDiGetClassInstallParams(DeviceInfoSet, DeviceInfoData,
        (PSP_CLASSINSTALL_HEADER)&AddPropertyPageData,
        sizeof(SP_ADDPROPERTYPAGE_DATA), NULL )) {
   …
   ```

   The example zero-initializes the structure in which the class install parameters will be returned and sets the size of the class install header in the **cbSize** field as required by **SetupDiGetClassInstallParams**. The class install header is the first member of each class install parameters structure.

   When a DIF_ADDPROPERTYPAGE_ADVANCED request is active, this function returns an SP_ADDPROPERTYPAGE_DATA structure at *ClassInstallParams*. Therefore, the call to the **SetupDi** function can cast the returned *ClassInstallParams* to a structure of this type.

2. Ensure that the maximum number of dynamic pages for the device has not yet been met, with a statement like the following:

   ```
   if (AddPropertyPageData.NumDynamicPages <
       MAX_INSTALLWIZARD_DYNAPAGES)
    …
   ```

   If the test fails, do not initialize or create the page. Simply return NO_ERROR.

3. Allocate memory in which to save any device-specific data that will be needed later in the dialog box procedure and initialize this area with the data. This data must include the *DeviceInfoSet* and *DeviceInfoData* passed with the request.

   For example, a property page provider can define and use a structure as follows:

   ```
   typedef struct _TEST_PROP_PAGE_DATA {
       HDEVINFO DeviceInfoSet;
       PSP_DEVINFO_DATA DeviceInfoData;
   } TEST_PROP_PAGE_DATA, *PTEST_PROP_PAGE_DATA;
   …
   PTEST_PROP_PAGE_DATA     pMyPropPageData;
   …
   pMyPropPageData = HeapAlloc(GetProcessHeap(), 0,
        sizeof(TESTPROP_PAGE_DATA));
   ```

4. Initialize a PROPSHEETPAGE structure with information about the custom property page:
   - In **dwFlags**, set the PSP_USECALLBACK flag and any other flags required for the custom property page.

  The PSP_USECALLBACK flag indicates that a callback function has been supplied.
  - In **pfnCallback**, set a pointer to the callback function for the property page.
  - In **pfnDlgProc**, set a pointer to the dialog box procedure for the property page.
  - In **lParam**, pass a pointer to the memory area allocated and initialized in step 3.
  - Set other members as appropriate for the custom property page. See the *Platform SDK* documentation for more information on the PROPSHEETPAGE structure.
5. Call **CreatePropertySheetPage** to create the new page.
6. Add the new page to the list of dynamic property pages in the **DynamicPages** member of the class install parameters and increment the **NumDynamicPages** member.
7. Repeat steps 2 through 6 for each additional custom property page.
8. Call **SetupDiSetClassInstallParams** to set the new class install parameters, which include the updated property page structure.
9. Return NO_ERROR.

Setup adds the newly created property pages to the property sheet for the device, and the Device Manager makes Win32 API calls to create the sheet. When the property page is displayed, the system calls the dialog box procedure specified in the PROPSHEETPAGE structure.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Property Page Callback Function

When an installer sets up a property page for its device or device class, it supplies a pointer to a callback function. The callback function is called once when the property page is created and again when it is about to be destroyed.

The callback is a **PropSheetPageProc** function, which is described in the Platform SDK documentation. It must be able to handle the PSPCB_CREATE and PSPCB_RELEASE actions.

The callback is called with a PSPCB_CREATE message when a property page is being created. In response to this message, the callback can allocate memory for data associated with the page. The function should return TRUE to continue creating the page or FALSE if for some reason the page should not be created.

Property pages for a device are destroyed when the user clicks OK or Cancel on the page's dialog box or clicks Uninstall on the Drivers tab.

When a property page is being destroyed, the callback is called with a PSPCB_RELEASE message. The function should free any data allocated when the property page was created. Typically, this involves freeing the data at **lParam** in the PROPSHEETPAGE structure. The return value is ignored when the page is being destroyed.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Handling Windows Messages for Property Pages

When an installer that supplies property pages handles a DIF_ADDPROPERTYPAGE_ADVANCED request, it sets the address of a dialog box procedure for the property page. The dialog box procedure must initialize the property page when it gets a WM_INITDIALOG message, and it must be prepared to handle changes to device properties when it gets a WM_NOTIFY message. The procedure can also handle any other such messages it may require, as described in the Platform SDK documentation.

In response to a WM_INITDIALOG message, the dialog box procedure initializes information in the property page. Such information might include an icon that represents the device, the friendly name of the device, and its PnP device description.

SetupDiLoadClassIcon loads the icons for a specified device class and returns a handle to the loaded large icon that can be used in a subsequent call to **SendDlgItemMessage**. For example:

```
if (SetupDiLoadClassIcon(
        &pTestPropPageData->DeviceInfoData->ClassGuid, &ClassIcon,
        NULL)) {
    OldIcon = (HICON)SendDlgItemMessage(hDlg, IDC_TEST_ICON,
        STM_SETICON, (WPARAM)ClassIcon, 0);
    if (OldIcon) {
                DestroyIcon(OldIcon);
    }
}
```

The handle returned in ClassIcon can be cast to the WPARAM required by the **SendDlgItemMessage** function. In the example, IDC_TEST_ICON identifies the control in the dialog box that receives the STM_SETICON message. The value of IDC_TEST_ICON must be defined in the installer. See Device Installation Functions for additional **SetupDi** functions that manipulate icons and bitmaps, and see the Platform SDK documentation for more information on **SendDlgItemMessage**, **DestroyIcon**, and using icons in dialog boxes.

In addition to an icon representing the device, a typical device property page includes a description or "friendly name" of the device and shows the current settings of device properties. The PnP Manager stores the PnP properties of each device in the registry. A property page provider can call **SetupDiGetDeviceRegistryProperty** to get the value of any such property. If device- or class-specific configuration information has also been stored in the registry as part of the installation process, a property page provider can use other **SetupDi** functions to extract that information for display. See Device Installation Functions for details.

When certain types of changes occur on the page, the property sheet sends a WM_NOTIFY message to the dialog box procedure. The dialog box procedure should be prepared to extract the notification code from the message parameters and respond appropriately.

The PSN_APPLY notification is of particular interest to an installer. The property sheet sends this message when the user clicks OK or Apply. In response, the installer should validate the changes and set the DI_FLAGSEX_PROPCHANGE_PENDING flag in the device install parameters, as follows:

1. If it has not already done so, get a pointer to the device install parameters (SP_DEVINSTALL_PARAMS structure) for the device. This structure is available by calling **SetupDiGetDeviceInstallParams**, passing the saved *DeviceInfoSet* and *DeviceInfoData* that were passed in the area at *lParam*.
2. Ensure that the user's changes are valid.
3. Set the DI_FLAGSEX_PROPCHANGE_PENDING flag in the **FlagsEx** field of the returned SP_DEVINSTALL_PARAMS structure. However, if the installer can be absolutely certain that the changes do not require the device's drivers to be stopped and then restarted, it need not set this flag.
4. Call **SetupDiSetDeviceInstallParams** with the changed SP_DEVINSTALL_PARAMS structure to set the new parameters.

After the installer sets the new parameters, Setup or the Device Manager sends a DIF_PROPERTYCHANGE request.

The property sheet sends a PSN_RESET notification message when the user clicks Cancel. In response to this message, the dialog box procedure should discard any changes made by the user.

See the Platform SDK documentation for more information on the other notifications that such a procedure might encounter and how it should handle them.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Troubleshooting Device Installation

To verify that your device installed properly, you should do the following:

- Use the Device Manager to view system information about the device.
- Use SetupAPI Logging to identify installation errors.
- Run test programs to exercise the device, including the testing and debugging tools supplied with this DDK.

Additionally, a co-installer or class installer can provide a troubleshooter that helps users troubleshoot problems with your device. See DIF_TROUBLESHOOTER for more information.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Using the Device Manager

To start the Device Manager, right-click on the My Computer icon, select Manage from the menu, and select Device Manager from the System Tools listed in the resulting display. The Device Manager displays information about each device, including the device type, device status, manufacturer, device-specific properties, and information about the driver for the device. See the online help in the Device Manager for more information.

If your device is a boot device, a problem with your device installation can prevent the machine from booting. In such cases, you will need to use the kernel debugger to troubleshoot your device installation. For more information, see the online documentation supplied with the Microsoft debuggers.

If your device is not required to boot the machine, you will typically notice a problem with your device installation because your device is not working properly and the Device Manager marks your device with a yellow exclamation point. The Device Manager also provides an error message. See Device Manager Error Messages.

When you are testing the installation of a new PnP device, it can be useful to have Device Manager list hidden devices. For more information, see Viewing Hidden Devices.

For Windows XP and later, the Device Manager provides a **Details** tab containing information that can be useful when debugging drivers. For more information, see Device Manager **Details** Tab.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Device Manager Error Messages

When the Device Manager marks a device with a yellow exclamation point, it also provides an error message. The following table lists the errors reported by the Device Manager. (Error codes are defined in *cfg.h*.)

Error Code	**Problem Name**
Code 1	CM_PROB_NOT_CONFIGURED
Code 2	CM_PROB_DEVLOADER_FAILED
Code 3	CM_PROB_OUT_OF_MEMORY
Code 4	CM_PROB_ENTRY_IS_WRONG_TYPE
Code 5	CM_PROB_LACKED_ARBITRATOR
Code 6	CM_PROB_BOOT_CONFIG_CONFLICT
Code 7	CM_PROB_FAILED_FILTER
Code 8	CM_PROB_DEVLOADER_NOT_FOUND
Code 9	CM_PROB_INVALID_DATA
Code 10	CM_PROB_FAILED_START
Code 11	CM_PROB_LIAR
Code 12	CM_PROB_NORMAL_CONFLICT
Code 13	CM_PROB_NOT_VERIFIED
Code 14	CM_PROB_NEED_RESTART
Code 15	CM_PROB_REENUMERATION
Code 16	CM_PROB_PARTIAL_LOG_CONF
Code 17	CM_PROB_UNKNOWN_RESOURCE
Code 18	CM_PROB_REINSTALL
Code 19	CM_PROB_REGISTRY
Code 20	CM_PROB_VXDLDR
Code 21	CM_PROB_WILL_BE_REMOVED
Code 22	CM_PROB_DISABLED
Code 23	CM_PROB_DEVLOADER_NOT_READY
Code 24	CM_PROB_DEVICE_NOT_THERE
Code 25	CM_PROB_MOVED
Code 26	CM_PROB_TOO_EARLY
Code 27	CM_PROB_NO_VALID_LOG_CONF
Code 28	CM_PROB_FAILED_INSTALL
Code 29	CM_PROB_HARDWARE_DISABLED
Code 30	CM_PROB_CANT_SHARE_IRQ
Code 31	CM_PROB_FAILED_ADD
Code 32	CM_PROB_DISABLED_SERVICE
Code 33	CM_PROB_TRANSLATION_FAILED
Code 34	CM_PROB_NO_SOFTCONFIG
Code 35	CM_PROB_BIOS_TABLE
Code 36	CM_PROB_IRQ_TRANSLATION_FAILED
Code 37	CM_PROB_FAILED_DRIVER_ENTRY
Code 38	CM_PROB_DRIVER_FAILED_PRIOR_UNLOAD
Code 39	CM_PROB_DRIVER_FAILED_LOAD
Code 40	CM_PROB_DRIVER_SERVICE_KEY_INVALID
Code 41	CM_PROB_LEGACY_SERVICE_NO_DEVICES
Code 42	CM_PROB_DUPLICATE_DEVICE

Code 43	CM_PROB_FAILED_POST_START
Code 44	CM_PROB_HALTED
Code 45	CM_PROB_PHANTOM
Code 46	CM_PROB_SYSTEM_SHUTDOWN
Code 47	CM_PROB_HELD_FOR_EJECT
Code 48	CM_PROB_DRIVER_BLOCKED
Code 49	IDS_PROB_REGISTRY_TOO_LARGE

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_NOT_CONFIGURED

There is a device on the system for which there is no **ConfigFlags** registry entry, meaning no driver is installed. Typically this means an INF file could not be found.

Error Code

1

Display Message – NT-based Systems

"This device is not configured correctly. (Code 1)"

"To Update the drivers for this device, click Update Driver. If that doesn't work, see your hardware documentation for more information."

Recommended Resolution – NT-based Systems

Select **Update Driver**, which starts the Update Driver wizard.

Display Message – Windows 98/Me

"This device is not configured correctly. (Code 1)"

"To Update the drivers for this device, click Update Driver. If that doesn't work, see your hardware documentation for more information."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DEVLOADER_FAILED

Device loader failed to load drivers.

Error Code

2

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

1. If this is a Root bus device loader:

   "Windows could not load the driver for this device because the computer is reporting two <*bustype*> bus types. (Code 2)"

   "Contact your computer manufacturer to get an updated BIOS for your computer."

2. If this is not a Root bus device loader:

   "The <*device-loader-names*> device loader(s) for this device could not load the device driver. (Code 2)"

   "To fix this, click Update Driver to update the device driver."

Recommended Resolution – Windows 98/Me

Solution for (1): Update BIOS.

Solution for (2): Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_OUT_OF_MEMORY

Running out of memory – the system is probably running low on system memory.

Error Code

3

Display Message – NT-based Systems

"The driver for this device might be corrupted, or your system may be running low on memory or other resources. (Code 3)"

Recommended Resolution – NT-based Systems

Select **Update Driver**, which starts the Update Driver wizard.

Display Message – Windows 98/Me

"The driver for this device may be bad, or your system may be running low on memory or other resources. (Code 3)"

"To Update the drivers for this device, click Update Driver. To check your computer's memory and system resources, right-click My Computer on your desktop, click Properties, and then click the Performance tab."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_ENTRY_IS_WRONG_TYPE

Registry entries are incorrect.

Error Code

4

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"This device is not working properly because one of its drivers may be bad, or your registry may be bad. (Code 4)"

To Update the drivers for this device, click Update Driver. If that doesn't work, run Scanregw.exe (click Start, click Run, type Scanregw.exe, and then click OK) to check your registry."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard, or run *scanregw.exe*.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_LACKED_ARBITRATOR

The system lacks a resource arbitrator to resolve resource conflicts.

Error Code

5

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"The driver for this device requested a resource that Windows does not know how to handle. (Code 5)"

"To fix this, click Update Driver to update the driver for this device."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_BOOT_CONFIG_CONFLICT

The boot configuration contains resource conflicts.

Error Code

6

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Another device is using the resources this device needs. (Code 6)"

"To fix this, shut down your computer, turn it off, and then change the resources for this device. When you have finished, start Device Manager and change the resource settings for this device."

Recommended Resolution – Windows 98/Me

Select **Hardware Troubleshooter**, which will launch the hardware troubleshooter wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_FAILED_FILTER

Filtering failed.

Error Code

7

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

If the device's hardware key contains a "FailReasonString" value, the value string is displayed as the error message. (A driver or enumerator supplies this registry string value.) If the hardware key does not contain a "FailReasonString" value, the following generic error message is displayed:

"The drivers for this device need to be reinstalled. (Code 7)"

"To reinstall the drivers for this device, click Reinstall Driver."

Recommended Resolution – Windows 98/Me

Select **Reinstall Driver**, which starts the Update Driver wizard and reinstalls the current driver.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DEVLOADER_NOT_FOUND

A device loader was not found.

Error Code

8

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

1. If the device loader is a system device loader (that is, it begins with a **"*"**, meaning it is part of VMM32) then the following text is displayed:

   "This device is not working properly because Windows cannot load the file (*device-loader-filename*) that loads the drivers for the device. (Code 8)"

   "To fix this problem, run Windows Setup again using your Windows CD-ROM."

2. If this is not a system devloader and the devloader cannot be found (the file is missing), the following text is displayed:

   "This device is not working properly because Windows cannot load the file (*device-loader-filename*) that loads the drivers for the device. (Code 8)"

   "To fix this problem, click Reinstall Device to reinstall this device."

3. If this is not a system devloader and it can be found on the disk (the file does exist) the following text is displayed:

   "This device is not working properly because the file (*device-loader-filename*) that loads the drivers for this device is bad. (Code 8)

   "To fix this problem, click Update Driver to update the drivers for this device."

4. If the device's software key does not contain a "devloader" entry, or if the entry does not have a value assigned to it, the following text is displayed:

   "Device failure: Try changing the driver for this device. If that doesn't work, see your hardware documentation. (Code 8)"

Recommended Resolution – Windows 98/Me

Solution for (1) : Run Windows Setup again using your Windows CD-ROM.

Solution for (2): Select **Reinstall Driver**, which starts the Update Driver wizard and reinstalls the current driver.

Solution for (3): Select **Update Driver**, which starts the Update Driver wizard.

Solution for (4): Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_INVALID_DATA

Invalid data has been detected in the registry.

Error Code

9

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

1. If the device is enumerated by the BIOS or ACPI, the following text is displayed:

   "This device is not working properly because the BIOS in your computer is reporting the resources for the device incorrectly. (Code 9)"

   "Contact your computer manufacturer to get an updated BIOS for your computer."

2. If the device is not enumerated by the BIOS or ACPI (that is, it is an add-in card or a device that was plugged into the computer), the following text is displayed:

   "This device is not working properly because the BIOS in the device is reporting the resources for the device incorrectly. (Code 9)"

   "Contact the device manufacturer to get an updated BIOS for your "device.

Recommended Resolution – Windows 98/Me

Solution for (1): Update the BIOS.

Solution for (2): Update the BIOS.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_FAILED_START

The device failed to start.

Error Code

10

Display Message – NT-based Systems

If the device's hardware key contains a "FailReasonString" value, the value string is displayed as the error message. (A driver or enumerator supplies this registry string value.) If the hardware key does not contain a "FailReasonString" value, the following generic error message is displayed:

"This device cannot start. (Code 10)"

"Try upgrading the device drivers for this device."

Recommended Resolution – NT-based Systems

Select **Update Driver**, which starts the Update Driver wizard.

This error code is set when one of the drivers in the device's driver stack fails IRP_MN_START_DEVICE. If there are many drivers in the stack, it can be difficult to determine the one that failed.

Display Message – Windows 98/Me

If the device's hardware key contains a "FailReasonString" value, the value string is displayed as the error message. (A driver or enumerator supplies this registry string value.) If the hardware key does not contain a "FailReasonString" value, the following generic error message is displayed:

"This device is either not present, not working properly, or does not have all the drivers installed. (Code 10)"

"Try upgrading the device drivers for this device."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_LIAR

Automatic Skip Driver Agent (ASD) error.

Error Code

11

Display Message– NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Windows stopped responding while attempting to start this device, and therefore will never attempt to start this device again. (Code 11)"

"For more information, look up ASD in Windows Help."

"Try upgrading the device drivers for this device."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_NORMAL_CONFLICT

Two devices have been assigned the same I/O ports, the same interrupt, or the same DMA channel (either by the BIOS, the operating system, or a combination of the two).

Error Code

12

Display Message – NT-based Systems

"This device cannot find enough free resources that it can use. (Code 12)

"If you want to use this device, you will need to disable one of the other devices on this system."

Recommended Resolution – NT-based Systems

Select **Hardware Troubleshooter**, which starts the hardware troubleshooter.

This error message can also appear if the BIOS did not give a device enough resources (for example, if a USB controller doesn't get an interrupt from the BIOS because of a bad MPS table).

Display Message – Windows 98/Me

"This device cannot find any free <*resource-type*> resources to use. (Code 12)"

"If you want to use this device, you must disable another device that is using the resources this device needs. To do this, click Hardware Troubleshooter and follow the instructions in the wizard."

Recommended Resolution – Windows 98/Me

Select **Hardware Troubleshooter**, which starts the hardware troubleshooter.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_NOT_VERIFIED

The device could not be verified.

Error Code

13

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"This device is either not present, not working properly, or does not have all the drivers installed. (Code 13)"

"To have Windows detect whether this device is present or not, click Detect Hardware."

Recommended Resolution – Windows 98/Me

Select **Detect Hardware**, which starts the legacy detection wizard.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_NEED_RESTART

The system must be restarted.

Error Code

14

Display Message – NT-based Systems

"This device cannot work properly until you restart your computer. (Code 14)

"To restart your computer now, click Restart Computer."

Recommended Resolution – NT-based Systems

Select **Restart Computer**, which restarts your computer.

Display Message – Windows 98/Me

"This device cannot work properly until you restart your computer. (Code 14)"

"To restart your computer now, click Restart Computer."

Recommended Resolution – Windows 98/Me

Select **Restart Computer**, which restarts your computer.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_REENUMERATION

The device is causing a resource conflict.

Error Code

15

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"This device is causing a resource conflict. (Code 15)"

"To resolve the conflict, click Hardware Troubleshooter and follow the instructions in the wizard."

Recommended Resolution – Windows 98/Me

Select **Hardware Troubleshooter**, which starts the hardware troubleshooter.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_PARTIAL_LOG_CONF

The device is only partially configured.

Error Code

16

Display Message – NT-based Systems

"Windows cannot identify all the resources this device uses. (Code 16)

"To specify additional resources for this device, click the Resources tab and fill in the missing settings. Check your hardware documentation to find out what settings to use."

Recommended Resolution – NT-based Systems

Use the Resources tab to supply additional resources.

Display Message – Windows 98/Me

"Windows could not identify all the resources this device uses. (Code 16)"

"To specify additional resources for this device, click the Resources tab and fill in the missing settings. Check your hardware documentation to find out what settings to use."

Recommended Resolution – Windows 98/Me

Use the Resources tab to supply additional resources.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_UNKNOWN_RESOURCE

An unknown resource was specified in an INF file.

Error Code

17

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"The driver information file (*INF-file-name*) is telling this child device to use a resource that the parent device does not have or recognize. (Code 17)"

"To fix this, click Update Driver to update the drivers for this device."

Recommended Resolution – Windows 98/Me

Select **Update Driver** to update the device's drivers.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_REINSTALL

Drivers must be reinstalled.

Error Code

18

Display Message – NT-based Systems

"Reinstall the drivers for this device. (Code 18)"

Recommended Resolution – NT-based Systems

Windows XP starts the Hardware Update wizard.

Display Message – Windows 98/Me

"The drivers for this device need to be reinstalled. (Code 18)"

"To reinstall the drivers for this device, click Reinstall Driver."

Recommended Resolution – Windows 98/Me

Select **Reinstall Driver**, which will reinstall the drivers.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_REGISTRY

A registry problem was detected.

Error Code

19

Display Message – NT-based Systems

"Windows cannot start this hardware device because its configuration information (in the registry) is incomplete or damaged. (Code 19)"

Recommended Resolution – NT-based Systems

This can be set if more than one service is defined for a device, there is a failure opening the service key, or the driver name can't be obtained from the service key.

Display Message – Windows 98/Me

"Your registry may be bad. (Code 19)"

"To check your registry, click Check Registry. If the registry is bad, Windows will restart your system and go back to a previous registry that is good."

Recommended Resolution – Windows 98/Me

Select **Check Registry**, which will start the registry checker utility.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_VXDLDR

VxDLdr returned an unknown result.

Error Code

20

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Windows could not load one of the drivers for this device. (Code 20)"

"To fix this, click Update Driver to update the drivers for this device."

Recommended Resolution – Windows 98/Me

Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_WILL_BE_REMOVED

The system will remove the device.

Error Code

21

Display Message – NT-based Systems

"Windows is removing this device. (Code 21)"

"Close this dialog box, and then wait a few seconds. If this problem continues, restart your computer."

Recommended Resolution – NT-based Systems

Select **Restart Computer**, which will restart the computer.

Display Message – Windows 98/Me

"Windows is removing this device. (Code 21)"

"Close this dialog box, and then wait a few seconds. If this problem continues, restart your computer."

Recommended Resolution – Windows 98/Me

Select **Restart Computer**, which will restart the computer.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DISABLED

The device is disabled.

Error Code

22

Display Message – NT-based Systems

"This device is disabled. (Code 22)"

Recommended Resolution – NT-based Systems

The device is disabled because the user disabled it using Device Manager. Select **Enable Device**, which will enable the device.

Display Message – Windows 98/Me

1. If this device is disabled because the user disabled it using Device Manager, the following text is displayed:

   "This device is disabled. (Code 22)"

   "Click Enable Device to enable this device."

2. If the device is not started, the following text is displayed:

   "This device is not started. (Code 22)"

   "Click Start Device to start this device."

3. If the device is disabled by a driver or software program, the following text is displayed:

   "This device is disabled. (Code 22)"

   "You can't enable this device because it has been disabled by a Windows driver."

Recommended Resolution – Windows 98/Me

Solution for (1): Select **Enable Device**, which will enable the device.

Solution for (2): Select **Start Device**, which will start the device.

Solution for (3): None.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DEVLOADER_NOT_READY

The device loader was not ready.

Error Code

23

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

1. If this device is a secondary display adapter and the primary display adapter was located, the following text is displayed:

   "This display adapter is functioning correctly. (Code 23)"

   "The problem is with the main display adapter. To view the properties for the main display adapter, click Properties."

2. If this device is a secondary display adapter and the primary display adapter cannot be located, the following text is displayed:

   "This display adapter is functioning correctly. (Code 23)"

   "The problem is with the main display adapter. Fix the main display adapter, and then this display adapter will work."

3. If this device is not a display adapter, the following text is displayed:

   "The loaders for this device cannot load the required drivers. (Code 23)"

   "To update the device drivers, click Update Driver."

Recommended Resolution – Windows 98/Me

Solution for (1): Select **Properties**, which will display the property page for the primary display adapter.

Solution for (2): Fix the main display adapter.

Solution for (3): Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_DEVICE_NOT_THERE

The device doesn't appear to be present.

Error Code

24

Display Message – NT-based Systems

"This device is not present, is not working properly, or does not have all its drivers installed. (Code 24)"

Recommended Resolution – NT-based Systems

The problem could be bad hardware, or a new driver might be needed. Devices stay in this state if they have been prepared for removal. This error code can be set if a driver's **DriverEntry** routine detects a device but the **DriverEntry** routine later fails. (For Windows XP and later, **DriverEntry** problems have separate error codes.)

Display Message – Windows 98/Me

1. If this device is a legacy (root detected) device then the following text will be displayed:

   "This device is either not present, not working properly, or does not have all the drivers installed. (Code 24)"

   "To have Windows detect whether this device is present or not, click Detect Hardware."

2. If this is a Plug and Play device then the following text is displayed:

   "This device is either not present, not working properly, or does not have all the drivers installed. (Code 24)"

   "Try upgrading the device drivers for this device."

Recommended Resolution – Windows 98/Me

Solution for (1): Select **Detect Hardware**, which starts the legacy detection wizard.

Solution for (2): Select **Update Driver**, which starts the Update Driver wizard.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_MOVED

The device has been moved.

Error Code

25

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Windows is in the process of setting up this device. (Code 25)"

"To complete the setup, click Restart Computer to restart you computer."

Recommended Resolution – Windows 98/Me

Select **Restart Computer**, which will restart the computer.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_TOO_EARLY

Too early.

Error Code

26

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Windows is in the process of setting up this device. (Code 26)"

"To complete the setup, click Restart Computer to restart you computer."

Recommended Resolution – Windows 98/Me

Select **Restart Computer**, which will restart the computer.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_NO_VALID_LOG_CONF

A valid logical configuration does not exist.

Error Code

27

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"Windows can't specify the resources for this device. (Code 27)"

"Click the Resources tab, and then select the basic configuration for the resources this device uses. To see which resources this device uses, see the documentation for this device."

Recommended Resolution – Windows 98/Me

Select resources for the device.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_FAILED_INSTALL

The device's drivers are not installed.

Error Code

28

Display Message – NT-based Systems

"The drivers for this device are not installed. (Code 28)"

Recommended Resolution – NT-based Systems

Reinstall the device's drivers.

Display Message – Windows 98/Me

"The drivers for this device are not installed (Code 28)."

"To reinstall the drivers for this device, click Reinstall Driver."

Recommended Resolution – Windows 98/Me

Select **Reinstall Driver**, which reinstalls the device's drivers.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_HARDWARE_DISABLED

The device is disabled.

Error Code

29

Display Message – NT-based Systems

"This device is disabled because the firmware of the device did not give it the required resources. (Code 29)"

Recommended Resolution – NT-based Systems

Enable the device in the BIOS.

Display Message – Windows 98/Me

"This device is disabled because the BIOS for the device did not give it any resources. (Code 29)"

"You must enable the device in the BIOS. See your hardware documentation for details, or contact your computer manufacturer to get an updated BIOS."

Recommended Resolution – Windows 98/Me

Enable the device in the BIOS.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_CANT_SHARE_IRQ

The IRQ cannot be shared.

Error Code

30

Display Message – NT-based Systems

Not used.

Recommended Resolution – NT-based Systems

Not used.

Display Message – Windows 98/Me

"This device is using an Interrupt Request (IRQ) resource that is in use by another device and cannot be shared. You must change the conflicting setting or remove the real-mode driver causing the conflict. (Code 30)"

Recommended Resolution – Windows 98/Me

Change the device's resources or remove the other device.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_FAILED_ADD

A driver's attempt to add a device failed.

Error Code

31

Display Message – NT-based Systems

"This device is not working properly because Windows cannot load the drivers required for this device. (Code 31)"

Recommended Resolution – NT-based Systems

CM_PROB_FAILED_ADD can occur if the service key is corrupted, the driver's image could not be loaded from the disk, its **DriverEntry** routine failed, it couldn't unload due to a leaked reference, it was a legacy driver that created no device objects, or its *AddDevice* routine failed.

For Windows XP and later, CM_PROB_FAILED_ADD only occurs if the driver's *AddDevice* routine fails. All the other failures have separate problem codes.

Display Message – Windows 98/Me

"This device is not working properly because <*dependent-device-name*> is not working properly. (Code 31)"

"To view the properties for <*dependent-device-name*> and see why it is not working, click Properties."

Recommended Resolution – Windows 98/Me

Select **Properties** to bring up the property page for the dependent device that has the problem.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_DISABLED_SERVICE

The driver has been disabled (NT-based systems).

The installation media is not ready (Windows 98/Me).

Error Code

32

Display Message – NT-based Systems

"A driver (service) for this device has been disabled. An alternate driver may be providing this functionality. (Code 32)"

Recommended Resolution – NT-based Systems

The start type for this service is set to Disabled in the registry. If the driver really is required, change the start type.

Display Message – Windows 98/Me

"Windows cannot install the drivers for this device because it cannot access the drive or network location that has the setup files on it. (Code 32)"

"To fix this problem, click Restart Computer to restart your computer. If that doesn't work copy all the setup files onto your local hard disk, and run setup from there."

Recommended Resolution – Windows 98/Me

Select **Restart Computer**, which will restart the computer. If that doesn't work, copy setup files to your hard disk.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_TRANSLATION_FAILED

Translator failed. (NT-based systems)

Hardware malfunction (Windows 98/Me)

Error Code

33

Display Message – NT-based Systems

"Windows cannot determine which resources are required for this device. (Code 33)"

Recommended Resolution – NT-based Systems

Configure, repair, or replace hardware.

Display Message – Windows 98/Me

If the device's hardware key contains a "FailReasonString" value, the value string is displayed as the error message. (A driver or enumerator supplies this registry string value.) If the hardware key does not contain a "FailReasonString" value, the following generic error message is displayed:

"This device isn't responding to its driver. (Code 33)"

"For more information, contact your hardware vendor."

Recommended Resolution – Windows 98/Me

Configure, repair, or replace hardware.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_NO_SOFTCONFIG

The device requires a forced configuration.

Error Code

34

Display Message – NT-based Systems

"Windows cannot determine the settings for this device. Consult the documentation that came with this device and use the Resource tab to set the configuration. (Code 34)"

Recommended Resolution – NT-based Systems

Change the HW settings (by setting jumpers or running a vendor-supplied utility) and then use the Device Manager's **Resources** tab to set the forced configuration.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_BIOS_TABLE

The MPS table is bad and needs to be updated.

Error Code

35

Display Message – NT-based Systems

"Your computer's system firmware does not include enough information to properly configure and use this device. To use this device, contact your computer manufacturer to obtain a firmware or BIOS update. (Code 35)"

Recommended Resolution – NT-based Systems

Obtain a new BIOS from the system vendor.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_IRQ_TRANSLATION_FAILED

IRQ translation failed.

Error Code

36

Display Message – NT-based Systems

"This device is requesting a PCI interrupt but is configured for an ISA interrupt (or vice versa). Please use the computer's system setup program to reconfigure the interrupt for this device. (Code 36)"

Recommended Resolution – NT-based Systems

Try using the BIOS setup utility to change settings for IRQ reservations, if such an option exists. (The BIOS might have options to reserve certain IRQs for PCI or ISA devices.)

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_FAILED_DRIVER_ENTRY

The driver returned failure from its **DriverEntry** routine.

Error Code

37

Display Message – NT-based Systems

"Windows cannot initialize the device driver for this hardware. (Code 37)"

Recommended Resolution – NT-based Systems

Reinstall or obtain a new driver.

(Note that if **DriverEntry** returns STATUS_INSUFFICIENT_RESOURCES, the error code is CM_PROB_OUT_OF_MEMORY.)

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DRIVER_FAILED_PRIOR_UNLOAD

The driver could not be loaded because a previous instance is still loaded.

Error Code

38

Display Message – NT-based Systems

"Windows cannot load the device driver for this hardware because a previous instance of the device driver is still in memory. (Code 38)"

Recommended Resolution – NT-based Systems

A previous driver instance can still be loaded due to an improper reference count or a race between load and unload operations. Additionally, this message can appear if a driver is referenced by multiple INF **AddService** directives in one ore more INF files.

Select **Restart Computer**, which will restart the computer.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_DRIVER_FAILED_LOAD

The driver could not be loaded.

Error Code

39

Display Message – NT-based Systems

"Windows cannot load the device driver for this hardware. The driver may be corrupted or missing. (Code 39)"

Recommended Resolution – NT-based Systems

Reinstall or obtain a new driver.

Reasons for this error include a driver that is not present, a binary file that is corrupted, a file I/O problem, or a driver that references an entry point in another binary that could not be loaded.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_DRIVER_SERVICE_KEY_INVALID

Information in the registry's service key for the driver is invalid.

Error Code

40

Display Message – NT-based Systems

"Windows cannot access this hardware because its service key information in the registry is missing or recorded incorrectly. (Code 40)"

Recommended Resolution – NT-based Systems

Reinstall the driver.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_LEGACY_SERVICE_NO_DEVICES

A driver was loaded but Windows cannot find the device.

Error Code

41

Display Message – NT-based Systems

"Windows successfully loaded the device driver for this hardware but cannot find the hardware device. (Code 41)"

Recommended Resolution – NT-based Systems

Reinstall the device.

This is a legacy root service that didn't create a device object.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_DUPLICATE_DEVICE

A duplicate device was detected.

Error Code

42

Display Message – NT-based Systems

"Windows cannot load the device driver for this hardware because there is a duplicate device already running in the system. (Code 42)"

Recommended Resolution – NT-based Systems

Select **Restart Computer**, which will restart the computer.

This error occurs when a bus driver erroneously creates two identically named children, or when a device with a serial number is discovered in a new location before it is noticed missing from the old location.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_FAILED_POST_START

A driver has reported a device failure.

Error Code

43

Display Message – NT-based Systems

"Windows has stopped this device because it has reported problems. (Code 43)"

Recommended Resolution – NT-based Systems

Uninstall and reinstall the device.

One of the drivers controlling the device told the OS the device failed in some manner in response to IRP_MN_QUERY_PNP_DEVICE_STATE.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_HALTED

The device has been stopped.

Error Code

44

Display Message – NT-based Systems

"An application or service has shut down this hardware device. (Code 44)"

Recommended Resolution – NT-based Systems

Solution is to reboot.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_PHANTOM

The device is not present.

Error Code

45

Display Message – NT-based Systems

"Currently, this hardware device is not connected to the computer. (Code 45)"

"To fix this problem, reconnect this hardware device to the computer."

Recommended Resolution – NT-based Systems

None. This problem code should only appear when the SHOW_NONPRESENT_DEVICES environment variable is set.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

CM_PROB_SYSTEM_SHUTDOWN

The device is not available because the system is shutting down.

Error Code

46

Display Message – NT-based Systems

"Windows cannot gain access to this hardware device because the operating system is in the process of shutting down. (Code 46)

"The hardware device should work correctly next time you start your computer."

Recommended Resolution – NT-based Systems

None. This error code is only set when Driver Verifier is enabled and all applications have already been shut down.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

CM_PROB_HELD_FOR_EJECT

The device has been prepared for ejection.

Error Code

47

Display Message – NT-based Systems

"Windows cannot use this hardware device because it has been prepared for 'safe removal', but it has not been removed from the computer. (Code 47)"

"To fix this problem, unplug this device from your computer and then plug it in again."

Recommended Resolution – NT-based Systems

Unplug the device and plug it in again. Alternately, selecting **Restart Computer** will restart the computer and make the device available.

This error should occur only if the user invoked the hotplug applet to prepare the device for removal (which calls **CM_Request_Device_Eject**), or pressed a physical eject button (which calls **IoRequestDeviceEject**). It is possible that the user prepared a device that is currently not removable, such as a CD-ROM trapped between the laptop and the docking station tray.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

CM_PROB_DRIVER_BLOCKED

The system will not load the driver because it is listed in the defective drivers database supplied by Windows Update.

Error Code

48

Display Message – NT-based Systems

"The software for this device has been blocked from starting because it is known to have problems with Windows. Contact the hardware vendor for a new driver. (Code 48)"

Recommended Resolution – NT-based Systems

Obtain a new driver from the hardware vendor.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

IDS_PROB_REGISTRY_TOO_LARGE

Error Code

49

Display Message – NT-based Systems

"Windows cannot start new hardware devices because the system hive is too large (exceeds the Registry Size Limit). (Code 49)"

Recommended Resolution – NT-based Systems

Set the environment variable DEVMGR_SHOW_NONPRESENT_DEVICES to 1, which causes the Device Manager to display installed devices that are not currently present. Use the Device Manager to remove these devices. If the registry is still too large, reinstall Windows.

Display Message – Windows 98/Me

Not used.

Recommended Resolution – Windows 98/Me

Not used.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Device Manager Details Tab

For Windows XP and later, the Device Manager provides a **Details** tab for each device. This tab displays lots of information useful to driver developers and testers, and aids Microsoft Product Support Services (PSS) in diagnosing customer problems. The tab's page displays device identification strings, along with device and driver configuration information that can be useful when debugging drivers.

Viewing a Device's Details Tab

The **Details** tab is disabled by default. To enable it, open a Command Prompt window and type the following command:

```
set DEVMGR_SHOW_DETAILS=1
```

After you have issued this command, a device's **Details** tab will be available when you select the device in the Device Manager.

Providing Firmware Revision Numbers for the Details Tab

The Device Manager's **Details** tab can display a device's firmware revision number, if available. A driver can supply a firmware revision number by responding to a WMI request. Specifically, the driver's ***DpWmiQueryDataBlock*** routine should support **MSDeviceUI_FirmwareRevision_GUID** by returning a DEVICE_UI_FIRMWARE_REVISION structure (defined in *wmidata.h*). The structure must contain the firmware revision number as a NULL-terminated WCHAR string, preceded by a USHORT value that contains the string length (including the NULL).

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Viewing Hidden Devices

The Device Manager lists the devices that are installed on the machine. By default, certain devices are not shown in the list. These "hidden" devices include devices that have the devnode status bit DN_NO_SHOW_IN_DM set and devices that are part of a setup class that is marked as a **NoDisplayClass** in the registry (for example, printers and non-PnP drivers).

To include hidden devices in the Device Manager display, click **View** and select **Show Hidden Devices**.

Another category of devices, not shown in the Device Manager by default, includes devices that have been physically removed from the machine but whose registry entries have not been deleted. These devices can be considered "nonpresent" devices. Users should never need to view such devices because a nonpresent device should not need their attention and should not cause any problems. If a user needs to view your device when it is not present, there is likely a problem with your driver design.

During testing, however, you might need to view such devices. To view nonpresent devices, open a Command Prompt window and type the following command:

```
set DEVMGR_SHOW_NONPRESENT_DEVICES=1
```

(You can also set this value in a permanent environment value using the **Advanced** tab of the system property sheet.) After you set this value, run the Device Manager and select **Show Hidden Devices**.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Using SetupAPI Logging

The Windows Setup and Device Installer Services, also known as SetupAPI, include the Windows functions that control Setup and device installation. As Setup proceeds, the general Setup functions (**Setup***Xxx* functions) and device installation functions (**SetupDi***Xxx* functions) create a log file that provides useful information for troubleshooting device installation problems.

SetupAPI logs to the file *%systemroot%\setupapi.log*. The log file is a plain text file. To reset the log, rename or delete the file.

This section includes the following information:

Setting SetupAPI Logging Levels

Interpreting a Sample SetupAPI Log File

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Setting SetupAPI Logging Levels

You can control the amount of information that is written to the SetupAPI log, either for all Setup applications or for individual Setup applications.

To change the level of information written to the SetupAPI log for all Setup applications, create (or modify) the following registry value:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup\LogLevel
‹ ›
```

By setting this value (using the values listed in the tables below) you can choose the level of errors that are logged, modify the verbosity of logging, or turn off logging. You can also log information to a debugger as well as to the log file.

To specify logging levels for individual Setup applications, create a registry entry under the following key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup\AppLogLevels
```

Under this key, create a value name representing the application's executable file name, and assign the desired logging level to that name (using the values listed in the tables below), such as **service.exe=***LoggingLevel*.

The logging level is a DWORD value. If this value is not specified or is zero, SetupAPI uses a default behavior, as indicated in the tables below.

The DWORD value is made up of three parts, formatted as 0x*SSSSDDGG*. The low eight bits, represented by the mask 0x000000FF, set the logging level for general Setup operations. The next-higher eight bits, represented by the mask 0x0000FF00, set the logging level for device installation operations. The highest bits are special flags.

The following tables contain the general logging levels, device installation logging levels, and special logging flags for Windows 2000 and later.

General Logging Levels	**Meaning**
0x00000000	Use default settings (currently 0x20).
0x00000001	Off (no Setup logging).
0x00000010	Log errors.
0x00000020	Log errors and warnings.
0x00000030	Log errors, warnings and other information.
0x00000040	Log errors, warnings and other information in verbose mode.
0x00000050	Log errors, warnings and other information in verbose mode, plus time-stamped entries.
0x00000060	Log errors, warnings and other information in verbose mode, plus time entries. Additionally, all entries are time-stamped.
0x00000070	Log errors, warnings and other information in verbose mode, plus time messages. All entries are time-stamped. Additional messages that can slow down the system, such as cache hits, are included.
0x000000FF	Specifies the most verbose logging available.

Device Logging Levels	**Meaning**
0x00000000	Use default settings (currently 0x3000).
0x00000100	Off (no Setup logging).
0x00001000	Log errors.
0x00002000	Log errors and warnings.
0x00003000	Log errors, warnings and other information.
0x00004000	Log errors, warnings and other information in verbose mode.
0x00005000	Log errors, warnings and other information in verbose mode, plus time-stamped entries.
0x00006000	Log errors, warnings and other information in verbose mode, plus time entries. Additionally, all entries are time-stamped.
0x00007000	Log errors, warnings and other information in verbose mode, plus time messages. All entries are time-stamped. Additional messages that can slow down the system, such as cache hits, are included.
0x0000FF00	Specifies the most verbose logging available.

Special Flags	Meaning
0x08000000	(*Windows XP and later*) Add a time stamp to all log entries.
0x20000000	(*Windows XP and later*) Don't flush logging information to disk after each entry is written. (Logging is faster, but information could be lost if the system crashes.)
0x40000000	Write log entries chronologically instead of grouping entries.
0x80000000	Send the output to the debugger as well as to the log file.

For example, SetupAPI interprets some sample *LoggingFlags* values as follows:

```
0x00000000 means default logging
0x0000FFFF means verbose logging
0x8000FF00 means log verbose device installation information
        to both the log file and the debugger
```

To modify the default SetupAPI logging levels during a clean installation, edit the registry during the period between text-mode Setup and GUI-mode Setup. The following steps describe the procedure. These steps assume that you are installing to *D:\Winnt* and have a working build of the same version of Windows on another partition. Change the SetupAPI logging levels as follows:

1. Start the installation of the clean build you are testing.
2. Stop the setup process during the first boot after text-mode Setup (that is, before GUI-mode Setup).
3. Boot into the working build by selecting it from the boot menu, and log on as Administrator.
4. Find the registry hives (files) in *D:\Winnt\System32\config*. In this case, you need to modify the registry hive in *Software.sav*.
5. On Windows 2000, run *Regedt32*, select the "HKEY_LOCAL_MACHINE on Local Machine" window, and select the HKEY_LOCAL_MACHINE key. Then click on the **Registry** menu and select **Load Hive**.

   On Windows XP and later, run *RegEdit*. Highlight HKEY_LOCAL_MACHINE, click on the **File** menu and select **Load Hive**.

6. Browse the files and select *D:\Winnt\System32\config\software.sav*. When prompted for key name, enter "_sw.sav "
7. Open the _sw.sav key under HKEY_LOCAL_MACHINE and highlight the following key:

   ```
   HKEY_LOCAL_MACHINE\_sw.sav\Microsoft\Windows\CurrentVersion\Setup
   ```

   On Windows 2000, click on the **Security** menu, select **Permissions**, and grant full control to Administrator.

   On Windows XP and later, click on the **Edit** menu, select **Permissions**, and grant full control to Administrator.

8. On Windows 2000, add the necessary registry values under this key using clicking on **Edit** and selecting **Add Value**.

   On Windows XP and later, click on **Edit** and select **New DWORD Value**.

   Enter the value. For example, add "0xFFFF" to enable full verbose logging.

9. Select HKEY_LOCAL_MACHINE_sw.sav, and unload the hive (using the **Registry** menu on Windows 2000, or the **File** menu on Windows XP and later)The _sw.sav key should disappear.
10. Copy *D:\Winnt\System32\config\software.sav* to *D:\Winnt\System32\config\software*.
11. Reboot and continue into Setup.
12. To verify this change, press SHIFT+F10 in GUI-mode Setup, then run *regedit.exe* and check the logging level.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Interpreting a Sample SetupAPI Log File

The sample logs below illustrate the information that is contained in a SetupAPI log file.

In general, all parts of an installation appear together in the log file. An installation section in the log starts with an entry of the format [*year/month/day time process-id.instance description*] where *instance* is a number that ensures that two sections instantiated at the same time for the same process are unique.

This topic includes:

Sample Windows 2000 SetupAPI Log File

Sample Windows XP SetupAPI Log File

Sample Windows 2000 SetupAPI Log File

In the following sample Windows 2000 log file, the indented sections are explanatory annotations. The sample log was generated with **LogLevel**=0xFFFF for verbose output:

```
[1999/10/15 15:14:25 376.1560 Driver Install]
```

SetupAPI lists the hardware and compatible IDs of the device being
installed. The first ID listed is the most exact match. Lines without a time
stamp provide context for time-stamped entries.

```
Searching for hardware ID(s): acpi\pnp0200,*pnp0200
Enumerating files E:\WINNT\inf\*.inf
@ 15:14:25.637 : Opened PNF of "E:\WINNT\inf\1394.inf".
```

PNFs are preprocessed INF files.
:
Searching other INFs...
:

```
@ 15:14:26.688 : Opened PNF of "E:\WINNT\inf\machine.inf".
```

Setup found a match here, so it logs the INF in which it found
the match, what hardware ID it matched on, descriptive information,
and the install section.

```
@ 15:14:26.698 : Found *PNP0200 in E:\WINNT\inf\machine.inf; Device:
Direct memory access controller; Driver: Direct memory access
controller; Provider: Microsoft; Mfg: (Standard system devices);
Section: NO_DRV_X
@ 15:14:26.698 : Decorated section name: NO_DRV_X
```

:
Setup searches other INFs.
:

```
@ 15:14:31.575 : Opened PNF of "E:\WINNT\inf\wordpad.inf".
```

Setup has finished searching INFs. Next, it loads class

co-installers and class installers that are registered for
this device's setup class.

```
Loading co-installer modules for Direct memory access controller.
@ 15:14:31.615 : Verifying file: E:\WINNT\System32\SysSetup.Dll using
key: SysSetup.Dll in catalog: -.
@ 15:14:31.695 : Obtained module "E:\WINNT\System32\SysSetup.Dll"
procedure "CriticalDeviceCoInstaller" for use.
```

:
Setup performs various device installation operations
before it sends a DIF_INSTALLDEVICEFILES request.
:

```
@ 15:14:32.026 : Device install function: DIF_INSTALLDEVICEFILES.
```

Setup logs each device installation function code.
Co-installers are numbered in the order that they are loaded.

```
@ 15:14:32.026 : Executing co-installer 1 of 1.
@ 15:14:32.036 : Completed co-installer 1 of 1.
```

In this example, there is no class installer for the
device's setup class. You can tell this because there is
no entry for "Executing class-installer."

```
@ 15:14:32.046 : Completed class-installer.
```

Now Setup calls the default handler for this DIF request.
If "Executing default installer" is listed before the class
installer completes, that means the class installer called
the default handler function directly.

```
@ 15:14:32.046 : Executing default installer.
@ 15:14:32.056 : Install Device: Begin.
```

Setup lists the devnode here.

```
@ 15:14:32.066 : Doing copy-only install of ACPI\PNP0200\4&12BA42B2&0.
```

Setup lists the INF and decorated INF section being used.

```
@ 15:14:32.136 : Installing section NO_DRV_X from e:\winnt\inf\machine.inf.
@ 15:14:32.146 : Install Device: Queueing files from INF(s).
@ 15:14:32.156 : Install Device: Queueing co-installer files from
INF(s).
Processing co-installer registration section NO_DRV_X.CoInstallers.
@ 15:14:32.236 : Verifying file: E:\WINNT\INF\certclas.inf using key:
certclas.inf in catalog: -.
@ 15:14:32.346 : Co-Installers Registered.
@ 15:14:32.356 : Installing section NO_DRV_X.Interfaces from
e:\winnt\inf\machine.inf.
@ 15:14:32.366 : Interfaces installed.
@ 15:14:32.376 : Device install finished successfully
(ACPI\PNP0200\4&12BA42B2&0).
@ 15:14:32.386 : Install Device: End.
@ 15:14:32.396 : Completed default installer.
```

The installation pass for copying files has finished successfully.

Now carry on with the rest of the installation operations for this device.

```
@ 15:14:32.457 : Verifying file: e:\winnt\inf\machine.inf using key:
machine.inf in catalog: -.
@ 15:14:32.497 : Verifying file: e:\winnt\inf\layout.inf using key:
layout.inf in catalog: -.
@ 15:14:32.557 : Pruning Files: Verifying catalogs/infs.
@ 15:14:32.567 : Pruning Files: Verifying catalogs/infs completed.
```

:
Setup performs various other device installation functions before sending DIF_INSTALLDEVICE.
:

```
@ 15:14:33.498 : Device install function: DIF_INSTALLDEVICE.
@ 15:14:33.498 : Executing co-installer 1 of 1.
@ 15:14:33.518 : Completed co-installer 1 of 1.
@ 15:14:33.528 : Completed class-installer.
@ 15:14:33.538 : Executing default installer.
@ 15:14:33.538 : Install Device: Begin.

@ 15:14:33.548 : Doing full install of ACPI\PNP0200\4&12BA42B2&0.
@ 15:14:33.628 : Install Device: Changing registry settings specified
by INF(s).
@ 15:14:33.638 : Install Device: Writing driver specific registry
settings.
@ 15:14:33.668 : Install Device: Installing required Windows services.
@ 15:14:33.678 : Install Device: Writing drive descriptive registry
settings.
@ 15:14:33.708 : Install Device: Restarting device.
@ 15:14:33.768 : Install Device: Restarting device completed.
@ 15:14:33.788 : Device install finished successfully
(ACPI\PNP0200\4&12BA42B2&0).
@ 15:14:33.798 : Install Device: End.
@ 15:14:33.798 : Completed default installer.
@ 15:14:33.808 : Executing co-installer (Post Processing).
@ 15:14:33.838 : Completed co-installer 1 (Post Processing).
```

:
Setup performs various device installation functions after sending DIF_INSTALLDEVICE.
:

Sample Windows XP SetupAPI Log File

For Windows XP, each log entry includes a message identifier consisting of a letter or dash (-) followed by a number. The following table describes the format for message identifiers.

Message ID	Message Type
#E*nnn*	Error message.
#W*nnn*	Warning message.
#I*nnn*	Informational message.
#T*nnn*	Timing message.
#V*nnn*	Verbose message.
#-*nnn*	Status message.

Following are a couple of segments from a Windows XP error log:

```
[2001/02/27 20:14:30 1148.173]
#-198 Command line processed: C:\WINDOWS\system32\mmc.exe "C:\WINDOWS\system32\devmgmt.msc"
#-147 Loading class installer module for "Communications Port".
@ 20:14:33.381 #V132 File "C:\WINDOWS\INF\certclas.inf" (key "certclas.inf") is signed in cata
@ 20:14:33.810 #V132 File "C:\WINDOWS\System32\MsPorts.Dll" (key "MsPorts.Dll") is signed in
@ 20:14:33.873 #V146 Using exported function "PortsClassInstaller" in module "C:\WINDOWS\Syste
@ 20:14:33.873 #V166 Device install function: DIF_UPDATEDRIVER_UI.
@ 20:14:33.873 #T152 Executing class installer.
@ 20:14:33.873 #V153 Completed class installer.
@ 20:14:33.889 #V155 Executing default installer.
@ 20:14:33.889 #V156 Completed default installer.
@ 20:14:36.302 #I060 Set selected driver.
#-019 Searching for hardware ID(s): *pnp0501
@ 20:14:36.318 #V017 Enumerating files "C:\WINDOWS\inf".
(following #V lines are logged only if driver bits of log level >= 0x7000)
@ 20:14:36.556 #V392 Using INF cache "C:\WINDOWS\inf\INFCACHE.1".
@ 20:14:36.731 #V073 Cache: Excluding INF "accessor.inf".
@ 20:14:36.731 #V073 Cache: Excluding INF "agtinst.inf".
:
:
@ 20:14:37.318 #T075 Enumerating files: Directory pass completed.
@ 20:14:37.398 #V005 Opened the PNF file of "C:\WINDOWS\inf\msports.inf" (Languge = 0409).
@ 20:14:37.413 #I022 Found "*PNP0501" in C:\WINDOWS\inf\msports.inf; Device: "Communications
(rank of 0 is absolute best match)
@ 20:14:37.413 #I023 Actual install section: [ComPort.NT]. Rank: 0x00001000. Effective driver
@ 20:14:37.413 #I022 Found "*PNP0501" in C:\WINDOWS\inf\msports.inf; Device: "Communications
@ 20:14:37.413 #I023 Actual install section: [ComPort.NT]. Rank: 0x00000000. Effective driver
@ 20:14:37.413 #T076 Enumerating files: Cache pass completed.
@ 20:15:01.383 #V166 Device install function: DIF_SELECTBESTCOMPATDRV.
@ 20:15:01.383 #T152 Executing class installer.
@ 20:15:01.383 #V153 Completed class installer.
@ 20:15:01.399 #V155 Executing default installer.
@ 20:15:01.399 #I063 Selected driver installs from section [ComPort] in "c:\windows\inf\msport
@ 20:15:01.526 #I320 Class GUID of device remains {4D36E978-E325-11CE-BFC1-08002BE10318}.
@ 20:15:01.526 #I060 Set selected driver.
@ 20:15:01.526 #I058 Selected best compatible driver.
@ 20:15:01.526 #V156 Completed default installer.
@ 20:15:02.447 #V166 Device install function: DIF_ALLOW_INSTALL.
@ 20:15:02.447 #T152 Executing class installer.
@ 20:15:02.447 #V153 Completed class installer.
@ 20:15:02.447 #V155 Executing default installer.
@ 20:15:02.447 #V156 Completed default installer.
@ 20:15:02.463 #V166 Device install function: DIF_INSTALLDEVICEFILES.
@ 20:15:02.463 #T152 Executing class installer.
@ 20:15:02.463 #V153 Completed class installer.
@ 20:15:02.463 #V155 Executing default installer.
@ 20:15:02.463 #T200 Install Device: Begin.
@ 20:15:02.478 #V124 Doing copy-only install of "ROOT\*PNP0501\PNPBIOS_17".
@ 20:15:02.478 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:02.478 #V005 Opened the PNF file of "c:\windows\inf\layout.inf" (Languge = 0409).
@ 20:15:02.494 #V011 Installing section [ComPort.NT] from "c:\windows\inf\msports.inf".
@ 20:15:02.494 #T203 Install Device: Queuing files from INF(s).
@ 20:15:02.510 #V005 Opened the PNF file of "C:\WINDOWS\INF\drvindex.inf" (Languge = 0409).
@ 20:15:02.590 #V094 Queued copy from section [ComPort.NT.Copy] in "c:\windows\inf\msports.in
@ 20:15:02.590 #V096 Source in section [sourcedisksfiles] in "c:\windows\inf\layout.inf"; Med
@ 20:15:02.605 #V132 File "C:\WINDOWS\INF\certclas.inf" (key "certclas.inf") is signed in cata
@ 20:15:02.605 #V005 Opened the PNF file of "C:\WINDOWS\INF\certclas.inf" (Languge = 0409).
@ 20:15:02.685 #V094 Queued copy from section [ComPort.NT.Copy] in "c:\windows\inf\msports.in
@ 20:15:02.685 #V096 Source in section [sourcedisksfiles] in "c:\windows\inf\layout.inf"; Med
@ 20:15:02.685 #T204 Install Device: Queuing coinstaller files from INF(s).
@ 20:15:02.685 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:02.701 #V005 Opened the PNF file of "c:\windows\inf\layout.inf" (Languge = 0409).
#-046 Processing Coinstaller registration section [ComPort.NT.CoInstallers].
@ 20:15:02.701 #V056 Coinstallers registered.
@ 20:15:02.717 #V011 Installing section [ComPort.NT.Interfaces] from "c:\windows\inf\msports.
@ 20:15:02.732 #V054 Interfaces installed.
@ 20:15:02.732 #V121 Device install of "ROOT\*PNP0501\PNPBIOS_17" finished successfully.
@ 20:15:02.732 #T201 Install Device: End.
@ 20:15:02.748 #V156 Completed default installer.
@ 20:15:02.748 #T185 Pruning Files: Verifying catalogs/infs.
@ 20:15:02.764 #V132 File "c:\windows\inf\msports.inf" (key "msports.inf") is signed in catal
@ 20:15:02.812 #V132 File "c:\windows\inf\layout.inf" (key "layout.inf") is signed in catalog
@ 20:15:02.812 #T186 Pruning Files: Verifying catalogs/infs completed.
@ 20:15:02.859 #V132 File "C:\WINDOWS\System32\DRIVERS\serial.sys" (key "serial.sys") is sign
@ 20:15:02.875 #V191 File "C:\WINDOWS\System32\DRIVERS\serial.sys" pruned from copy.
@ 20:15:02.875 #V132 File "C:\WINDOWS\System32\DRIVERS\serenum.sys" (key "serenum.sys") is si
@ 20:15:02.875 #V191 File "C:\WINDOWS\System32\DRIVERS\serenum.sys" pruned from copy.
@ 20:15:02.875 #V166 Device install function: DIF_REGISTER_COINSTALLERS.
@ 20:15:02.923 #T152 Executing class installer.
@ 20:15:02.939 #V153 Completed class installer.
@ 20:15:02.939 #V155 Executing default installer.
@ 20:15:02.939 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:02.939 #I056 Coinstallers registered.
@ 20:15:02.955 #V156 Completed default installer.
@ 20:15:02.955 #V166 Device install function: DIF_INSTALLINTERFACES.
@ 20:15:02.955 #T152 Executing class installer.
@ 20:15:02.955 #V153 Completed class installer.
@ 20:15:02.955 #V155 Executing default installer.
@ 20:15:02.971 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:02.971 #V011 Installing section [ComPort.NT.Interfaces] from "c:\windows\inf\msports.
@ 20:15:02.986 #I054 Interfaces installed.
@ 20:15:02.986 #V156 Completed default installer.
@ 20:15:02.986 #V166 Device install function: DIF_INSTALLDEVICE.
@ 20:15:03.002 #T152 Executing class installer.
@ 20:15:03.002 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:03.018 #T200 Install Device: Begin.
@ 20:15:03.018 #I123 Doing full install of "ROOT\*PNP0501\PNPBIOS_17".
@ 20:15:03.018 #V005 Opened the PNF file of "c:\windows\inf\msports.inf" (Languge = 0409).
@ 20:15:03.018 #T211 Install Device: Changing registry settings as specified by the INF(s).
@ 20:15:03.034 #T212 Install Device: Writing driver specific registry settings.
@ 20:15:03.050 #T213 Install Device: Installing required Windows services.
#-035 Processing service Add/Delete section [ComPort.NT.Services].
@ 20:15:03.320 #V282 Add Service: Modified existing service "Serial".
@ 20:15:03.653 #V282 Add Service: Modified existing service "Serenum".
@ 20:15:03.669 #T214 Install Device: Writing drive descriptive registry settings.
@ 20:15:03.669 #T216 Install Device: Restarting device.
@ 20:15:06.399 #T217 Install Device: Restarting device completed.
@ 20:15:06.399 #W114 Device "ROOT\*PNP0501\PNPBIOS_17" required reboot: Device has problem: 0
@ 20:15:06.399 #T222 Install Device: Calling RunOnce/GrpConv items.
@ 20:15:06.415 #I138 Executing RunOnce to process 5 RunOnce entries.
@ 20:15:07.241 #I121 Device install of "ROOT\*PNP0501\PNPBIOS_17" finished successfully.
@ 20:15:07.272 #T201 Install Device: End.
@ 20:15:07.336 #V153 Completed class installer.
@ 20:15:07.368 #V166 Device install function: DIF_NEWDEVICEWIZARD_FINISHINSTALL.
@ 20:15:07.368 #T152 Executing class installer.
@ 20:15:07.495 #V153 Completed class installer.
@ 20:15:07.495 #V155 Executing default installer.
@ 20:15:07.495 #V156 Completed default installer.
@ 20:15:42.100 #V166 Device install function: DIF_DESTROYPRIVATEDATA.
@ 20:15:42.100 #T152 Executing class installer.
@ 20:15:42.100 #V153 Completed class installer.
@ 20:15:42.116 #V155 Executing default installer.
@ 20:15:42.116 #V156 Completed default installer.
```

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Updating Driver Files

Drivers are updated when a user runs the Hardware Update Wizard. Following is a set of guidelines to follow when writing installation software and INF files that update existing drivers.

- Installation software can call **UpdateDriverForPlugAndPlayDevices**, supplying an INF file and a hardware ID, to update drivers for devices that match the hardware ID. For more information, see Writing a Device Installation Application.
- When upgrading a driver, class installers and co-installers should not supply finish-install pages in response to DIF_NEWDEVICEWIZARD_FINISHINSTALL unless absolutely necessary. If possible, obtain finish-install information from the settings of the previous installation.
- To the extent possible, class installers and co-installers should avoid basing behavior on whether they are providing an initial installation or are updating drivers for an already-installed device.
- For Windows XP and later, the registry values **CoInstallers32** and **EnumPropPages32** are deleted prior to delivery of DIF_REGISTER_COINSTALLERS. INF files for earlier OS versions must explicitly either delete these values or perform a non-appending modify operation on them.
- For Windows XP and later, the registry values **UpperFilters** and **LowerFilters** are deleted prior to delivery of DIF_INSTALLDEVICE. INF files for earlier OS versions must explicitly either delete these values or perform a non-appending modify operation on them.
- Do *not* use INF **DelFiles** directives or INF **RenFiles** directives when updating drivers. Setup cannot guarantee that a particular file is not being used by another device. (Class installers and co-installers can delete or rename files, *if* they can reliably determine that no devices are using the files.)
- Use the INF **DelReg** directive to remove old device-specific registry entries from a previous installation of the device, if the entries are no longer needed. (Do not remove global registry entries.)
- Do *not* use the INF **DelService** directive in an INF *DDInstall*.**Services** section to remove previously installed device/driver services from the target machine. Setup cannot guarantee that a particular service is not being used by another device. (Class installers and co-installers can delete services, *if* they can reliably determine that no devices are using the services.)
- When updating a class installer, class co-installer, or service DLL, you must give the new version a new file name.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Drivers with Special Installation Requirements

This chapter describes how to install boot, filter, and "null" drivers. It contains the following sections:

Installing a Boot Driver

Installing a Filter Driver

Installing a Null Driver

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Installing a Boot Driver

During the text-mode setup phase of Windows 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 user can install an additional third-party driver during text-mode setup by pressing the appropriate key and inserting a floppy disk from the third-party vendor. A third-party vendor provides a *txtsetup.oem* file on a floppy disk to enable a user to install such a driver during text-mode setup.

To install a device that is required to boot the machine, a user installs the device hardware, turns on the machine, and installs the device's driver (if it is not included with the operating system), in the following manner:

- During the beginning of the text-mode setup phase of booting, the system displays a message indicating that the user can press a key to install custom drivers.
- At this point, the user presses an **F***n* key to install a driver for a third-party device, then inserts the vendor's floppy disk.

Certain error messages displayed when a machine fails to boot can indicate that a boot driver is missing. For example:

- If a machine fails to boot and displays the message "Inaccessible boot device," it might be because the boot disk is a third-party mass-storage device that requires a driver not included with the operating system.
- If a machine fails to boot and displays the message "Setup could not determine your machine type," it could be because a new HAL driver is required. This error does not occur on most machines, but it might occur on a high-end server.

Note that this procedure is for installing a driver that is not included "in-the-box" with the operating system. Do not use this procedure to replace or update a driver included with the operating system. Instead, wait until the system boots and use the Device Manager to perform an "update driver" operation on the device.

If a vendor distributes a driver for a device required during boot and intends for the user to install it during text-mode setup, the following rules apply:

- Driver file names must not exceed eight characters, and extensions must not exceed three characters.
- The vendor's distribution disk must include a file named *txtsetup.oem* in addition to the other driver files.

A *txtsetup.oem* file is a text file that contains a list of hardware components, a list of files to be copied, and a list of registry keys and vaules to be created. For details about the contents of a *txtsetup.oem* file, see The *txtsetup.oem* File Format.

A single distribution disk with a single *txtsetup.oem* file can support multiple hardware devices. Such a file would contain a single **Defaults** section with an entry specifying the default choice for each hardware type. The file would also contain a *HwComponent* entry for each hardware type and the appropriate associated **Files.***HwComponent.ID* and **Config.***DriverKey* sections.

A sample *txtsetup.oem* file is provided with this DDK, under the installed DDK's \src directory.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Installing a Filter Driver

A PnP filter driver can support a specific device or all devices in a setup class and can attach below a device's function driver (a lower filter) or above a device's function driver (an upper filter). See Types of WDM Drivers for more information on PnP driver layers.

Installing a Device-Specific Filter Driver

To register a device-specific filter driver, create a registry entry through an **AddReg** entry in the *DDInstall***.HW** section of the device's INF file. For a device-specific upper filter, create an entry named **UpperFilters**. For a device-specific lower filter, create an entry named **LowerFilters**. For example, the following INF excerpt installs *cdaudio* as an upper filter on the *cdrom* driver:

```
:
; 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=cdaudio_addreg

[cdaudio_install.Services]
AddService=cdrom,0x00000002,cdrom_ServiceInstallSection
AddService=cdaudio,,cdaudio_ServiceInstallSection
:

[cdaudio_addreg]
HKR,,"UpperFilters",0x00010000,"cdaudio" ; REG_MULTI_SZ value
:

[cdaudio_ServiceInstallSection]
DisplayName    = %cdaudio_ServiceDesc%
ServiceType    = 1      ; SERVICE_KERNEL_DRIVER
StartType      = 3      ; SERVICE_DEMAND_START
ErrorControl   = 1      ; SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\cdaudio.sys
:
```

Installing a Class Filter Driver

To install a class-wide upper- or lower-filter for a device setup class, you can supply a device installation application that installs the necessary services. The application can then register the service as being an upper- or lower-filter for the desired device setup classes. To copy the service binaries, the application can use **SetupInstallFilesFromInfSection**. To install the services, the application can use **SetupInstallServicesFromInfSection**. To register the services as upper- and/or lower-filters for particular device setup classes, the application calls **SetupInstallFromInfSection** for each device setup class of interest, using the registry key handle they retrieved from **SetupDiOpenClassRegKey** for the *RelativeKeyRoot* parameter. For example, consider the following INF sections:

```
:

[DestinationDirs]
upperfilter_copyfiles = 12

[upperfilter_inst]
CopyFiles = upperfilter_copyfiles
AddReg = upperfilter_addreg

[upperfilter_copyfiles]
upperfilt.sys

[upperfilter_addreg]
; append this service to existing REG_MULTI_SZ list, if any
HKR,,"UpperFilters",0x00010008,"upperfilt"
```

```
[upperfilter_inst.Services]
AddService = upperfilt,,upperfilter_service

[upperfilter_service]
DisplayName   = %upperfilter_ServiceDesc%
ServiceType   = 1   ; SERVICE_KERNEL_DRIVER
StartType     = 3   ; SERVICE_DEMAND_START
ErrorControl  = 1   ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\upperfilt.sys
:
```

The device installation application would:

1. Call **SetupInstallFilesFromInfSection** for the [upperfilter_inst] section.
2. Call **SetupInstallServicesFromInfSection** for the [upperfilter_inst.Services] section.
3. Call **SetupInstallFromInfSection** for the [upperfilter_inst] section, once for each class key it wants to register the *upperfilt* service for.

Each call would specify **SPINST_REGISTRY** for the *Flags* argument, to indicate that only registry modifications need to be performed.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Installing a Null Driver

You might install a "null driver" (that is, non-existent driver) for a device if the device is not used on the machine and should not be started. Such devices do not typically exist on a machine, but if they do, you can install a null driver by specifying entries like the following in the INF for the device:

```
:
[MyModels]
%MyDeviceDescription% = MyNullInstallSection, &BadDeviceHardwareID%
:

[MyNullInstallSection]
; The install section is typically empty, but can contain entries that
; copy files or modify the registry.

[MyNullInstallSection.Services]
AddService = ,2    ; no value for the service name
:
```

The hardware ID for the device in the *Models* section should identify the device specifically, using the subsystem vendor ID and whatever other information is relevant.

The operating system will create a devnode for the device, but if the device is not capable of executing in raw mode, the operating system will not start the device because a function driver has not been assigned to it. Note, however, that if the device has a boot configuration, those resources will be reserved.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Device Identification Strings

When a Plug and Play bus enumerator detects a child device, it constructs a list of device identification strings for the child. One entry in the list is a complete identification of the device. This entry is called the *device ID* and it becomes the name of the device's hardware key in the registry. Additional entries in the list are less specific.

The Plug and Play Manager uses all of the identifiers in the list when it tries to match a device to an INF file, but it tries to use the most specific identifier first. This allows the setup software to give preference to drivers in the order of their suitability, those drivers supplied by the vendor being at the top of the list.

For a detailed discussion of the role device identification strings play in the selection of drivers, and the proper means of locating and ordering them in a driver's INF file, see How Setup Selects Drivers.

Identifiers are divided into the following groups:

Device IDs

Hardware IDs

Compatible IDs

Instance IDs

Device instance IDs

This section deals with the first three of these types of IDs.

The most specific of the first three types of identifiers is the device ID. Each device has only one device ID, and the device ID uniquely identifies the device.

The next most specific is the hardware ID. In most cases, the device ID is just the most specific of the hardware IDs, although there are exceptions (see Identifiers for 1394 Devices). The least specific of the first three types of identifier is the compatible ID. A device can only have one device ID.

Drivers can retrieve a device's hardware IDs by calling **IoGetDeviceProperty** with the *DeviceProperty* parameter set to **DevicePropertyHardwareID**, or by querying the bus driver directly with an IRP of type **IRP_MN_QUERY_ID** whose **Parameters.QueryId.IdType** field is set to **BusQueryHardwareIDs**.

The device ID is usually the first identifier in the list of hardware IDs obtained by either of these methods, but drivers can specifically query for the device ID alone by means of an IRP of type **IRP_MN_QUERY_ID** whose **Parameters.QueryId.IdType** field is set to **BusQueryDeviceID**.

Drivers can retrieve a device's compatible IDs by calling **IoGetDeviceProperty** with the *DeviceProperty* parameter set to **DevicePropertyCompatibleID**, or by querying the bus driver directly with an IRP of type **IRP_MN_QUERY_ID** whose **Parameters.QueryId.IdType** field is set to **BusQueryCompatibleIDs**.

Formats for hardware and compatible IDs vary according to the enumerator. Furthermore, each enumerator has its own policy for defining which identifiers are hardware IDs and which identifiers are compatible IDs.

The following sections provide additional information about device, hardware, and compatible IDs for devices connected to some of the most important buses:

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Generic Identifiers

Most, but not all, identifier strings are bus-specific. The Plug and Play Manager also supports a set of generic device identifiers for devices that can appear on many different buses. These identifiers are of the form:

*PNP*d(4)*

where *d(4)* is a 4-digit, hexadecimal type identifier. In the case of the PCMCIA bus, compatible IDs are formatted in this manner (see the following discussion of the PCMCIA bus). The official list of these identifiers can be found on the Microsoft hardware developer's website at http://www.microsoft.com/hwdev. This information is also published in the Microsoft® MSDN™.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Identifiers for PCI Devices

The device ID of a PCI device has the form

PCI\VEN_*v(4)*&DEV_*d(4)*&SUBSYS_*s(8)*&REV_*r(2)*

where *v(4)* is the vendor identifier that the PCI Special Interest Group assigned to the manufacturer of the card, *d(4)* is the device identifier that the manufacturer assigned to the card, *s(8)* is the subsystem identifier (often zero) reported by the card, and *r(2)* is the revision number.

An example of how this identifier might appear for a display adapter on a laptop computer is:

PCI\VEN_102C&DEV_00E0&SUBSYS_00000000&REV_04

The PCI bus driver supplies two hardware IDs. The format of the second hardware ID is the same as the first, but without the revision information.

PCI\VEN_*v(4)*&DEV_*d(4)*&SUBSYS_*s(8)*

In addition to the two hardware IDs, the PCI bus driver also generates between eight and ten compatible IDs, depending on whether the system identifier is zero or not. If there is no system ID, then the two hardware IDs indicated above collapse into the format of the first two compatible IDs in the list below, and in that case, these two IDs cease to be compatible IDs and become hardware IDs:

PCI\VEN_*v(4)*&DEV_*d(4)*&REV_*r(2)* (compatible ID only if there is system ID, otherwise a hardware ID)

PCI\VEN_*v(4)*&DEV_*d(4)* (compatible ID only if there is system ID, otherwise a hardware ID)

PCI\VEN_*v(4)*&DEV_*d(4)*&REV_*r(2)*&CC_c(2)s(2)

PCI\VEN_*v(4)*&DEV_*d(4)*&CC_c(2)s(2)*p(2)*

PCI\VEN_*v(4)*&DEV_*d(4)*&CC_c(2)s(2)

PCI\VEN_*v(4)*&CC_c(2)s(2)*p(2)*

PCI\VEN_*v(4)*&CC_c(2)s(2)

PCI\VEN_*v(4)*

PCI\CC_c(2)s(2)*p(2)*

PCI\CC_c(2)s(2)

In the above list, *c(2)* represents the base class code from the configuration space, *s(2)* is the subclass code, and *p(2)* is the programming interface. Thus, any of the following compatible IDs for the display adapter in the previous example would have matched the information in an INF file for that adapter:

PCI\VEN_102C&DEV_00E0&REV_04

PCI\VEN_102C&DEV_00E0

PCI\VEN_102C&DEV_00E0&REV_04&CC_0300

PCI\VEN_102C&DEV_00E0&CC_030000

PCI\VEN_102C&DEV_00E0&CC_0300

PCI\VEN_102C&CC_030000

PCI\VEN_102C&CC_0300

PCI\VEN_102C

PCI\CC_030000

PCI\CC_0300

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Identifiers for SCSI Devices

The device ID format for a small computer system interface (SCSI) device is:

SCSI\tv(8)p(16)r(4)*

where *t** is a device type code of variable length, *v(8)* is an 8-character vendor identifier, *p(16)* is a 16-character product identifier, and *r(4)* is a 4-character revision level value. The bus enumerator determines the device type by indexing an internal string table, using a numerically encoded SCSI device type code, obtained by querying the device, as shown in the following table. The remaining components are just strings returned by the device, but with special characters (including space, comma, and any nonprinting graphic) replaced with an underscore.

The ScsiPort driver currently returns the following device type strings, the first nine of which correspond to standard SCSI type codes.

SCSI Type Code	Device Type	Generic Type	Peripheral ID
DIRECT_ACCESS_DEVICE (0)	Disk	GenDisk	DiskPeripheral
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential		TapePeripheral
PRINTER_DEVICE (2)	Printer	GenPrinter	PrinterPeripheral
PROCESSOR_DEVICE (3)	Processor		OtherPeripheral
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm	WormPeripheral
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom	CdRomPeripheral
SCANNER_DEVICE (6)	Scanner	GenScanner	ScannerPeripheral
OPTICAL_DEVICE (7)	Optical	GenOptical	OpticalDiskPeripheral
MEDIUM_CHANGER (8)	Changer	ScsiChanger	MediumChangerPeripheral
COMMUNICATION_DEVICE (9)	Net	ScsiNet	CommunicationsPeripheral
10	ASCIT8	ScsiASCIT8	ASCPrePressGraphicsPeripheral
11	ASCIT8	ScsiASCIT8	ASCPrePressGraphicsPeripheral
12	Array	ScsiArray	ArrayPeripheral
13	Enclosure	ScsiEnclosure	EnclosurePeripheral
14	RBC	ScsiRBC	RBCPeripheral
15	CardReader	ScsiCardReader	CardReaderPeripheral
16	Bridge	ScsiBridge	BridgePeripheral
17	Other	ScsiOther	OtherPeripheral

An example of a device ID for a disk drive would be:

SCSI\DiskSEAGATE_ST39102LW_______0004

There are four hardware IDs in addition to the device ID:

SCSI\tv(8)p(16)*

SCSI\tv(8)*

SCSI\v(8)p(16)r(1)

V(8)p(16)r(1)

In the third and fourth of these additional identifiers, *r(1)* represents just the first character of the revision identifier. These hardware IDs are illustrated by the following examples:

SCSI\DiskSEAGATE_ST39102LW_______

SCSI\DiskSEAGATE_

SCSI\DiskSEAGATE_ST39102LW_______0

SEAGATE_ST39102LW_______0

The ScsiPort driver supplies only one compatible ID, one of the variable-sized generic type codes from the preceding table.

For example, the compatible ID for a disk drive is:

GenDisk

The generic identifier is used in INF files for SCSI devices more than any other, because SCSI drivers tend to be generic. Note that ScsiPort returns no generic name at all for sequential access and "processor" devices.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Identifiers for IDE Devices

Identifiers for integrated device electronics (IDE) devices are very similar to SCSI identifiers. The device ID format is:

IDE\tv(40)r(8)*

where *t** is a device-type code of variable length, *v(40)* is a string containing the vendor name, an underscore, vendor's product name, and enough underscores to bring the total to 40 characters; *r(8)* is an 8-character revision number.

There are three hardware IDs, in addition to the device ID:

IDE\v(40)r(8)

IDE\tv(40)*

V(40)r(8)

As in the SCSI case, there is only one compatible ID, a generic type name similar to the standard SCSI type names supplied by ScsiPort driver, but having only eleven entries instead of eighteen. The generic device-type names for IDE devices are as follows:

IDE Type Code	Device Type	Generic Type	Peripheral ID
DIRECT_ACCESS_DEVICE (0)	Disk	GenDisk	DiskPeripheral
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential	GenSequential	TapePeripheral
PRINTER_DEVICE (2)	Printer	GenPrinter	PrinterPeripheral
PROCESSOR_DEVICE (3)	Processor	GenProcessor	OtherPeripheral
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm	WormPeripheral
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom	CdRomPeripheral
SCANNER_DEVICE (6)	Scanner	GenScanner	ScannerPeripheral
OPTICAL_DEVICE (7)	Optical	GenOptical	OpticalDiskPeripheral
MEDIUM_CHANGER (8)	Changer	GenChanger	MediumChangerPeripheral
COMMUNICATION_DEVICE (9)	Net	GenNet	CommunicationsPeripheral
10	Other	GenOther	OtherPeripheral

For IDE changer devices, the generic type name is **GenChanger** instead of **ScsiChanger** and for communication devices the generic type name is **GenNet** instead of **ScsiNet**. ScsiPort returns no generic name at all for sequential access and "processor" devices, whereas the IDE bus driver returns **GenSequential** and **GenProcessor**. Also, the IDE bus driver returns only ten generic types, whereas ScsiPort currently returns eighteen. In other respects, the generic names returned by the IDE bus driver are the same as those returned by the ScsiPort driver.

The compatible ID for an IDE tape drive is:

GenSequential

In the special case of an LS-120 device, the IDE bus driver returns the following compatible ID:

GenSFloppy

The following illustrates the sort of identifiers that can be generated for an IDE hard drive:

IDE\DiskMaxtor_91000D8_____________________SASX1B18

IDE\Maxtor_91000D8_________________________SASX1B18

IDE\DiskMaxtor_91000D8_____________________

Maxtor_91000D8_________________________SASX1B18

GenDisk

Built on Thursday, July 19, 2001

Send feedback to MSDN. *Look here for MSDN Online resources.*

Device Installation: Windows DDK

Identifiers for PCMCIA Devices

For Personal Computer Memory Card International Association (PCMCIA) devices, the device ID can take a number of different forms. For devices that are not multifunctional, the device identifier is formatted as follows:

PCMCIA*Manufacturer-Product-Crc(4)*

where **Manufacturer** is the name of the manufacturer and **Product** is the name of the product. The PCMCIA enumerator retrieves these strings directly from tuples on the card. Both **Manufacturer** and **Product** are variable-length strings that do not exceed 64 characters. *Crc(4)* is the 4-digit hexadecimal CRC (cyclic redundancy check) checksum for the card. Numbers less than four digits long have a leading zero fill. For example, the device ID for a network card might look like this:

PCMCIA\MEGAHERTZ-CC10BT/2-BF05

For a multifunction card, each individual function has an identifier of the form:

PCMCIA*Manufacturer-Product*-DEV*d(4)-Crc(4)*

The child function number (*d(4)* in this example) is a decimal number without leading zeros.

If the card doesn't have a name of the manufacturer, the identifier has one of these three forms:

PCMCIA\UNKNOWN_MANUFACTURER-*Crc(4)*

PCMCIA\UNKNOWN_MANUFACTURER-DEV*d(4)-Crc(4)*

PCMCIA\MTD-*MemoryType(4)*

The last of these three alternatives is for a flash memory card with no manufacturer identifier on the card. *MemoryType(4)* is one of two 4-digit hexadecimal numbers: 0000 for SRAM cards and 0002 for ROM cards.

In addition to the various forms of device ID just described, an INF file's **Model** section can also contain a hardware ID composed by replacing the 4-digit hexadecimal cyclic redundancy check (CRC) with a string containing the 4-digit hexadecimal manufacturer code, a hyphen, and the 4-digit hexadecimal manufacturer information code (both from onboard tuples). For example:

PCMCIA\MEGAHERTZ-CC10BT/2-0128-0103

PCMCIA-compatible IDs correspond to the generic device IDs mentioned in the Generic Identifiers section. Currently, PCMCIA-compatible IDs are generated for only three device types:

*PNP0600

*PNP0D00

*PNPC200

an AT Attachment (ATA) disk driver, a multi-function 3.0 PC card, and a modem card respectively.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Identifiers for ISAPNP Devices

Every ISAPNP card supports a readable resource data structure that describes the resources supported and those requested by the card. This structure supports the concept of multiple functions (or "logical devices") for ISA card. A separate set of "tags" or "descriptors" are associated with each function of the card. Using this tag information, the ISAPNP enumerator constructs two hardware identifiers, formatted as:

ISAPNP\m(3)d(4)

**m(3)n(4)*

where *m(3)d(4)* together make up an EISA-style identifier for the device — three letters to identify the manufacturer and 4 hexadecimal digits to identify the particular device.

The following pair of hardware IDs might be produced by a specific function on a multifunction card:

ISAPNP\CSC6835_DEV0000

*CSC0000

The first of the two hardware IDs is the device ID. If the device in question is one function of a multifunction card, the device ID takes this form:

ISAPNP\m(3)d(4)_DEV*n(4)*

where *n(4)* is the decimal index (with leading zeros) of the function.

The second of the two hardware identifiers is also a compatible ID. The ISAPNP enumerator generates one or more compatible IDs the first of which is always the second hardware ID. The first three characters, *m(3)*, that follow the "*" in an ISAPNP-compatible ID are frequently "PNP." For example, the compatible ID for a serial port might look like this:

PNP0501

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Identifiers for 1394 Devices

The 1394 bus driver constructs these identifiers for a device:

1394\VendorName&*ModelName*

1394\UnitSpecId&*UnitSwVersion*

where *VendorName* is the name of the hardware vendor, *ModelName* identifies the device, *UnitSpecId* identifies the software specification authority, and *UnitSwVersion* identifies the software specification. The information used to construct these identifiers comes from the device's configuration ROM.

If a device has vendor and model name strings, the 1394 bus driver uses the first identifier as both, the device ID and the hardware ID, and the second identifier as the compatible ID. If a device lacks a vendor or model name string, the bus driver uses the second identifier as both, the device ID and the compatible ID, and returns double NULL if queried for the hardware ID. Thus, the IEEE1394 bus driver, under certain circumstances, supplies a device ID but no hardware ID. This is an exception to the general rule that the device ID is one of the hardware IDs.

The device ID for a camera on a IEEE1394 might appear as follows

1394\SONY&CCM-DS250_1.08

Multifunction devices have a separate set of identifiers for each unit directory in the device's configuration ROM.

If the device's function driver sits on top of the SBP-2 port driver, then its device ID has the following format.

SBP2\VendorName&ModelName&LUNn*

where *VendorName* is the hardware vendor, *ModelName* identifies the device, and *n** is a string representing the lower-order 2 bytes of the logical unit number in hexadecimal. Various functions on a multifunction device produce device IDs that are identical except for this number.

The device ID for a SBP-2 1394 hard disk might be as follows:

SBP2\VST_TECHNOLOGIESINC.&VST_FULL_HEIGHT_FIREWIRE_DRIVE&LUN0

As with the 1394 bus, the SBP2 port driver does not classify the device ID as a hardware ID. However, whereas the 1394 bus distinguishes between hardware IDs and compatible IDs, the SBP2 port driver does not. For **IRP_MN_QUERY_ID** IRPs of type **BusQueryHardwareIDs** and **IRP_MN_QUERY_ID** IRPs of type **BusQueryCompatibleIDs** SBP2 returns the same set of four identifiers:

SBP2*VendorName*&*ModelName*&CmdSetId*n**

SBP2*Gen*

Gen

SBP2*n**&*d**

where *n** is the Command Set ID number, *Gen* is one of the generic names listed in the *Generic Type* column of the table below, and *d** is a number formed by taking the lower five bits of the upper two bytes of the logical unit number. This number is the numeric code for the generic name of the device corresponding to the string identifier *Gen*. The fourth ID, listed in the preceding example (SBP2*n**&*d**), is unique among all of the SBP2 hardware identifiers in that both *n**, the Command Set ID number and *d**, the numeric code of the generic name are in decimal, not hexadecimal.

This table lists the generic device names returned by the SBP2 port driver. Most, but not all, of the generic names generated by the SBP2 port driver are a subset of those generated by ScsiPort driver.

1394 Type Code	**Device Type**	**Generic Type**
RBC_DEVICE or DIRECT_ACCESS_DEVICE (0)	Disk	GenDisk
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential	GenSequential
PRINTER_DEVICE (2)	Printer	GenPrinter
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom
SCANNER_DEVICE (6)	Scanner	GenScanner

OPTICAL_DEVICE (7)	Optical	GenOptical
MEDIUM_CHANGER (8)	Changer	GenChanger
Default Type (all values not listed above)	Other	GenSbp2Device

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Identifiers for USB Devices

The set of identifiers generated for USB devices depends on whether the device is a single-interface device or a multiple-interface device.

Single-Interface USB Devices

When a new USB device is plugged in, Microsoft® Windows® USBHUB driver composes the following device ID using information extracted from the device's device descriptor:

USB\VID_*v(4)*&PID_*d(4)*&REV_*r(4)*

where *v(4)* is the 4-digit hexadecimal vendor code that the USB committee assigns to the vendor, *d(4)* is the 4-digit hexadecimal product code that the vendor assigns to the device, and *r(4)* is the revision code. The hub driver extracts the vendor and product codes from the *idVendor* and *idProduct* fields of the device descriptor, respectively.

An INF model section can also specify the following hardware ID:

USB\VID_*v(4)*&PID_*d(4)*

and the following compatible IDs:

USB\CLASS_*c(2)*&SUBCLASS_*s(2)*&PROT_*p(2)*

USB\CLASS_*c(2)*&SUBCLASS_*s(2)*

USB\CLASS_*c(2)*

USB\COMPOSITE

where *c(2)* is the class code, *s(2)* is the subclass code, and *p(2)* is the protocol code. The class code, subclass code, and protocol code are determined by the *bDeviceClass, bDeviceSubClass,* and *bDeviceProtocol* fields of the device descriptor, respectively. These values are in 2-digit hexadecimal format.

Multiple-Interface USB Devices

Devices with multiple interfaces are called *composite* devices. When a new USB composite device is plugged into a Windows 2000 or later machine, the USB hub driver creates a PDO and notifies the operating system that its set of child devices has changed. After querying the hub driver for the hardware identifiers associated with the new PDO, the system searches the appropriate INF files to find a match for the identifiers. If it finds a match other than *USB\COMPOSITE*, it loads the driver indicated in the INF file. If, however, no other match is found, the operating system makes use of the compatible ID *USB\COMPOSITE*, for which it loads the USB Generic Parent driver. The Generic Parent driver then creates a separate PDO and generates a separate set of hardware identifiers for each interface of the composite device.

Each interface has a device ID of the following form:

USB\ VID_*v(4)*&PID_*d(4)*&MI_*z(2)*

where *v(4)* is the 4-digit hexadecimal vendor code that the USB committee assigns to the vendor, *d(4)* is the 4-digit hexadecimal product code that the vendor assigns to the device, and *z(2)* is the interface number, extracted from the *bInterfaceNumber* field of the interface descriptor.

An INF model section can also specify the following compatible IDs:

USB\CLASS_*c(2)*&SUBCLASS_*s(2)*&PROT_*p(2)*

USB\CLASS_*c(2)*&SUBCLASS_*s(2)*

USB\CLASS_*c(2)*

USB\COMPOSITE

where *c(2)* is the class code, *s(2)* is the subclass code, and *p(2)* is the protocol code. The class code, subclass code, and protocol code are determined by the *bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol* fields of the interface descriptor, respectively. These values are in 2-digit hexadecimal format.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

Device Classes

This section provides information about two device classes used to place devices and drivers into groups whose members have similar characteristics:

Device Setup Classes
  Used to group together devices that are installed and configured in a similar manner.
Device Interface Classes
  Used to group together devices that provide similar capabilities.

Because of similarities in the way both types of device classes are specified and used, the section provides a topic entitled Setup Classes Versus Interface Classes.

Setup functions use device information sets to expose the members of a class to user-mode code.

Built on Thursday, July 19, 2001

Send feedback to MSDN.   Look here for MSDN Online resources.

Device Installation: Windows DDK

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 the class installer and class co-installers 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 doesn't need to define a new setup class because cameras fall under the Image setup class. Similarly, uninterruptible power supply (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 section provides the following topics:

System-Supplied Device Setup Classes

Creating a New Device Setup Class

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

System-Supplied Device Setup Classes

This section lists the system-supplied device setup classes. In the definition for each 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. The GUIDs for system-supplied device setup classes are defined in *devguid.h*.

Supplying the appropriate class GUID value in the INF for a device, rather than or in addition to the **Class=***class-name* entry, significantly improves the performance of system INF searching. 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 **ClassGuid=**{00000000-0000-0000-0000-000000000000} in their **Version** sections to cut down on the system's INF searching time.

Following are two lists of system-defined device setup classes:

System-defined device setup classes available to vendors

System-defined device setup classes reserved for system use

System-Defined Device Setup Classes Available to Vendors

Battery Devices
    Class = Battery
    ClassGuid = {72631e54-78a4-11d0-bcf7-00aa00b7b32a}

This class includes drivers of battery devices and UPSs.

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 Plug and Play 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.)

IEEE 1284.4 Devices
Class=Dot4
ClassGuid = {48721b56-6795-11d2-b1a8-0080c72e74a2}

This class includes drivers that control the operation of multifunction IEEE 1284.4 peripheral devices.

IEEE 1284.4 Print Functions
Class = Dot4Print
ClassGuid = {49ce6ac8-6f86-11d2-ble5-0080c72e74a2}

This class includes Dot4 print functions. A Dot4 print function is a function on a Dot4 device and has a single child device, which is a member of the Printer device setup class.

IEEE 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.

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.

Media 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.

Modem
Class = Modem
ClassGuid = {4d36e96d-e325-11ce-bfc1-08002be10318}

This class includes 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 mouse devices and other kinds of pointing devices, such as trackballs. That is, this class 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 Plug and Play multifunction device is installed under this class and enumerates the modem and netcard separately as its child devices.

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.

Multiport 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 ports)
Class = Ports
ClassGuid = {4d36e978-e325-11ce-bfc1-08002be10318}

This class includes drivers of serial and parallel port devices. See also the MultiportSerial class.

Printers
Class = Printer
ClassGuid = {4d36e979-e325-11ce-bfc1-08002be10318}

This class includes printer drivers.

Printers, Bus-specific class drivers
Class = PNPPrinters
ClassGuid = {4658ee7e-f050-11d1-b6bd-00c04fa372a7}

This class includes bus-specific print class drivers, which provide printer communication for a specific bus.

Processors
Class = Processor
ClassGuid = {50127dc3-0f36-415e-a6cc-4cb3be910b65}

This class includes processor types.

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 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.

System-Defined Device Setup Classes Reserved for System Use

The following classes and GUIDs should not be used to install devices (or drivers) on Windows 2000 or later 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.

Bluetooth
Class = Bluetooth
ClassGUID = {e0cbf06c-cd8b-4647-bb8a-253b43f0f974}

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.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Creating a New Device Setup Class

You should only create a new device setup class if absolutely necessary. It is usually possible to assign your device to one of the system-supplied device setup classes.

If your device meets both of the following criteria, you should assign it to an existing device setup class:

- Your device's installation and configuration requirements match those of an existing class.
- Your device's capabilities match those of an existing class.

Under the either of following circumstances, you should consider providing a device co-installer:

- Your device has installation requirements that are not supported by an existing device type-specific INF file.
- Installation of your device requires device property pages that are not provided by an existing class.

If your device provides capabilities that are significantly different from the capabilities provided by devices that belong to existing classes, it might merit a new device setup class. However, you must never create a new setup class for a device that belongs to one of the system-supplied classes. If you do, you will bypass the system-supplied class installer and your device will not be properly integrated into the system.

If you think a new device setup class is needed, your new class should be based on new device capabilities, and not on the device's location. For instance, supporting an existing device on a new bus should not require a new setup class.

Before creating a new device setup class, contact Microsoft to find out if a new system-supplied device setup class is being planned for your device type

You can create a new device setup class using an INF file. Besides installing support for a device, an INF file can initialize a new device setup class for the device. Such an INF file has an INF **ClassInstall32** section.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Device Interface Classes

Device interface classes are the means by which drivers make devices available to applications and other drivers.

This section contains the following topics:

Introduction to Device Interfaces

Registering a Device Interface Class

Enabling and Disabling a Device Interface Instance

Using a Device Interface

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Introduction to Device Interfaces

Any driver of a physical, logical, or virtual device to which user-mode code can direct I/O requests must supply some sort of name for its user-mode clients. Using the name, a user-mode application (or other system component) identifies the device from which it is requesting I/O.

In Windows NT® 4.0 and earlier versions of the NT-based operating system, drivers named their device objects and then set up symbolic links in the registry between these names and a user-visible Win32® logical name.

For Windows® 2000 and later, drivers do not name device objects. Instead, they make use of *device interface classes*. A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, then enable an instance of the class for each device object to which user-mode I/O requests might be sent.

Each device interface class is associated with a GUID. The system defines GUIDs for common device interface classes in device-specific header files. Vendors can create additional device interface classes.

For example, three different types of mice could be members of the same device interface class, even if one connects through a USB port, a second through a serial port, and the third through an infrared port. Each driver registers its device as a member of the interface class GUID_DEVINTERFACE_MOUSE. This GUID is defined in the header file *ntddmou.h*.

Typically, drivers register for only one interface class. However, drivers for devices that have specialized functionality beyond that defined for their standard interface class might also register for an additional class. For example, a driver for a disk that can be mounted should register for both its disk interface class (GUID_DEVINTERFACE_DISK) and the mountable device class (MOUNTDEV_MOUNTED_DEVICE_GUID).

When a driver registers an instance of a device interface class, the I/O Manager associates the device and the device interface class GUID with a symbolic link name. The link name is stored in the registry and persists across system boots. An application that uses the interface can query for instances of the interface and receive a symbolic link name representing a device that supports the interface. The application can then use the symbolic link name as a target for I/O requests.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Registering a Device Interface Class

There are three ways to register a device interface class:

- Kernel-mode components, such as most drivers, can call I/O Manager routines. This topic describes how to use these routines.
- User-mode setup applications call call **SetupDi***Xxx* functions. See Device Interface Functions for information about these functions.
- An INF file can contain INF *DDInstall*.**Interfaces** sections.

A WDM driver does not name its device objects. Instead, when the driver calls **IoCreateDevice** to create a device object, it should specify a NULL string for the device name. Bus drivers should set the FILE_AUTOGENERATED_DEVICE_NAME flag. All PnP function, filter, and bus drivers should set the FILE_DEVICE_SECURE_OPEN flag. In response, the system chooses a unique device name for the PDO.

After creating the device object and attaching it to the device stack, one driver calls **IoRegisterDeviceInterface** to register a device interface class and to create an instance of the interface. Typically, the function driver makes this call from its *AddDevice* routine, but sometimes the filter driver registers the interface.

The routine returns a symbolic link name. A driver passes the link name when it enables or disables the device interface instance. Other system components cannot use a device interface instance until the driver has enabled it. See Enabling and Disabling a Device Interface Instance for details.

The driver also uses the symbolic link name to access the registry key, in which it can store information specific to the device interface. (See **IoOpenDeviceInterfaceRegistryKey** for more information.) Applications use the link name to open the device.

A driver can call **IoRegisterDeviceInterface** as many times as necessary to register instances of additional device interface classes.

Built on Thursday, July 19, 2001

Send feedback to MSDN. Look here for MSDN Online resources.

Device Installation: Windows DDK

Enabling and Disabling a Device Interface Instance

After successfully starting the device, the driver that registered the interface calls **IoSetDeviceInterfaceState** to enable an interface instance. The driver passes the symbolic link name returned by **IoRegisterDeviceInterface** along with the Boolean value TRUE to enable the interface instance.

If the driver can successfully start its device, it should call this routine while handling the Plug and Play (PnP) Manager's IRP_MN_START_DEVICE request.

After the IRP_MN_START_DEVICE request completes, the PnP Manager issues device interface arrival notifications to any kernel-mode or user-mode components that requested them. For more information, see Registering for Device Interface Change Notification.

To disable a device interface instance, a driver calls **IoSetDeviceInterfaceState**, passing the *SymbolicLinkName* returned by **IoRegisterDeviceInterface** and FALSE as the value of *Enable*.

A driver should disable a device's interfaces when it handles an IRP_MN_SURPRISE_REMOVAL or IRP_MN_REMOVE_DEVICE for the device. A driver should not disable the interfaces when the device is stopped (IRP_MN_STOP_DEVICE) or put in a sleep state; it should leave any device interfaces enabled and queue I/O instead.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Using a Device Interface

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code must can **SetupDi***Xxx* functions to find out about registered, enabled device interfaces. See Device Interface Functions for details.

Before a kernel-mode component can use a specific device or file object, it must do the following:

1. Determine whether the required device interface class is registered and enabled.

   A driver can register with the PnP Manager to be notified when an instance of a device interface is enabled or disabled. To register, the component calls **IoRegisterPlugPlayNotification**. This routine stores the address of a driver-supplied callback, which is called whenever an instance of a device interface instance is enabled or disabled, for a specified device class. The callback routines receives the DEVICE_INTERFACE_CHANGE_NOTIFICATION structure, which contains a Unicode string representing the interface instance's symbolic link. See Using PnP Device Interface Change Notification for more information.

   A driver or other kernel-mode component can also call **IoGetDeviceInterfaces** to get a list of all registered, enabled device interface instances for a specific device interface class. The returned list contains pointers to the Unicode symbolic link strings that identify the device interface instances.

2. Get a pointer to a device or file object corresponding to an instance of the interface.

   To access a specific device object, the driver must call **IoGetDeviceObjectPointer**, passing the Unicode string for the required interface in the *ObjectName* parameter. To access a file object, the driver must call **InitializeObjectAttributes**, passing the Unicode string in the *ObjectName* parameter, and then pass the successfully initialized attribute structure in a call to **ZwCreateFile**.

Built on Thursday, July 19, 2001

Device Installation: Windows DDK

Setup Classes Versus Interface Classes

It is important to distinguish between the two types of device classes: device *interface* classes and device *setup* classes. The two can be easily confused because in user-mode code the same set of device installation functions and the same set of data structures (device information sets) are used with both classes. Moreover, a device often belongs to both a setup class and several interface classes at the same time. Nevertheless, the two types of classes serve different purposes, make use of different areas in the registry, and rely on a different set of header files for defining class GUIDs.

Device setup classes provide a mechanism for grouping devices that are installed and configured in the same way. A setup class identifies the class installer and class co-installers that are involved in installing the devices that belong to the class. For example, all CD-ROM drives belong to the CDROM setup class and will use the same co-installer when installed.

Device interface classes provide a mechanism for grouping devices according to shared characteristics. Rather than tracking the presence in the system of an individual device, drivers and user applications can register to be notified of the arrival or removal of any device that belongs to a particular interface class.

Setup classes are defined in the system file *devguid.h*. This file defines a series of GUIDs for setup classes. However, the device setup classes represented in *devguid.h* must not be confused with device *interface* classes. The *devguid.h* file only contains GUIDs for setup classes.

Definitions of interface classes are not provided in a single file. A device interface class is always defined in a header file that belongs exclusively to a particular class of devices. For instance, *ntddmou.h* contains the definition of GUID_CLASS_MOUSE, the GUID representing the mouse interface class; *ntddpar.h* defines the interface class GUID for parallel devices; *ntddpcm.h* defines the standard interface class GUID for PCMCIA devices; *ntddstor.h* defines the interface class GUID for storage devices, and so on.

The GUIDs in header files that are specific to the device interface class should be used to register for notification of arrival of an instance of a device interface. If a driver registers for notification using a setup class GUID instead of an interface class GUID, then it will not be notified when an interface arrives.

When defining a new setup class or interface class, *do not use a single GUID to identify both a setup class and an interface class.*

For additional information about GUIDs, see Using GUIDs in Drivers.

Built on Thursday, July 19, 2001

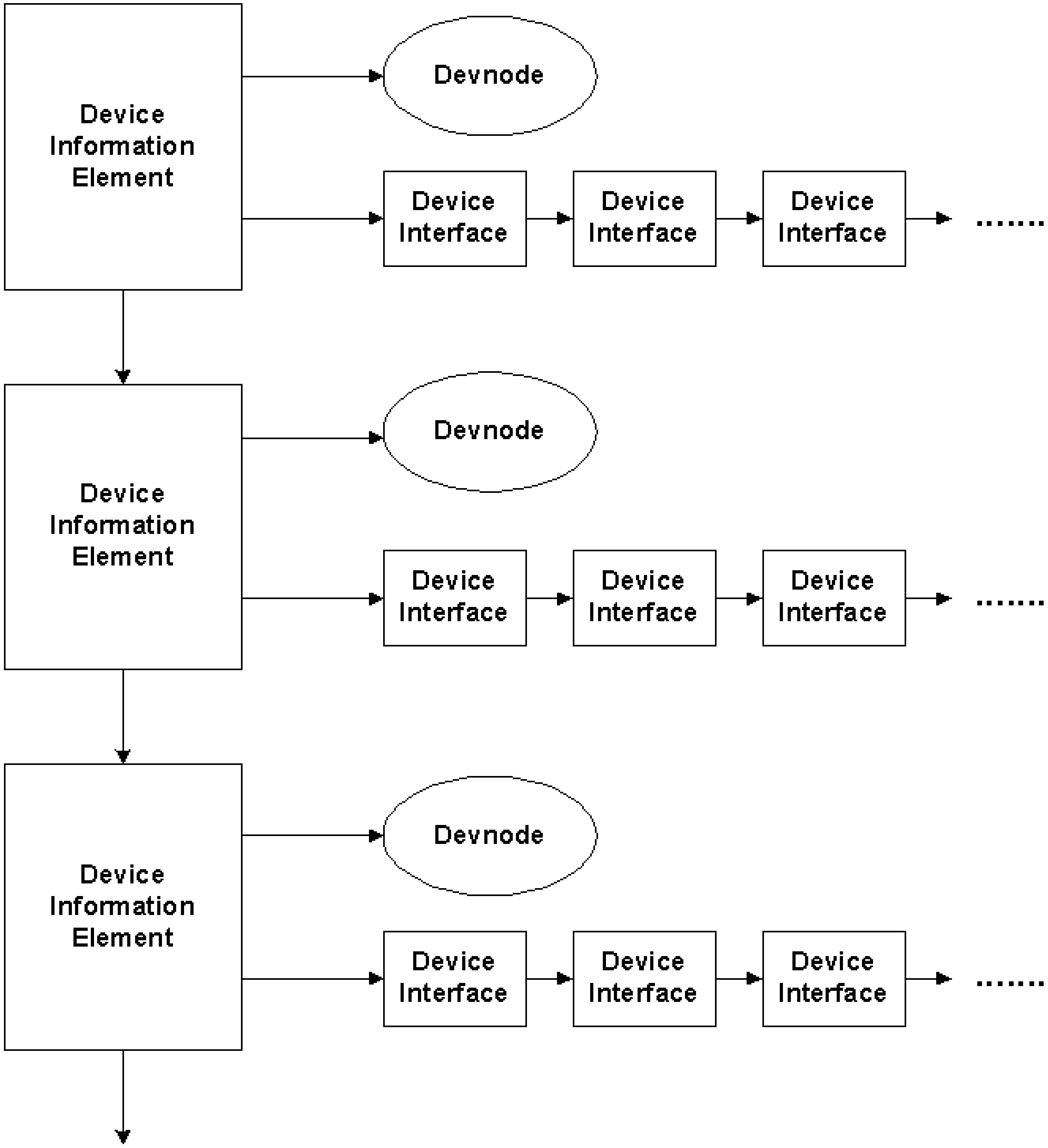
Device Installation: Windows DDK

Device Information Sets

In user mode, devices that belong to either device setup classes or device interface classes are managed using *device information elements* and *device information sets.* A device information set consists of device information elements for all the devices belonging to some device setup class or device interface class.

Each device information element contains a handle to the device's devnode, and a pointer to a linked list of all the device interfaces associated with the device described by that element. If a device information set describes members of a setup class, the element might not point to any device interfaces, since setup class members are not necessarily associated with an interface.

The following diagram depicts the internal structure of a device information set.



Device Information Set

Creating a Device Information Set

After creating a device information set with **SetupDiCreateDeviceInfoList**, device information elements can be created and added to the list one at a time using **SetupDiCreateDeviceInfo**. Alternatively, **SetupDiGetClassDevs** can be called to create a device information set composed of all devices associated with a specified device setup class or device interface class.

Enumerating Device Information

Once a device information set has been created, both the devices and the device interfaces that belong to the set can be enumerated, but different operations are required for each type of enumeration. **SetupDiEnumDeviceInfo** enumerates all devices that belong to the information set that meet certain criteria. Each call to **SetupDiEnumDeviceInfo** extracts a SP_DEVINFO_DATA structure that roughly corresponds to a device information element. SP_DEVINFO_DATA contains the GUID of the class that the device belongs to and a *device instance* handle that points to the devnode for the device. The principal difference between an SP_DEVINFO_DATA structure and a complete device element is that SP_DEVINFO_DATA does *not* contain the linked list of interfaces associated with the device. Therefore, **SetupDiEnumDeviceInfo** can not be used to enumerate the interfaces in the device information set.

To enumerate the device interfaces in a device information set, call **SetupDiEnumDeviceInterfaces**. This routine steps through all the device information elements in the device information set, extracts the interfaces in the interface list of each element, and returns one interface with each call. If **SetupDiEnumDeviceInterfaces** is passed an SP_DEVINFO_DATA structure as input in its second parameter, it constrains the enumeration to only those interfaces that are associated with the device indicated by SP_DEVINFO_DATA.

SetupDiEnumDeviceInterfaces returns an SP_DEVICE_INTERFACE_DATA structure. SP_DEVICE_INTERFACE_DATA contains the interface class GUID and other information about the interface, including a reserved field with encoded information that can be used to obtain the name of the interface. To get the interface name, one further step is required: **SetupDiGetDeviceInterfaceDetail** must be called. **SetupDiGetDeviceInterfaceDetail** returns a structure of type SP_DEVICE_INTERFACE_DETAIL_DATA that contains the path in the system object tree that defines the interface.

Built on Thursday, July 19, 2001