

# Contents

[Device and Driver Installation](#)

[Roadmap for Device and Driver Installation](#)

[Overview](#)

[Overview of Device and Driver Installation](#)

[How Windows Installs Devices](#)

[Step 1: The New Device is Identified](#)

[Step 2: A Driver for the Device is Selected](#)

[Step 3: The Driver for the Device is Installed](#)

[Components](#)

[System-provided Device Installation Components](#)

[Plug and Play Manager](#)

[Driver Store](#)

[Vendor-provided Device Installation Components](#)

[Installation Files](#)

[Device Installation Files](#)

[Sample Device Installation Files](#)

[Device Installation Types](#)

[How Windows Selects Drivers](#)

[Overview of the Driver Selection Process](#)

[How Windows selects a driver for a device](#)

[How Windows Ranks Drivers](#)

[How Windows Ranks Drivers](#)

[Signature Categories and Driver Installation](#)

[Signature Score](#)

[LowerLogoVersion](#)

[Feature Score](#)

[Identifier Score](#)

[Driver Rank Ranges](#)

[Driver Rank Example](#)

[Driver Rank Information in the SetupAPI Log](#)

[AllSignedEqual Group Policy](#)

## [Concepts](#)

[Driver Packages](#)

[Overview of Driver Packages](#)

[Components of a Driver Package](#)

[Installation Component Overview](#)

[Supplying an INF File](#)

[INF Files](#)

[Overview of INF Files](#)

[Looking at an INF File](#)

[Using a Universal INF File](#)

[Using an Extension INF File](#)

[Using an Extension INF File Template](#)

[Updating Device Firmware using Windows Update](#)

[Summary of INF Sections](#)

[Summary of INF Directives](#)

[General Guidelines for INF Files](#)

[General Syntax Rules for INF Files](#)

[Specifying the Source and Target Locations for Device Files](#)

[Source Media for INF Files](#)

[Target Media for INF Files](#)

[Copying INF Files](#)

[Creating INF Files for Multiple Platforms and Operating Systems](#)

[INF File Platform Extensions and x64-Based Systems](#)

[INF File Platform Extensions and x86-Based Systems](#)

[Cross-Platform INF Files](#)

[Combining Platform Extensions with Other Section Name Extensions](#)

[Combining Platform Extensions with Operating System Versions](#)

[Sample INF Models Sections for One or More Target Operating Systems](#)

[Sample INF Models Sections for Only One Target Operating System](#)

[Sample INF File for Device Installation on Multiple Versions of Windows](#)

[Creating International INF Files](#)

[Specifying Driver Load Order](#)

[Using Dirids](#)

[Accessing INF Files from a Device Installation Application](#)

[Opening and Closing an INF File](#)

[Retrieving Information from an INF File](#)

[Providing Icons for a Device](#)

[Driver Signing](#)

[Driver Signing](#)

[Digital Signatures](#)

[Authenticode Digital Signatures](#)

[Catalog Files and Digital Signatures](#)

[Embedded Signatures in a Driver File](#)

[Digital Signatures and PnP Device Installation](#)

[Digital Certificates](#)

[Certificate Stores](#)

[Local Machine and Current User Certificate Stores](#)

[Trusted Root Certification Authorities Certificate Store](#)

[Trusted Publishers Certificate Store](#)

[Kernel-Mode Code Signing Policy](#)

[Windows 10 in S mode Driver Requirements](#)

[Managing the Signing Process](#)

[Kernel-Mode Code Signing Requirements](#)

[PnP Device Installation Signing Requirements](#)

[Authenticode Signing of Third-party CSPs](#)

[Managing the Digital Signature or Code Signing Keys](#)

[Cross-Certificates for Kernel Mode Code Signing](#)

[Introduction to Test-Signing](#)

[How to Test-Sign a Driver Package](#)

[How to Install a Test-signed Driver Required for Windows Setup and Boot](#)

[Creating Test Certificates](#)

[Installing Test Certificates](#)

[Viewing Test Certificates](#)

[Creating a Catalog File for Test-Signing a Driver Package](#)

[Using Inf2Cat to Create a Catalog File](#)

[Using MakeCat to Create a Catalog File](#)

[Test-Signing a Driver Package's Catalog File](#)

[Test-Signing a Driver through an Embedded Signature](#)

[Configuring the Test Computer to Support Test-Signing](#)

[Enable Loading of Test Signed Drivers](#)

[Enabling Code Integrity Event Logging and System Auditing](#)

[Verifying the Test Signature](#)

[Installing a Test-Signed Driver Package on the Test Computer](#)

[Using the DevCon Tool to Install a Driver Package](#)

[WHQL Test Signature Program](#)

[MakeCert Test Certificate](#)

[Commercial Test Certificate](#)

[Enterprise CA Test Certificate](#)

[Test-Signing Driver Packages](#)

[Test-Signing a Driver File](#)

[Verifying the Signature of a Test-Signed Driver File](#)

[Creating a Catalog File for a Test-Signed Driver Package](#)

[Test-Signing a Catalog File](#)

[Verifying the Signature of a Test-Signed Catalog File](#)

[Installing Test-Signed Driver Packages](#)

[Installing a Test Certificate on a Test Computer](#)

[Using CertMgr to Install Test Certificates on a Test Computer](#)

[Deploying a Test Certificate by Using the Default Domain Policy](#)

[Installing a Test-Signed Catalog File for a Non-PnP Driver](#)

[Installing a Catalog File by using CryptCATAdminAddCatalog](#)

[Installing a Catalog File by using SignTool](#)

[Installing an Unsigned Driver during Development and Test](#)

[Signing Drivers for Public Release](#)

[Introduction to Release-Signing](#)

How to Release-Sign a Driver Package  
Obtaining a Software Publisher Certificate (SPC)  
Personal Information Exchange (.pfx) Files  
Importing an SPC into a Certificate Store  
Determining an SPC's Cross-Certificate  
Creating a Catalog File for Release-Signing a Driver Package  
Release-Signing a Driver Package's Catalog File  
Release-Signing a Driver through an Embedded Signature  
Verifying the Release-Signature  
Configuring a Computer to Support Release-Signing  
Installing a Release-Signed Driver Package  
WHQL Release Signature  
Release Certificates  
Software Publisher Certificate  
Commercial Release Certificate  
Deprecation of Software Publisher Certificates, Commercial Release Certificates, and Commercial Test Certificates  
Release-Signing Driver Packages  
Release-Signing a Driver File  
Verifying the Signature of a Release-Signed Driver File  
Creating a Catalog File for a PnP Driver Package  
Signing a Catalog File with an SPC  
Verifying the SPC Signature of a Catalog File  
Signing a Catalog File with a Commercial Release Certificate  
Verifying the Signature of a Catalog File Signed by a Commercial Release Certificate  
Creating a Catalog File for a Non-PnP Driver Package  
Installing a Release-Signed Catalog File for a Non-PnP Driver  
Detecting Driver Load Errors  
Code Integrity Diagnostic System Log Events  
Viewing Code Integrity Events  
Enabling the System Event Audit Log  
Code Integrity Event Log Messages  
SetupAPI Device Installation Log Entries

[Windows Driver Signing Tutorial](#)

[Test Signing](#)

[Release Signing](#)

[Troubleshooting Driver Signing Installation](#)

[Appendix 1: Enforcing Kernel-Mode Signature Verification in Kernel Debugging Mode](#)

[Appendix 2: Signing Drivers with Visual Studio](#)

[Appendix 3: Enable Code Integrity Event Logging and System Auditing](#)

[Appendix 4: Driver Signing Issues](#)

[Device Metadata Packages](#)

[Overview of Device Metadata Packages](#)

[Windows Metadata and Internet Services](#)

[Device Metadata Retrieval Client](#)

[Device Metadata Store](#)

[Device Metadata Cache](#)

[Device Metadata Package Structure](#)

[PackageInfo XML Document](#)

[DeviceInfo XML Document](#)

[Device Icon File](#)

[WindowsInfo XML Document](#)

[SoftwareInfo XML Document](#)

[Building Device Metadata Packages](#)

[Installing Device Metadata Packages to an Offline Windows Image](#)

[Installing Device Metadata Packages from WMIS](#)

[Installing Device Metadata Packages through an Application](#)

[Installing Device Metadata Packages through a Driver Package](#)

[Manually Adding Device Metadata Packages](#)

[How the DMRC Selects a Device Metadata Package](#)

[How the DMRC Determines When to Search the WMIS Server](#)

[Debugging Device Metadata Packages By Using Event Viewer](#)

[Debugging Device Metadata Packages by Using Problem Reports](#)

[Device Metadata Error Codes](#)

[Best Practices for Specifying Hardware IDs](#)

Specifying Hardware IDs for a Bluetooth Device  
Specifying Hardware IDs for a Computer  
Specifying Hardware IDs for a Multifunction Device  
Best Practices for Specifying Model IDs  
Best Practices for Testing the Download of Device Metadata Packages

Device and Driver Installation Software

SetupAPI

Using General Setup Functions  
INF File Processing Functions  
Disk Prompting and Error Handling Functions  
File Queuing Functions  
Default Queue Callback Routine Functions  
Cabinet File Function  
Disk-Space List Functions  
MRU Source List Functions  
File Log Functions  
User Interface Functions  
SetupAPI Logging Functions  
Using Device Installation Functions  
Basic Installation Operations  
Functions that Simplify Driver Installation  
Using SetupAPI To Verify Driver Authenticode Signatures  
Determining the Parent of a Device  
Obtaining the Parent of a Device in the Device Tree  
Determining the Parent of a Nonpresent Device  
Obtaining the Original Source Path of an Installed INF File  
Guidelines for Using SetupAPI  
DIFx Guidelines  
Device and Driver Installation Software Components  
Writing a Co-installer  
Co-installer Operation  
Co-installer Interface

Co-installer Functionality

Handling DIF Codes

Registering a Co-installer

Registering a Device-Specific Co-installer

Registering a Class Co-installer

Calling the Default DIF Code Handlers

Finish-Install Actions

Overview of Finish-Install Actions

How Finish-Install Actions are Processed

Marking a Device as having a Finish Install Action to Perform

Running Finish-Install Actions

Running the Default Finish-Install Action

Implementing Finish-Install Actions

Guidelines for Implementing Finish-Install Actions

Code Example: Finish-Install Actions in a Class Installer

Code Example: Finish-Install Actions in a Co-installer

Overview of Device Property Pages

How Device Property Pages are Displayed

Types of Device Property Page Providers

General Requirements for Device Property Page Providers

Creating Custom Property Pages

Property Page Callback Function

Handling Windows Messages for Property Pages

Sample Custom Property Page

Specific Requirements for Device Property Page Providers (Co-Installers)

Specific Requirements for Device Property Page Providers (Property Page Extension DLLs)

Invoking a Device Properties Dialog Box

Invoking a Device Properties Dialog Box Programmatically in an Installation Application

Invoking a Device Properties Dialog Box from a Command-line Prompt

DeviceProperties\_RunDLL Function Prototype

Writing a Device Installation Application

## [Guidelines for Writing Device Installation Applications](#)

### [Installing Device-Specific Applications](#)

#### [Device Installation Application Not Included in the Driver Package](#)

#### [Device Installation Application that is Included in the Driver Package](#)

#### [Device Installation Application Started through AutoRun](#)

### [Guidelines for Starting Device Installation Applications through Co-installers](#)

#### [Preloading Driver Packages](#)

#### [Preinstalling Driver Packages](#)

#### [Hardware-First Installation](#)

#### [Installing an In-box Driver or a Preinstalled Driver](#)

#### [Software-First Installation](#)

#### [Checking for In-Progress Installations](#)

#### [Determining Whether a Device Is Plugged In](#)

#### [Uninstalling Devices and Driver Packages](#)

#### [How Devices and Driver Packages are Uninstalled](#)

#### [Using Device Manager to Uninstall Devices and Driver Packages](#)

#### [Using SetupAPI to Uninstall Devices and Driver Packages](#)

### [Updating Driver Files](#)

#### [Creating Secure Device Installations](#)

#### [Using a Component INF File](#)

#### [Pairing a driver with a Universal Windows Platform \(UWP\) app](#)

### [Device Identification](#)

#### [Device Identification Strings](#)

#### [Plug and Play ID - PNPID Request](#)

#### [Device ID](#)

#### [Hardware ID](#)

#### [Compatible ID](#)

#### [Instance ID](#)

#### [Device Instance ID](#)

#### [Container ID](#)

#### [Overview of Container IDs](#)

#### [How Container IDs are Generated](#)

[Container IDs Generated from a Bus-Specific Unique ID](#)

[How USB Devices are Assigned Container IDs](#)

[Using ACPI to Configure USB Ports on a Computer](#)

[Best Practices for Supporting Container IDs on USB Devices](#)

[Using Microsoft OS ContainerID Descriptors](#)

[Using the USB Removable Capability for Device Container Grouping](#)

[Avoiding Device Container Conflicts](#)

[Container IDs for Bluetooth Devices](#)

[Container IDs for PnP-X Devices](#)

[Container IDs for DPWS Devices](#)

[Container IDs for UPnP Devices](#)

[Container IDs for 1394 Devices](#)

[Container IDs for eSATA Devices](#)

[Container IDs for PCI Express Devices](#)

[Container IDs Generated from the Removable Device Capability](#)

[Overview of the Removable Device Capability](#)

[How Container IDs are Generated from the Removable Device Capability](#)

[Container IDs Generated from a Removable Device Capability Override](#)

[Using ACPI for Device Container Grouping](#)

[Verifying the Implementation of Container IDs](#)

[Troubleshooting the Implementation of Container IDs](#)

[Generic Identifiers](#)

[Identifiers for PCI Devices](#)

[Identifiers for SCSI Devices](#)

[Identifiers for IDE Devices](#)

[Identifiers for PCMCIA Devices](#)

[Identifiers for ISAPNP Devices](#)

[Identifiers for 1394 Devices](#)

[Identifiers for Secure Digital \(SD\) Devices](#)

[Identifiers for USB Devices](#)

[Standard USB Identifiers](#)

[Identifiers Generated by USBSTOR.SYS](#)

# Identifiers Generated by USBPRINT.SYS

## Device Classes

[Overview of Device Setup Classes](#)

[Creating a New Device Setup Class](#)

[Overview of Device Interface Classes](#)

[Registering a Device Interface Class](#)

[Enabling and Disabling a Device Interface Instance](#)

[Using a Device Interface](#)

[Registering for Notification of Device Interface Arrival and Device Removal](#)

[Setup Classes Versus Interface Classes](#)

[Device Information Sets](#)

## Device Properties

[Overview of Device Properties](#)

[Unified Model \(Windows Vista and later\)](#)

[Unified Device Property Model](#)

[System-Defined Device Properties](#)

[Creating Custom Device Properties](#)

[Property Keys](#)

[Property-Data-Type Identifiers](#)

[Property Value Requirements](#)

[Properties and Related System-Defined Items](#)

[INF File Entry Values That Modify Device Properties](#)

## SetupAPI

[Using SetupAPI to Access Device Properties](#)

[Accessing Device Instance Properties](#)

[Determining Which Properties Are Set for a Device Instance](#)

[Retrieving A Device Instance Property Value](#)

[Setting a Device Instance Property Value](#)

[Accessing Device Class Properties](#)

[Determining Which Properties Are Set for a Device Class](#)

[Retrieving a Device Class Property Value](#)

[Setting a Device Class Property Value](#)

[Accessing Device Interface Properties](#)

[Determining Which Properties are Set for a Device Interface](#)

[Retrieving a Device Interface Property Value](#)

[Setting a Device Interface Property Value](#)

[Using the INF AddProperty Directive and the INF DelProperty Directive](#)

[Device Property Model \(Windows Server 2003, Windows XP, and Windows 2000\)](#)

[Device Property Representations \(Windows Server 2003, Windows XP, and Windows 2000\)](#)

[INF File Entry Values That Modify Device Properties](#)

[Using SetupAPI and Configuration Manager to Access Device Properties](#)

[Accessing Device Instance SPDRP\\_Xxx Properties](#)

[Retrieving a Device Instance Identifier](#)

[Accessing Device Driver Properties](#)

[Retrieving the Status and Problem Code for a Device Instance](#)

[Retrieving Device Relations](#)

[Accessing Device Setup Class Properties](#)

[Retrieving SPCRP\\_Xxx Properties](#)

[Setting SPCRP\\_Xxx Properties](#)

[Accessing the Friendly Name and Class Name of a Device Setup Class](#)

[Accessing Icon Properties of a Device Setup Class](#)

[Accessing Registry Entry Values Under the Class Registry Key](#)

[Accessing the Co-installers Registry Entry Value of a Device Setup Class](#)

[Accessing Device Interface Class Properties](#)

[Accessing Device Interface Properties](#)

[Accessing Custom Device Properties](#)

[System Restarts](#)

[Device Installations and System Restarts](#)

[Avoiding System Restarts during Device Installations](#)

[Initiating System Restarts During Device Installations](#)

[Device Installations on 64-Bit Systems](#)

[Registry](#)

[Registry Trees and Keys](#)

[Registry Trees for Devices and Drivers](#)

[HKLM\SYSTEM\CurrentControlSet\Services Registry Tree](#)  
[HKLM\SYSTEM\CurrentControlSet\Control Registry Tree](#)  
[HKLM\SYSTEM\CurrentControlSet\Enum Registry Tree](#)  
[HKLM\SYSTEM\CurrentControlSet\HardwareProfiles Registry Tree](#)  
[RunOnce Registry Key](#)  
[DeviceOverrides Registry Key](#)  
[HardwareID Registry Subkey](#)  
[CompatibleID Registry Subkey](#)  
[LocationPaths Registry Subkey](#)  
[ChildLocationPaths Registry Subkey](#)  
[LocationPath Registry Subkey](#)  
[\\* Registry Subkey](#)

## INF Sections

[INF ClassInstall32 Section](#)  
[INF ClassInstall32.Services Section](#)  
[INF ControlFlags Section](#)  
[INF DDInstall Section](#)  
[INF DDInstall.Coinstallers Section](#)  
[INF DDInstall.Components Section](#)  
[INF DDInstall.Events Section](#)  
[INF DDInstall.FactDef Section](#)  
[INF DDInstall.HW Section](#)  
[INF DDInstall.Interfaces Section](#)  
[INF DDInstall.LogConfigOverride Section](#)  
[INF DDInstall.Services Section](#)  
[INF DDInstall.Software Section](#)  
[INF DDInstall.WMI Section](#)  
[INF DefaultInstall Section](#)  
[INF DefaultInstall.Services Section](#)  
[INF DestinationDirs Section](#)  
[INF InterfaceInstall32 Section](#)  
[INF Manufacturer Section](#)

[INF Models Section](#)

[INF SignatureAttributes Section](#)

[INF SourceDisksFiles Section](#)

[INF SourceDisksNames Section](#)

[INF Strings Section](#)

[INF Version Section](#)

[INF Directives](#)

[INF AddComponent Directive](#)

[INF AddEventProvider Directive](#)

[INF AddInterface Directive](#)

[INF AddPowerSetting Directive](#)

[INF AddProperty Directive](#)

[INF AddReg Directive](#)

[INF AddService Directive](#)

[INF AddSoftware Directive](#)

[INF BitReg Directive](#)

[INF CopyFiles Directive](#)

[INF CopyINF Directive](#)

[INF DelFiles Directive](#)

[INF DelProperty Directive](#)

[INF DelReg Directive](#)

[INF DelService Directive](#)

[INF DriverVer Directive](#)

[INF FeatureScore Directive](#)

[INF HardwareId Directive](#)

[INF Ini2Reg Directive](#)

[INF LogConfig Directive](#)

[INF ProfileItems Directive](#)

[INF Reboot Directive](#)

[INF RegisterDlls Directive](#)

[INF RenFiles Directive](#)

[INF UnregisterDlls Directive](#)

[INF UpdateIniFields Directive](#)

[INF UpdateInis Directive](#)

## Best Practices

[Accessing Device Properties](#)

[Rules for Modifying Device Properties](#)

[Accessing and Modifying Files](#)

[Accessing Registry Keys Safely](#)

[Opening a Device's Hardware Key](#)

[Opening a Device's Software Key](#)

[Modifying Registry Values in a Device's Software Key](#)

[Enumerating Installed Device Setup Classes](#)

[Opening Registry Keys for a Device Setup Class](#)

[Opening Software Keys for All Devices in a Setup Class](#)

[Enumerating Installed Device Interfaces](#)

[Accessing the Properties of Installed Device Interfaces](#)

[Modifying Registry Keys by Class Installers and Co-installers](#)

[Modifying Registry Values by Class Installers and Co-installers](#)

[Deleting the Registry Keys of a Device](#)

[Calling SetupAPI Functions](#)

[Enumerating Installed Devices](#)

[General Guidelines for Device and Driver Installation](#)

[Porting code from SetupApi to CfgMgr32](#)

[Writing Class Installers and Co-Installers](#)

[Writing INF Files](#)

[Installing a Boot-Start Driver](#)

[Installing a Filter Driver](#)

[Installing a Null Driver](#)

[Installing a New Bus Driver](#)

[Using Custom Hardware IDs and Compatible IDs](#)

[Installing a New Device Setup Class for a Bus](#)

[Installing Private Builds of In-box Drivers](#)

[Overview of Installing Private Builds of In-box Drivers](#)

Creating a Private Build of an In-box Driver  
Configuring Windows to Rank Driver Signatures Equally  
Installing the Updated Version of the Driver Package  
Writing Win32 Services to Interact with Devices

## Troubleshooting

Troubleshooting Device and Driver Installations  
Device Manager for makers of devices and drivers  
Device Manager Error Messages

Code 1 - CM\_PROB\_NOT\_CONFIGURED  
Code 3 - CM\_PROB\_OUT\_OF\_MEMORY  
Code 9 - CM\_PROB\_INVALID\_DATA  
Code 10 - CM\_PROB\_FAILED\_START  
Code 12 - CM\_PROB\_NORMAL\_CONFLICT  
Code 14 - CM\_PROB\_NEED\_RESTART  
Code 16 - CM\_PROB\_PARTIAL\_LOG\_CONF  
Code 18 - CM\_PROB\_REINSTALL  
Code 19 - CM\_PROB\_REGISTRY  
Code 21 - CM\_PROB\_WILL\_BE\_REMOVED  
Code 22 - CM\_PROB\_DISABLED  
Code 24 - CM\_PROB\_DEVICE\_NOT THERE  
Code 28 - CM\_PROB\_FAILED\_INSTALL  
Code 29 - CM\_PROB\_HARDWARE\_DISABLED  
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 - CM\\_PROB\\_REGISTRY\\_TOO\\_LARGE](#)

[Code 50 - CM\\_PROB\\_SETPROPERTIES\\_FAILED](#)

[Code 51 - CM\\_PROB\\_WAITING\\_ON\\_DEPENDENCY](#)

[Code 52 - CM\\_PROB\\_UNSIGNED\\_DRIVER](#)

[Code 53 - CM\\_PROB\\_USED\\_BY\\_DEBUGGER](#)

[Code 54 - CM\\_PROB\\_DEVICE\\_RESET](#)

[Code 55 - CM\\_PROB\\_CONSOLE\\_LOCKED](#)

[Code 56 - CM\\_PROB\\_NEED\\_CLASS\\_CONFIG](#)

[Code 57 - CM\\_PROB\\_GUEST\\_ASSIGNMENT\\_FAILED](#)

[Device Manager Details Tab](#)

[Viewing Hidden Devices](#)

[SetupAPI Logging](#)

[SetupAPI Text Logs](#)

[Format of a Text Log Header](#)

[Format of a Text Log Section](#)

[Format of a Text Log Section Header](#)

[Format of a Text Log Section Body](#)

[Format of a Text Log Section Footer](#)

[Format of Log Entries That Are Not Part of a Text Log Section](#)

[SetupAPI Logging Registry Settings](#)

[Setting the Event Level for a Text Log](#)

[Enabling Event Categories for a Text Log](#)

[Setting the Directory Path of the Text Logs](#)

[Using the SetupAPI Logging Functions](#)

[Writing Log Entries in a Text Log](#)

[Calling SetupWriteTextLog](#)

[Writing an Information Log Entry](#)

[Writing an Error or Warning Log Entry](#)

[Calling SetupWriteTextLogError](#)

[Calling SetupWriteTextLogInfLine](#)

[Writing Indented Log Entries](#)

[Log Tokens](#)

[Setting and Getting a Log Token for a Thread](#)

[SetupAPI Logging](#)

[SetupAPI Logging \(Windows Server 2003, Windows XP, and Windows 2000\)](#)

[Setting SetupAPI Logging Levels](#)

[Interpreting a Sample SetupAPI Log File](#)

[Debugging](#)

[Debugging Device Installations](#)

[Enabling Support for Debugging Device Installations](#)

[Debugging Device Installations with a User-mode Debugger](#)

[Debugging Device Installations with the Kernel Debugger \(KD\)](#)

[Early Launch AntiMalware](#)

[Overview of Early Launch AntiMalware](#)

[ELAM Prerequisites](#)

[ELAM Driver Requirements](#)

[ELAM Driver Submission](#)

[Reference](#)

[BUS1394\\_CLASS\\_GUID](#)

[CdChangerClassGuid](#)

[CdRomClassGuid](#)

[CM\\_Add\\_Range](#)

[CM\\_Create\\_DevNode](#)

[CM\\_Create\\_DevNode\\_Ex](#)

[CM\\_Create\\_Range\\_List](#)

[CM\\_Delete\\_Class\\_Key\\_Ex](#)

[CM\\_Delete\\_DevNode\\_Key\\_Ex](#)  
[CM\\_Delete\\_Range](#)  
[CM\\_Detect\\_Resource\\_Conflict](#)  
[CM\\_Detect\\_Resource\\_Conflict\\_Ex](#)  
[CM\\_Disable\\_DevNode\\_Ex](#)  
[CM\\_Dup\\_Range\\_List](#)  
[CM\\_Enable\\_DevNode\\_Ex](#)  
[CM\\_Find\\_Range](#)  
[CM\\_First\\_Range](#)  
[CM\\_Free\\_Range\\_List](#)  
[CM\\_Get\\_Class\\_Key\\_Name](#)  
[CM\\_Get\\_Class\\_Key\\_Name\\_Ex](#)  
[CM\\_Get\\_Class\\_Name](#)  
[CM\\_Get\\_Class\\_Name\\_Ex](#)  
[CM\\_Get\\_Device\\_Interface\\_Alias\\_Ex](#)  
[CM\\_Get\\_Device\\_Interface\\_List\\_Ex](#)  
[CM\\_Get\\_Device\\_Interface\\_List\\_Size\\_Ex](#)  
[CM\\_Get\\_DevNode\\_Custom\\_Property](#)  
[CM\\_Get\\_DevNode\\_Custom\\_Property\\_Ex](#)  
[CM\\_Get\\_DevNode\\_Registry\\_Property\\_Ex](#)  
[CM\\_Get\\_Global\\_State](#)  
[CM\\_Get\\_Global\\_State\\_Ex](#)  
[CM\\_Get\\_Hardware\\_Profile\\_Info](#)  
[CM\\_Get\\_Hardware\\_Profile\\_Info\\_Ex](#)  
[CM\\_Intersect\\_Range\\_List](#)  
[CM\\_Invert\\_Range\\_List](#)  
[CM\\_Merge\\_Range\\_List](#)  
[CM\\_Move\\_DevNode](#)  
[CM\\_Move\\_DevNode\\_Ex](#)  
[CM\\_Next\\_Range](#)  
[CM\\_Open\\_Class\\_Key\\_Ex](#)  
[CM\\_Open\\_DevNode\\_Key\\_Ex](#)

[CM\\_Query\\_Arbitrator\\_Free\\_Data](#)  
[CM\\_Query\\_Arbitrator\\_Free\\_Data\\_Ex](#)  
[CM\\_Query\\_Arbitrator\\_Free\\_Size](#)  
[CM\\_Query\\_Arbitrator\\_Free\\_Size\\_Ex](#)  
[CM\\_Query\\_Remove\\_SubTree](#)  
[CM\\_Query\\_Remove\\_SubTree\\_Ex](#)  
[CM\\_Register\\_Device\\_Driver](#)  
[CM\\_Register\\_Device\\_Driver\\_Ex](#)  
[CM\\_Register\\_Device\\_Interface](#)  
[CM\\_Register\\_Device\\_Interface\\_Ex](#)  
[CM\\_Remove\\_SubTree](#)  
[CM\\_Remove\\_SubTree\\_Ex](#)  
[CM\\_Run\\_Detection](#)  
[CM\\_Run\\_Detection\\_Ex](#)  
[CM\\_Set\\_DevNode\\_Registry\\_Property\\_Ex](#)  
[CM\\_Set\\_HW\\_Prof](#)  
[CM\\_Set\\_HW\\_Prof\\_Ex](#)  
[CM\\_Set\\_HW\\_Prof\\_Flags](#)  
[CM\\_Set\\_HW\\_Prof\\_Flags\\_Ex](#)  
[CM\\_Setup\\_DevNode\\_Ex](#)  
[CM\\_Test\\_Range\\_Available](#)  
[CM\\_Uninstall\\_DevNode\\_Ex](#)  
[CM\\_Unregister\\_Device\\_Interface](#)  
[CM\\_Unregister\\_Device\\_Interface\\_Ex](#)  
[CM\\_WaitNoPendingInstallEvents](#)  
[Config.DriverKey Section of a TxtSetup.oem File](#)  
[Defaults Section of a TxtSetup.oem File](#)  
[DEFINE\\_DEVPROPKEY](#)  
[deleteBinaries XML Element](#)  
[DEVPKEY\\_Device\\_Address](#)  
[DEVPKEY\\_Device\\_BaseContainerId](#)  
[DEVPKEY\\_Device\\_BusNumber](#)

DEVPKEY\_Device\_BusRelations  
DEVPKEY\_Device\_BusReportedDeviceDesc  
DEVPKEY\_Device\_BusTypeGuid  
DEVPKEY\_Device\_Capabilities  
DEVPKEY\_Device\_Characteristics  
DEVPKEY\_Device\_Children  
DEVPKEY\_Device\_Class  
DEVPKEY\_Device\_ClassGuid  
DEVPKEY\_Device\_CompatibleIds  
DEVPKEY\_Device\_ConfigFlags  
DEVPKEY\_Device\_ContainerId  
DEVPKEY\_Device\_DeviceDesc  
DEVPKEY\_Device\_DevNodeStatus  
DEVPKEY\_Device\_DevType  
DEVPKEY\_Device\_DHP\_Rebalance\_Policy  
DEVPKEY\_Device\_DmaRemappingPolicy  
DEVPKEY\_Device\_Driver  
DEVPKEY\_Device\_DriverCoInstallers  
DEVPKEY\_Device\_DriverDate  
DEVPKEY\_Device\_DriverDesc  
DEVPKEY\_Device\_DriverInfPath  
DEVPKEY\_Device\_DriverInfSection  
DEVPKEY\_Device\_DriverInfSectionExt  
DEVPKEY\_Device\_DriverLogoLevel  
DEVPKEY\_Device\_DriverPropPageProvider  
DEVPKEY\_Device\_DriverProvider  
DEVPKEY\_Device\_DriverRank  
DEVPKEY\_Device\_DriverVersion  
DEVPKEY\_Device\_EjectionRelations  
DEVPKEY\_Device\_EnumName  
DEVPKEY\_Device\_Exclusive  
DEVPKEY\_Device\_FirstInstallDate

DEVPKEY\_Device\_FriendlyName  
DEVPKEY\_Device\_GenericDriverInstalled  
DEVPKEY\_Device\_HardwareIds  
DEVPKEY\_Device\_InstallDate  
DEVPKEY\_Device\_InstallState  
DEVPKEY\_Device\_InstanceID  
DEVPKEY\_Device\_Legacy  
DEVPKEY\_Device\_LegacyBusType  
DEVPKEY\_Device\_LocationInfo  
DEVPKEY\_Device\_LocationPaths  
DEVPKEY\_Device\_LowerFilters  
DEVPKEY\_Device\_Manufacturer  
DEVPKEY\_Device\_MatchingDeviceID  
DEVPKEY\_Device\_ModelID  
DEVPKEY\_Device\_NoConnectSound  
DEVPKEY\_Device\_Parent  
DEVPKEY\_Device\_PDOName  
DEVPKEY\_Device\_PhysicalDeviceLocation  
DEVPKEY\_Device\_PowerData  
DEVPKEY\_Device\_PowerRelations  
DEVPKEY\_Device\_ProblemCode  
DEVPKEY\_Device\_ProblemStatus  
DEVPKEY\_Device\_RemovalPolicy  
DEVPKEY\_Device\_RemovalPolicyDefault  
DEVPKEY\_Device\_RemovalPolicyOverride  
DEVPKEY\_Device\_RemovalRelations  
DEVPKEY\_Device\_Reported  
DEVPKEY\_Device\_ResourcePickerExceptions  
DEVPKEY\_Device\_ResourcePickerTags  
DEVPKEY\_Device\_SafeRemovalRequired  
DEVPKEY\_Device\_SafeRemovalRequiredOverride  
DEVPKEY\_Device\_Security

DEVPKEY\_Device\_SecuritySDS  
DEVPKEY\_Device\_Service  
DEVPKEY\_Device\_SessionId  
DEVPKEY\_Device\_Siblings  
DEVPKEY\_Device\_UINumber  
DEVPKEY\_Device\_UINumberDescFormat  
DEVPKEY\_Device\_UpperFilters  
DEVPKEY\_DeviceClass\_Characteristics  
DEVPKEY\_DeviceClass\_ClassCoInstallers  
DEVPKEY\_DeviceClass\_ClassInstaller  
DEVPKEY\_DeviceClass\_ClassName  
DEVPKEY\_DeviceClass\_DefaultService  
DEVPKEY\_DeviceClass\_DevType  
DEVPKEY\_DeviceClass\_DHPRebalanceOptOut  
DEVPKEY\_DeviceClass\_Exclusive  
DEVPKEY\_DeviceClass\_Icon  
DEVPKEY\_DeviceClass\_IconPath  
DEVPKEY\_DeviceClass\_LowerFilters  
DEVPKEY\_DeviceClass\_Name  
DEVPKEY\_DeviceClass\_NoDisplayClass  
DEVPKEY\_DeviceClass\_NoInstallClass  
DEVPKEY\_DeviceClass\_NoUseClass  
DEVPKEY\_DeviceClass\_PropPageProvider  
DEVPKEY\_DeviceClass\_Security  
DEVPKEY\_DeviceClass\_SecuritySDS  
DEVPKEY\_DeviceClass\_SilentInstall  
DEVPKEY\_DeviceClass\_UpperFilters  
DEVPKEY\_DeviceDisplay\_Category  
DEVPKEY\_DeviceInterface\_ClassGuid  
DEVPKEY\_DeviceInterface\_DefaultInterface  
DEVPKEY\_DeviceInterface\_Enabled  
DEVPKEY\_DeviceInterface\_FriendlyName

DEVPKEY\_DeviceInterface\_Restricted  
DEVPKEY\_DrvPkg\_BrandingIcon  
DEVPKEY\_DrvPkg\_DetailedDescription  
DEVPKEY\_DrvPkg\_DocumentationLink  
DEVPKEY\_DrvPkg\_Icon  
DEVPKEY\_DrvPkg\_Model  
DEVPKEY\_DrvPkg\_VendorWebSite  
DEVPKEY\_NAME (Device Instance)  
DEVPKEY\_NAME (Device Interface)  
DEVPKEY\_NAME (Device Setup Class)  
DEVPKEY\_Numa\_Proximity\_Domain  
DEVPRIVATE\_DES  
DEVPRIVATE\_RANGE  
DEVPRIVATE\_RESOURCE  
DEVPROP\_MASK\_TYPE  
DEVPROP\_MASK\_TYPEMOD  
DEVPROP\_TYPE\_BINARY  
DEVPROP\_TYPE\_BOOLEAN  
DEVPROP\_TYPE\_BYTE  
DEVPROP\_TYPE\_CURRENCY  
DEVPROP\_TYPE\_DATE  
DEVPROP\_TYPE\_DECIMAL  
DEVPROP\_TYPE\_DEVPROPKEY  
DEVPROP\_TYPE\_DEVPROPTYPE  
DEVPROP\_TYPE\_DOUBLE  
DEVPROP\_TYPE\_EMPTY  
DEVPROP\_TYPE\_ERROR  
DEVPROP\_TYPE\_FILETIME  
DEVPROP\_TYPE\_FLOAT  
DEVPROP\_TYPE\_GUID  
DEVPROP\_TYPE\_INT16  
DEVPROP\_TYPE\_INT32

DEVPROP\_TYPE\_INT64  
DEVPROP\_TYPE\_NTSTATUS  
DEVPROP\_TYPE\_NULL  
DEVPROP\_TYPE\_SBYTE  
DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR  
DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING  
DEVPROP\_TYPE\_STRING  
DEVPROP\_TYPE\_STRING\_INDIRECT  
DEVPROP\_TYPE\_STRING\_LIST  
DEVPROP\_TYPE\_UINT16  
DEVPROP\_TYPE\_UINT32  
DEVPROP\_TYPE\_UINT64  
DEVPROP\_TYPEMOD\_ARRAY  
DEVPROP\_TYPEMOD\_LIST  
DEVPROPKEY  
DIF\_ADDPROPERTYPAGE\_ADVANCED  
DIF\_ADDPROPERTYPAGE\_BASIC  
DIF\_ALLOW\_INSTALL  
DIF\_ASSIGNRESOURCES  
DIF\_CALCDISKSPACE  
DIF\_DESTROYPRIVATEDATA  
DIF\_DESTROYWIZARDDATA  
DIF\_DETECT  
DIF\_DETECTCANCEL  
DIF\_DETECTVERIFY  
DIF\_ENABLECLASS  
DIF\_FINISHINSTALL\_ACTION  
DIF\_FIRSTTIMESETUP  
DIF\_FOUNDDEVICE  
DIF\_INSTALLCLASSDRIVERS  
DIF\_INSTALLDEVICE  
DIF\_INSTALLDEVICEFILES

DIF\_INSTALLINTERFACES  
DIF\_INSTALLWIZARD  
DIF\_MOVEDevice  
DIF\_NEWDEVICEWIZARD\_FINISHINSTALL  
DIF\_NEWDEVICEWIZARD\_POSTANALYZE  
DIF\_NEWDEVICEWIZARD\_PREANALYZE  
DIF\_NEWDEVICEWIZARD\_PRESELECT  
DIF\_NEWDEVICEWIZARD\_SELECT  
DIF\_POWERMESSAGEWAKE  
DIF\_PROPERTIES  
DIF\_PROPERTYCHANGE  
DIF\_REGISTER\_COINSTALLERS  
DIF\_REGISTERDEVICE  
DIF\_REMOVE  
DIF\_SELECTBESTCOMPATDRV  
DIF\_SELECTCLASSDRIVERS  
DIF\_SELECTDEVICE  
DIF\_TROUBLESHOOTER  
DIF\_UNREMOVE  
DIF\_UPDATEDRIVER\_UI  
DIF\_VALIDATECLASSDRIVERS  
DIF\_VALIDATEDRIVER  
DiskClassGuid  
Disks Section of a TxtSetup.oem File  
dpinst XML Element  
dpinstTitle XML Element  
enableNotListedLanguages XML Element  
eula XML Element  
eulaHeaderTitle XML Element  
eulaNoButton XML Element  
eulaYesButton XML Element  
Files.HwComponent.ID Section of a TxtSetup.oem File

finishText XML Element  
finishTitle XML Element  
FloppyClassGuid  
forceIfDriverIsNotBetter XML Element  
group XML Element  
GUID\_61883\_CLASS  
GUID\_AVC\_CLASS  
GUID\_BTHPORT\_DEVICE\_INTERFACE  
GUID\_CLASS\_COMPORT  
GUID\_CLASS\_INPUT  
GUID\_CLASS\_KEYBOARD  
GUID\_CLASS\_MODEM  
GUID\_CLASS\_MOUSE  
GUID\_CLASS\_USB\_DEVICE  
GUID\_CLASS\_USB\_HOST\_CONTROLLER  
GUID\_CLASS\_USBHUB  
GUID\_DEVICE\_APPLICATIONLAUNCH\_BUTTON  
GUID\_DEVICE\_BATTERY  
GUID\_DEVICE\_LID  
GUID\_DEVICE\_MEMORY  
GUID\_DEVICE\_MESSAGE\_INDICATOR  
GUID\_DEVICE\_PROCESSOR  
GUID\_DEVICE\_SYS\_BUTTON  
GUID\_DEVICE\_THERMAL\_ZONE  
GUID\_DEVINTERFACE\_BRIGHTNESS  
GUID\_DEVINTERFACE\_CDCHANGER  
GUID\_DEVINTERFACE\_CDROM  
GUID\_DEVINTERFACE\_COMPORT  
GUID\_DEVINTERFACE\_DISK  
GUID\_DEVINTERFACE\_DISPLAY\_ADAPTER  
GUID\_DEVINTERFACE\_FLOPPY  
GUID\_DEVINTERFACE\_HID

GUID\_DEVINTERFACE\_I2C  
GUID\_DEVINTERFACE\_IMAGE  
GUID\_DEVINTERFACE\_KEYBOARD  
GUID\_DEVINTERFACE\_MEDIUMCHANGER  
GUID\_DEVINTERFACE\_MODEM  
GUID\_DEVINTERFACE\_MONITOR  
GUID\_DEVINTERFACE\_MOUSE  
GUID\_DEVINTERFACE\_NET  
GUID\_DEVINTERFACE\_OPM  
GUID\_DEVINTERFACE\_PARALLEL  
GUID\_DEVINTERFACE\_PARCLASS  
GUID\_DEVINTERFACE\_PARTITION  
GUID\_DEVINTERFACE\_SERENUM\_BUS\_ENUMERATOR  
GUID\_DEVINTERFACE\_SIDESHOW  
GUID\_DEVINTERFACE\_STORAGEPORT  
GUID\_DEVINTERFACE\_TAPE  
GUID\_DEVINTERFACE\_USB\_DEVICE  
GUID\_DEVINTERFACE\_USB\_HOST\_CONTROLLER  
GUID\_DEVINTERFACE\_USB\_HUB  
GUID\_DEVINTERFACE\_VIDEO\_OUTPUT\_ARRIVAL  
GUID\_DEVINTERFACE\_VOLUME  
GUID\_DEVINTERFACE\_WPD  
GUID\_DEVINTERFACE\_WPD\_PRIVATE  
GUID\_DEVINTERFACE\_WRITEONCEDISK  
GUID\_DISPLAY\_ADAPTER\_INTERFACE  
GUID\_DISPLAY\_DEVICE\_ARRIVAL  
GUID\_IO\_VOLUME\_DEVICE\_INTERFACE  
GUID\_PARALLEL\_DEVICE  
GUID\_PARCLASS\_DEVICE  
GUID\_SERENUM\_BUS\_ENUMERATOR  
GUID\_VIRTUAL\_AVG\_CLASS

HardwareIds.scsi.ID Section of a TxtSetup.oem File

[headerPath](#) XML Element

[HwComponent](#) Section of a `TxtSetup.oem` File

[icon](#) XML Element

[installAllOrNone](#) XML Element

[installHeaderTitle](#) XML Element

[InstallSelectedDriver](#)

`KSCATEGORY_ACOUSTIC_ECHO_CANCEL`

`KSCATEGORY_AUDIO`

`KSCATEGORY_AUDIO_DEVICE`

`KSCATEGORY_AUDIO_GFX`

`KSCATEGORY_AUDIO_SPLITTER`

`KSCATEGORY_BDA_IP_SINK`

`KSCATEGORY_BDA_NETWORK_EPG`

`KSCATEGORY_BDA_NETWORK_PROVIDER`

`KSCATEGORY_BDA_NETWORK_TUNER`

`KSCATEGORY_BDA_RECEIVER_COMPONENT`

`KSCATEGORY_BDA_TRANSPORT_INFORMATION`

`KSCATEGORY_BRIDGE`

`KSCATEGORY_CAPTURE`

`KSCATEGORY_CLOCK`

`KSCATEGORY_COMMUNICATIONSTRANSFORM`

`KSCATEGORY_CROSSBAR`

`KSCATEGORY_DATACOMPRESSOR`

`KSCATEGORY_DATADECOMPRESSOR`

`KSCATEGORY_DATATRANSFORM`

`KSCATEGORY_DRM_DESCRAMBLE`

`KSCATEGORY_ENCODER`

`KSCATEGORY_ESCALANTE_PLATFORM_DRIVER`

`KSCATEGORY_FILESYSTEM`

`KSCATEGORY_INTERFACETRANSFORM`

`KSCATEGORY_MEDIUMTRANSFORM`

`KSCATEGORY_MICROPHONE_ARRAY_PROCESSOR`

KSCATEGORY\_MIXER  
KSCATEGORY\_MULTIPLEXER  
KSCATEGORY\_NETWORK  
KSCATEGORY\_NETWORK\_CAMERA  
KSCATEGORY\_PREFERRED\_MIDIOUT\_DEVICE  
KSCATEGORY\_PREFERRED\_WAVEIN\_DEVICE  
KSCATEGORY\_PREFERRED\_WAVEOUT\_DEVICE  
KSCATEGORY\_PROXY  
KSCATEGORY\_QUALITY  
KSCATEGORY\_REALTIME  
KSCATEGORY\_RENDER  
KSCATEGORY\_SENSOR\_CAMERA  
KSCATEGORY\_SPLITTER  
KSCATEGORY\_SYNTHESIZER  
KSCATEGORY\_SYSAUDIO  
KSCATEGORY\_TEXT  
KSCATEGORY\_TOPOLOGY  
KSCATEGORY\_TVAUDIO  
KSCATEGORY\_TVTUNER  
KSCATEGORY\_VBICODEC  
KSCATEGORY\_VIDEO  
KSCATEGORY\_VIDEO\_CAMERA  
KSCATEGORY\_VIRTUAL  
KSCATEGORY\_VPMUX  
KSCATEGORY\_WDMAUD  
KSMFT\_CATEGORY\_AUDIO\_DECODER  
KSMFT\_CATEGORY\_AUDIO\_EFFECT  
KSMFT\_CATEGORY\_AUDIO\_ENCODER  
KSMFT\_CATEGORY\_DEMULTIPLEXER  
KSMFT\_CATEGORY\_MULTIPLEXER  
KSMFT\_CATEGORY\_OTHER  
KSMFT\_CATEGORY\_VIDEO\_DECODER

KSMFT\_CATEGORY\_VIDEO\_EFFECT  
KSMFT\_CATEGORY\_VIDEO\_ENCODER  
KSMFT\_CATEGORY\_VIDEO\_PROCESSOR  
language XML Element  
legacyMode XML Element  
MediumChangerClassGuid  
MOUNTDEV\_MOUNTED\_DEVICE\_GUID  
package XML Element  
PartitionClassGuid  
promptIfDriverIsNotBetter XML Element  
quietInstall XML Element  
scanHardware XML Element  
search XML Element  
SetupDiGetWizardPage  
SetupDiMoveDuplicateDevice  
SP\_ENABLECLASS\_PARAMS  
SP\_INSTALLWIZARD\_DATA  
SP\_MOVEDEV\_PARAMS  
StoragePortClassGuid  
subDirectory XML Element  
suppressAddRemovePrograms XML Element  
suppressWizard XML Element  
Class and ClassGuid entries for INF Version Section  
System-Defined Device Setup Classes Reserved for System Use  
TapeClassGuid  
VolumeClassGuid  
watermarkPath XML Element  
welcomeIntro XML Element  
welcomeTitle XML Element  
WriteOnceDiskClassGuid

# Device and Driver Installation

12/5/2018 • 2 minutes to read • [Edit Online](#)

This section explains how devices and drivers are installed in Windows.

If you are unfamiliar with the device and driver installation process, we recommend that you start by reviewing [Roadmap for Device and Driver Installation](#). You may also want to read [Overview of Device and Driver Installation](#) for a high-level overview of this process and its components.

# Roadmap for Device and Driver Installation

11/2/2020 • 2 minutes to read • [Edit Online](#)



To install a device and driver in the Windows operating system, follow these steps:

- Step 1: Learn the fundamentals of device and driver installation in Windows.

You must understand the fundamentals of device and driver installation in the Windows family of operating systems. This will help you to make appropriate design decisions and will allow you to streamline your development process. For more information, see [Overview of Device and Driver Installations](#).

- Step 2: Learn about driver packages and their components.

A driver package consists of all the components that you must supply to install your device and support it under Windows.

To install a device or a driver, you must have system-supplied and vendor-supplied components. The system provides generic installation software for all device classes. Vendors must supply one or more device-specific components within the driver package.

For more information, see [Driver Packages](#).

- Step 3: Learn about information (INF) files.

An INF file contains the information and device settings which the system-provided [device installation components](#) use to install your [driver package](#), such as the driver for the device and any device-specific applications.

For more information, see [INF Files](#).

- Step 4: Create a driver package for your device and drivers.

Your driver package must provide an INF file, the device's driver files, as well as optionally provide additional software components. You may refer to the sample Toaster driver package to determine which components are needed for your driver package.

For more information about the components of a driver package, see [Creating a Driver Package](#).

For more information about driver packages, see the [Toaster Sample](#).

- Step 5: Test-sign your driver package during development and testing.

Test-signing refers to using a test certificate to sign a prerelease version of a [driver package](#) for use on test computers. In particular, this allows developers to sign driver packages by using self-signed certificates, such as those the [MakeCert](#) tool generates. This capability allows developers to install and test driver packages in Windows with driver signature verification enabled.

For more information, see [Signing Drivers during Development and Test](#).

- Step 6: Release-sign your driver package for distribution.

After you have tested and verified your [driver package](#), you should release-sign the driver package. Release-signing identifies the publisher of a driver package. While this step is optional, driver packages should be release-signed for the following reasons:

- Ensure the authenticity, integrity, and reliability of driver packages. Windows uses digital signatures to verify the identity of the publisher and to verify that the driver has not been altered since it was published.
- Provide the best user experience by facilitating automatic driver installation.
- Run kernel-mode drivers on 64-bit versions of Windows Vista and later versions of Windows.
- Playback certain types of next-generation premium content.

[Driver packages](#) are release-signed through either:

- A WHQL Release Signature obtained through the [Windows Hardware Compatibility Program](#) (for Windows 10), or the [Windows Hardware Certification Program](#) (for Windows 8/8.1 and older operating systems).
- A release signature created through a [Software Publisher Certificate \(SPC\)](#).

For more information, see [Signing Drivers for Public Release](#).

- Step 7: Distribute your driver package.

The final step is to distribute the [driver package](#). If your driver package meets the quality standards that are defined in the the [Windows Hardware Compatibility Program](#) (for Windows 10), or the [Windows Hardware Certification Program](#) (for Windows 8/8.1 and older operating systems), you can distribute it through Microsoft Windows Update program. For more information, see [Publishing a driver to Windows Update](#).

These are the basic steps. Additional steps might be necessary based on the installation needs of your individual device and driver.

# Overview of Device and Driver Installation

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Windows operating system installs devices when the system restarts or when a user plugs in (or manually installs) a Plug and Play (PnP) device.

Specifically, Windows enumerates 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 Windows determine which devices are present.

## In this section

- [Step 1: The New Device is Identified](#)
- [Step 2: A Driver for the Device is Selected](#)
- [Step 3: The Driver for the Device is Installed](#)
- [Overview of the Driver Selection Process](#)

# Step 1: The New Device is Identified

10/7/2019 • 2 minutes to read • [Edit Online](#)

Before a driver is installed for a new device, the bus or hub driver to which the device is connected assigns a [hardware identifier \(ID\)](#) to the device. Windows uses hardware IDs to find the closest match between a device and a [driver package](#) that contains the driver for the device. For more information about hardware IDs, see [Device Identification Strings](#).

The format of the hardware ID typically consists of the following:

- A bus-specific prefix, such as PCI\ or USB\.
- Vendor-specific identifiers for the device, such as a vendor, model, and revision identifier. The format of these identifiers within the hardware ID is also specific to the bus driver.

An independent hardware vendor (IHV) can also define one or more [compatible IDs](#) for the device. Compatible IDs have the same format as hardware IDs; however, they are typically more generic than hardware IDs and do not require specific manufacturer or model information. Windows uses these identifiers to select a [driver package](#) for a device if the operating system cannot find a matching driver package for the device's hardware ID. IHVs specify one or more compatible IDs for the device within the driver package's [INF file](#).

Windows uses hardware IDs and compatible IDs to search for a [driver package](#) for the device. It finds a matching driver package for the device by comparing the device's hardware IDs and compatible IDs against those IDs that are specified within the package's [INF file](#).

For example, when a user plugs a wireless local area network (WLAN) adapter into the port of a USB hub that is attached to the computer, the following steps occur:

1. The device is detected by the USB hub driver. Based on information that it queries from the adapter, the hub driver creates a hardware ID for the device.

For example, the USB hub driver could create a hardware ID of USB\VID\_1234&PID\_5678&REV\_0001 for the WLAN adapter, where:

- VID\_1234 is the identifier of the vendor.
- PID\_5678 is the product, or model, identifier of the device.
- REV\_0001 is the revision identifier of the device.

For more information about the format of USB hardware IDs, see [Identifiers for USB Devices](#).

2. The USB hub driver notifies the [Plug and Play \(PnP\) manager](#) that a new device was detected. The PnP manager queries the hub driver for all of the device's hardware IDs. The hub driver can create multiple hardware IDs for the same device.
3. The [PnP manager](#) notifies Windows that a new device needs to be installed. As part of this notification, Windows is provided with the list of hardware IDs.
4. Windows starts a search for a [driver package](#) that matches one of the device's hardware IDs. If Windows cannot find a matching hardware ID, it searches for a driver package that has a matching compatible ID for the device.

For more information about this process, see [Step 2: A Driver for the Device is Selected](#).

Each bus driver constructs hardware IDs in its own, bus-specific manner.

For examples of standardized identifiers for other buses, see:

- Identifiers for PCI Devices
- Identifiers for SCSI Devices
- Identifiers for IDE Devices
- Identifiers for PCMCIA Devices
- Identifiers for ISAPNP Devices
- Identifiers for 1394 Devices
- Identifiers for Secure Digital (SD) Devices
- Identifiers for USB Devices

# Step 2: A Driver for the Device is Selected

11/2/2020 • 3 minutes to read • [Edit Online](#)

After a new device is detected and identified, Windows and its [device installation components](#) follow these steps:

1. Windows searches for an appropriate [driver package](#) for the device. For more information about this step, see [Searching for the Driver Package](#).
2. Windows selects the most appropriate driver for the device from one or more driver packages. For more information about this step, see [Selecting the Driver](#).

## Searching for the Driver Package

Using the [hardware identifier \(ID\)](#) that is reported by the bus or hub driver, Windows searches for [driver packages](#) that match a device. A driver package matches a device if the hardware ID matches a hardware ID or [compatible ID](#) in an [INF Models section](#) entry of the driver package's [INF file](#).

Depending on the operating system version, Windows searches for matching driver packages in various locations as described in the following table.

SEARCH PHASE	WINDOWS 7	WINDOWS 8 AND LATER VERSIONS OF WINDOWS
Before a driver is installed	DevicePath Windows Update <a href="#">Driver store</a>	<a href="#">Driver store</a>
After initial driver is selected	Not applicable	DevicePath Windows Update

For example, if a user plugs a wireless local area network (WLAN) adapter into a port of a USB hub on a computer that is running Windows 7, the following steps occur:

- After the USB hub driver creates a list of hardware IDs for the WLAN adapter, Windows first searches the [driver store](#) for a matching [driver package](#) for the device.
- The device installation process searches for a matching driver package from one of the following locations:
  - The Universal Naming Convention (*UNC*) paths that are identified by the **DevicePath** registry value of `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion`.
  - Windows Update
  - The distribution medium that the independent hardware vendor (IHV) provided for the device.

If a driver package is found, Windows stages the package into the driver store from which the driver for the device will be installed.

**Note** Starting with Windows Vista, the operating system always installs a [driver package](#) from the [driver store](#). If a matching driver package is found in another location, Windows first stages the package to the driver store before it installs the driver for the device.

As another example, if a user plugs a WLAN adapter into a port of a USB hub on a computer that is running

Windows 8, the following steps occur:

- After the USB hub driver creates a hardware ID for the WLAN adapter, Windows first searches the driver store for a matching [driver package](#) for the device. If a driver package is found in the driver store, Windows installs it on the device. This allows the device to begin working quickly.
- In a separate process, Windows searches Windows Update and the DevicePath for a better matching driver than was installed. If one is found, the driver is staged into the driver store, and then installed onto the device.

For more information about the [driver package](#) search process, see [Where Windows Searches for Drivers](#).

## Selecting the Driver

As soon as Windows has found one or more matching [driver packages](#) for the device, Windows selects the best driver by following these steps:

1. If Windows has found only one matching driver package, it installs the driver from that package for the device.
2. If Windows has found multiple matching driver packages, Windows first assigns a ranking value to the driver from each driver package. If only one driver has the lowest rank value, it installs the driver from that package for the device.

For more information about the ranking process, see [How Windows Ranks Drivers](#).
3. If multiple drivers have the same lowest rank value, Windows uses the following criteria to select the best driver for the device:
  - Whether the driver is digitally signed. Starting with Windows Vista, Windows always selects a signed driver instead of an unsigned driver regardless of other selection criteria. For more information about digital signatures for drivers, see [Driver Signing](#).
  - The driver date and version, where the date and version are specified by the [INF DriverVer directive](#) that is contained in the driver package's [INF file](#).

Once Windows has selected a driver for the device, Windows installs the driver as described in [Step 3: The Driver for the Device is Installed](#).

For more information about how drivers are selected for a device, see [How Windows Selects Drivers](#).

# Step 3: The Driver for the Device is Installed

11/2/2020 • 2 minutes to read • [Edit Online](#)

After Windows has selected the best driver for the new device, it saves information about the device and driver in the following way:

- Multiple devices of the same model and version can be connected to the computer. Each device connection is known as a *device instance*.

Windows represents each device instance as a unique device node (*devnode*). A devnode contains information about the device, such as whether the device was started and which drivers have registered for notification on the device.

- Windows represents a driver for the device as a *driver node*. A driver node is based on the information from a matching device entry within the **INF Models section** of the **driver package's INF file**. A driver node includes all the software support for a device, such as any services, device-specific co-installers, and registry entries.

As soon as the device and driver are instantiated, Windows installs the driver by following these steps:

1. Based on directives within the **driver package's INF file**, Windows does the following:
  - Copies the driver binaries and other associated files to locations on the hard disk as specified by the **INF CopyFiles directive**.
  - Performs any device-instance related configuration, such as registry key writes.
2. Windows determines the **device setup class** from the **Class** and **ClassGuid** entries in the **INF Version section** of the **driver package's INF file**. To optimize device installation, devices that are set up and configured in the same manner are grouped into the same device setup class.
3. As soon as the driver files are copied, Windows transfers control to the **Plug and Play (PnP) manager**. The PnP manager loads the drivers and starts the device.
4. 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, finally, 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.

As soon as this step is complete, the device is installed and ready to be used.

# System-Provided Device Installation Components

10/16/2019 • 2 minutes to read • [Edit Online](#)

The following list describes the device installation components that are provided by the Windows operating system:

## Plug and Play (PnP) Manager

The Plug and Play (PnP) manager provides the following support for PnP functionality within Windows:

- Device detection and enumeration while the system is booting
- Adding or removing devices while the system is running

For more information, see [PnP Manager](#).

## SetupAPI

The Setup application programming interface (*SetupAPI*) includes the general setup functions (**SetupXxx**) and the device installation functions (**SetupDiXxx** and **DiXxx**). 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, and registering device co-installers. Most of the other device installation components call these functions.

For more information, see [SetupAPI](#).

## Configuration Manager API

The PnP configuration manager API provides basic installation and configuration operations that are not provided by SetupAPI. The PnP configuration manager functions perform low-level tasks such as obtaining the status of a device node (*devnode*) and managing resource descriptors. These functions are primarily called by SetupAPI but can also be called by other device installation components.

## Driver Store

The driver store is a trusted collection of in-box and third-party [driver packages](#). The operating system maintains this collection in a secure location on the local hard disk. Only the driver packages in the driver store can be installed for a device.

For more information, see [Driver Store](#).

## Device Manager

With Device Manager, you can view and manage the devices on a system. For example, you can view device status and set device properties.

For more information, see [Using Device Manager](#). Also, see the Help documentation in Device Manager.

# Plug and Play Manager

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Plug and Play (PnP) manager provides the support for PnP functionality in Windows and is responsible for the following PnP-related tasks:

- Device detection and enumeration while the system is booting
- Adding or removing devices while the system is running

The kernel-mode PnP manager notifies the user-mode PnP manager that a new device is present on the system and must be installed.

The kernel-mode PnP manager also calls the *DriverEntry* and *AddDevice* routines of a device's driver and sends the **IRP\_MN\_START\_DEVICE** request to start the device.

The PnP manager maintains the *Device Tree* that keeps track of the devices in the system. The device tree contains information about the devices present on the system. When the computer starts, the PnP manager builds this tree by using information from drivers and other components, and updates the tree as devices are added or removed.

# Driver Store

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, the driver store is a trusted collection of inbox and third-party [driver packages](#).

The operating system maintains this collection in a secure location on the local hard disk. Only the driver packages in the driver store can be installed for a device.

When a driver package is copied to the driver store, all of its files are copied. This includes the [INF file](#) and all files that are referenced by the INF file. All files that are in the driver package are considered critical to the device installation. The INF file must reference all of the required files for device installation so that they are present in the driver store. If the INF file references a file that is not included in the driver package, the driver package is not copied to the store.

The process of copying a driver package to the driver store is called *staging*. A driver package must be *staged* to the driver store before the package can be used to install any devices. As a result, driver staging and device installation are separate operations.

A driver package is staged to the driver store by being verified and validated.

## Verifying the driver package integrity

Software integrity has become a top priority for Independent Hardware Vendors (IHVs) and Original Equipment Manufacturers (OEMs). Concerned by the increase in malicious software on the Internet, these customers want to be sure that their software has not been tampered with or corrupted.

Before a driver package is copied to the driver store, the operating system first verifies that the digital signature is correct. For more information about digital signatures, see [Driver Signing](#).

## Validating the driver package

The operating system validates the driver package in the following ways:

- The current user must have permission to install the driver package.
- The [INF file](#) of the driver package is syntactically correct, and all files referenced by the INF files are present in the driver package.

After a driver package has passed integrity and syntax checks, it is copied to the driver store. Afterwards, the operating system uses the driver package to automatically install new devices without requiring user interaction.

Once files are staged to the driver store, they should not be removed or modified in any way. Additionally, new files should not be added to the driver store outside of the staging process. This includes files being added, removed, or modified directly through programmatic calls, or indirectly through INF directives that will be processed at a later time.

# Vendor-Provided Device Installation Components

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic describes device installation components that are provided by an IHV or OEM.

## Driver Package

A [driver package](#) consists of all the software components that you must supply for your device to be supported under Windows. These components include the following:

- An [INF file](#), which provides information about the devices and drivers to be installed. For more information, see [Creating an INF File](#).
- A [catalog file](#), which contains the digital signature of the [driver package](#). For more information, see [Digital Signatures](#).
- The driver for the device.

## Drivers

A driver allows the system to interact with the hardware device. Windows copies the driver's binary file (.sys) to the %SystemRoot%\system32\drivers directory when the device is installed. Drivers are required for most devices.

For more information, see [Choosing a Driver Model](#).

For more information about drivers for Windows, see [Getting Started with Windows Drivers](#).

# Device Installation Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

The software that is 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 Windows components use to install support for the device. Windows copies this file to the `%SystemRoot%\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. Windows copies this file to the `%SystemRoot%\system32\drivers` directory when the driver is installed. Drivers are required for most devices.

For more information, see [Choosing a Driver Model](#).

- 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 obtains 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 [WHQL release signatures](#).

- Other files

A [driver package](#) can contain other files, such as a custom device installation application, a device icon, or a driver library file (such as for video drivers).

For more information, see [Providing Device Property Pages](#).

Also, see the device-type-specific documentation in the WDK.

The WDK includes various sample installation files. For more information, see [Sample Device Installation Files](#)

# Sample Device Installation Files

12/5/2018 • 2 minutes to read • [Edit Online](#)

The Windows Driver Kit (WDK) includes various sample installation files, such as INF files, device installation applications, class installers, co-installers, and property page providers.

Most sample installation files are located under the *src\setup* subdirectory of the WDK. Also see the *src\general\toaster* subdirectory of the WDK.

# Device Installation Types

12/5/2018 • 2 minutes to read • [Edit Online](#)

Windows uses INF files to install a driver package on a computer or device. All Windows platforms support universal INF files, while only Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) supports legacy INF files.

## INF files for universal and mobile driver packages

If you are building a universal or mobile driver package, you must supply a universal INF file. Universal INFs contain only the subset of INF sections and directives that are required to install and configure a device. These directives can be performed on an offline system, without any runtime operations. For more information, see [Using a Universal INF File](#).

## INF files for desktop driver packages

If you are building a desktop-only driver package, your INF file can use legacy, non-universal INF syntax.

Windows 10 for desktop editions continues to support legacy INF behavior, such as co-installers and class installers.

# Overview of the Driver Selection Process

11/2/2020 • 3 minutes to read • [Edit Online](#)

Windows represents a driver as a *driver node*, which includes all the software support for a device, such as any services, device-specific co-installers, and registry entries. The services for a device include a function driver and any upper-level and lower-level device filter drivers.

Some devices require a vendor-supplied driver that is designed specifically for that device or one that is designed to support a family of devices. However, other devices can be driven by a system-supplied driver that supports all the devices of a given [device setup class](#). Windows selects the driver that most closely matches the device. If Windows does not find such a driver, it selects from increasingly more general drivers.

## How Windows Searches for Drivers

Windows searches in specific locations for drivers that match a device. A driver matches a device if the following are true:

- One of the Plug and Play (PnP) [device identification strings](#) that is reported by the bus driver for the device matches a device identification string in an [INF Models section](#) entry of the driver's [INF file](#).
- If the matching device identification string in an [INF Models section](#) entry specifies a *TargetOSVersion* decoration, the decoration matches the operating system version on which the device is to be installed.

For more information about the *TargetOSVersion* decoration, see [Combining Platform Extensions with Operating System Versions](#).

For more information about where Windows searches for matching drivers, see [Where Windows Searches for Drivers](#).

## How Windows Ranks Drivers

Windows creates a list of all the matching drivers and assigns each driver a rank. Windows represents each driver's rank with an integer value that is greater than or equal to zero.

For more information about the ranking process, see [How Windows Ranks Drivers](#).

Starting with Windows Vista, Windows also ranks drivers based on whether the driver is digitally signed. Windows ranks drivers based on a digital signature as follows:

- If the [AllSignersEqual Group Policy](#) is disabled, Windows ranks drivers that are signed with a Microsoft signature higher than drivers that are signed with an [Authenticode](#) signature. This ranking occurs even if a driver that is signed with an Authenticode signature is, in all other aspects, a better match for a device.
- If the [AllSignersEqual Group Policy](#) is enabled, Windows ranks all digitally signed drivers equally.

**Note** Starting with Windows 7, the [AllSignersEqual Group Policy](#) is enabled by default. In Windows Vista and Windows Server 2008, the [AllSignersEqual Group Policy](#) is disabled by default. IT departments can override the default ranking behavior by enabling or disabling the [AllSignersEqual Group Policy](#).

Signatures from a Windows signing authority include the following:

- Premium Windows Hardware Quality Labs (WHQL) signatures and standard WHQL signatures
- Signatures for inbox drivers
- Windows Sustained Engineering (Windows SE) signatures

- A WHQL signature for a Windows version that is the same or later than the [LowerLogoVersion](#) value of the driver's device setup class

## How Windows Selects Drivers

Windows selects the driver with the lowest rank value as the best match for the device.

However, if there are multiple equally ranked drivers that are a best match for a device, Windows uses the driver's date and version to select a driver. The driver's date and version are specified by the [INF DriverVer directive](#) that is contained in the driver's [INF file](#).

Windows uses the following criteria to select a driver for a device:

- Windows selects the driver that has the lowest rank value as the best match for the device.
- For drivers that have equal rank, Windows selects the driver that has the most recent date.
- For the drivers that have equal rank and date, Windows selects the driver that has the highest version.
- For drivers that have equal rank, date, and version, Windows can select any driver.

# How Windows selects a driver for a device

11/2/2020 • 2 minutes to read • [Edit Online](#)

When a device is attached, Windows needs to find a corresponding device driver to install.

In Windows 10, this matching process happens in two phases. First, Windows 10 installs the best matching driver in the [driver store](#), allowing the device to begin operation quickly. After that driver is installed, Windows 10 also:

- Downloads any matching [driver packages](#) from Windows Update and puts them in the [driver store](#).
- Searches for driver packages that were preloaded in the locations specified by the **DevicePath** registry value.

The **DevicePath** registry value is located under the following subkey:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion`. By default, the **DevicePath** value specifies the `%SystemRoot%\INF` directory.

If Windows 10 finds a better matching driver package in those locations than was initially installed, Windows replaces the driver it installed from the driver store with the better match.

In Windows versions before Windows 8, the driver matching process looks only in **DevicePath**, if one is specified, and defaults to Windows Update otherwise.

The following table provides a quick summary of the information above:

SEARCH PHASE	WINDOWS 7 MATCH ORDER	WINDOWS 8, WINDOWS 10 MATCH ORDER
Before a driver is installed	<b>DevicePath</b> ; Windows Update; <a href="#">Driver store</a>	<a href="#">Driver store</a>
After initial driver is selected	Not applicable	<b>DevicePath</b> ; Windows Update

## NOTE

In Windows 10, version 1709 and greater, Windows offers the best matching driver, which is not necessarily the most recent. The driver selection process considers hardware ID, date/version, and critical/automatic/optional category. Windows prioritizes critical or automatic drivers highest. If a matching driver is not found, WU looks next for optional drivers. As a result, an older critical driver of otherwise equal value takes precedence over a newer optional driver.

Starting with Windows 10, version 2004, Windows automatically offers only the best automatic/critical matching driver, searching both the computer and Windows Update. To see matching drivers in the optional category, go to **Settings > Update & Security > Windows Update > View optional updates > Driver updates**. Windows still uses the same criteria to rank and select a driver.

# How Windows Ranks Drivers

11/2/2020 • 2 minutes to read • [Edit Online](#)

## NOTE

This page describes how Windows determines a driver rank value for a given driver that matches on a device. To understand how driver rank and other factors (including INF date, driver version, etc.) are used to determine which driver Windows selects for a device, see [How Windows Selects Drivers](#).

Windows assigns a rank to a driver that matches a device. The rank indicates how well the driver matches the device. A driver rank is represented by an integer that is equal to or greater than zero. The lower the rank, the better a match the driver is for the device.

The rank of a driver is a composite value that depends on a driver's signature, the features that are supported by the driver, and the type of match between the [device identification strings](#) that are reported by a device and the device identification strings that are specified in the entries of an [INF Models section](#) of a driver INF file.

A rank is represented by a value of type DWORD. A rank is sum of a signature score, a feature score, and an identifier score. A rank is formatted as  $0xSSGGTHHH$ , where  $S$ ,  $G$ ,  $T$ , and  $H$  are four-bitfields and the  $SS$ ,  $GG$ , and  $THHH$  fields represent the three ranking scores, as follows:

- The [signature score](#) ranks a driver based on whether its signature is trusted. The signature score depends only on the value of the  $SS$  field. An unspecified signature score is represented as  $0xSS0000000$ .

For an overview on how Windows Vista and later versions of Windows use a driver's signature to determine how the driver is installed, see [Signature Categories and Driver Installation](#).

- The [feature score](#) ranks a driver based on the features that the driver supports. The feature score depends only on the value of the  $GG$  field. An unspecified feature score is represented as  $0x00GG0000$ .
- The [identifier score](#) ranks a driver based on the type of match between a [device identification string](#) that is reported by a device and a device identification string that is listed in an entry of an [INF Models section](#) of a driver INF file. The identifier score depends only on the value of the  $THHH$  field. An unspecified identifier score is represented as  $0x0000THHH$ .

For information about entries in the SetupAPI log that indicate the rank of a driver and the type of driver signature, see [Driver Rank Information in the SetupAPI Log](#).

# Signature Categories and Driver Installation

12/21/2018 • 2 minutes to read • [Edit Online](#)

Before Windows Vista and later versions of Windows installs a driver, the operating system analyzes the driver's signature. If a signature is present, Windows validates all the driver's files against that signature. Based on the results of this analysis, Windows puts the driver in one of the following categories:

## **Signed by a Microsoft Windows signing authority.**

These drivers are either inbox, signed by WHQL, or signed by Windows Sustained Engineering.

## **Signed by a trusted publisher.**

These drivers were signed by a third-party, and the user has explicitly selected to always trust signed drivers from this publisher.

## **Signed by an untrusted publisher.**

These drivers were signed by a third-party, and the user has explicitly selected to never trust drivers from this publisher.

## **Signed by publisher of unknown trust.**

These drivers were signed by a third-party, and the user has not indicated whether to trust this publisher.

## **Altered.**

These drivers are signed, but Windows has detected that at least one file in the [driver package](#) was altered since the package was signed.

## **Unsigned.**

These drivers are either unsigned or have an invalid signature. Valid signatures must be created by using a certificate that was issued by a trusted Certificate Authority (CA).

After the driver is categorized, Windows determines whether it should be installed. The process depends on the type of user. For nonadministrative and standard users, Windows does not prompt the user. It automatically installs drivers signed through either a Windows signing authority or a trusted publisher, and silently refuses to install all others.

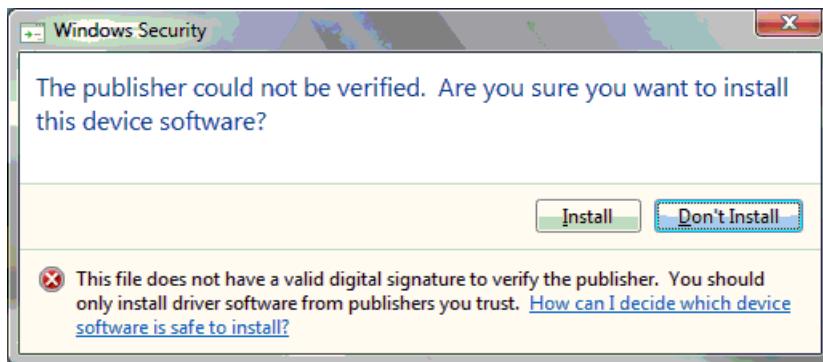
Administrative users have more flexibility:

- If a driver is signed by a Windows signing authority or a trusted publisher, Windows installs the driver without prompting the user.
- If the driver is signed by an untrusted publisher, Windows does not install the driver. Windows does not prompt the user in this case, but logs an error to *Setupapi.dev.log*.
- If the driver was signed by a publisher of unknown trust, Windows prompts the user with the following Windows Security dialog box.



The user must explicitly select whether to install this driver. The user is also able to add the publisher to the list of trusted publishers on the user's system. If the user selects this option, all future drivers from this publisher are treated as trusted when installed on the user's system. If the user does not select this option, the publisher remains in the unknown trust category and administrative users continue to receive this prompt if they attempt to install additional drivers from this publisher.

- If the driver lacks a valid signature or was altered, Windows prompts administrators with the following Windows Security dialog box. Again, the user must explicitly select whether to install the driver.



**Note** On Windows Vista and later versions of Windows, in order for users to play next-generation premium content, such as HD DVD and other formats that are licensed under the *Advanced Access Content System (AACSB) Specification*, all kernel-mode components on their system must be signed. That means that, if an administrative user selects to install an unsigned or altered driver, the system is not allowed to play premium content. For more information about how to protect media components in Windows Vista, see [Code Signing for Protected Media Components in Windows Vista](#).

# Signature Score

11/2/2020 • 2 minutes to read • [Edit Online](#)

This page describes the signature score for Windows drivers. Signature score is one of three scores comprised in a driver rank. Information on this page applies only to Windows Vista and later versions of the operating system.

A driver rank is formatted as  $0xSSGGTHHH$ , where the value of  $0xSS000000$  is the signature score, the value of  $0x00GG0000$  is the [feature score](#), and the value of  $0x0000THHH$  is the [identifier score](#).

The signature score ranks a driver according to how the driver is signed, as follows:

- Windows assigns the best signature score (lowest signature score value) to drivers that have a trusted signature. This includes the following:
  - Premium WHQL signatures and standard WHQL signatures.
  - Signatures for inbox drivers.
  - Windows Sustained Engineering (Windows SE) signatures.
  - A third-party signature using Authenticode technology. Valid third-party signature types include the following:
    - Drivers signed by using a code signing certificate from an Enterprise Certificate Authority (CA).
    - Drivers signed by using a code signing certificate issued by a Class 3 CA.
    - Drivers signed by using a code signing certificate created by the [MakeCert Tool](#).
- Windows assigns the second best signature score to [driver packages](#) that do not have a valid signature, but the driver is installed by an [INF DDInstall section](#) that has an .nt platform extension.

For more information about the .nt extension, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

- Windows assigns the third best signature score to driver packages that do not have a valid signature, and the driver is not installed by an INF DDInstall section that has an .nt platform extension.
- Windows assigns the fourth and worst signature score (highest signature score value) to drivers that are not signed or whose signing state is unknown.

For more information about driver ranking, see [How Windows Ranks Drivers](#).

# LowerLogoVersion

11/2/2020 • 2 minutes to read • [Edit Online](#)

**LowerLogoVersion** is a [device setup class property](#) that affects the signature score of a driver as follows:

- Windows assigns the best signature score to drivers that have a WHQL signature for a Windows version that is the same or later than the **LowerLogoVersion** value.
- Windows assigns the next best signature score to a driver that was signed by a third-party using Authenticode technology and to a driver that has a WHQL signature for a Windows version earlier than the **LowerLogoVersion** value.

A **LowerLogoVersion** value is a NULL-terminated string that specifies the Windows version, as indicated in the following table.

WINDOWS VERSION	LOWERLOGOVERSION VALUE
Windows 7	6.1
Windows Server 2008	6.1
Windows Vista	6.0
Windows Server 2003	5.2
Windows XP	5.1
Windows 2000	5.0

The system default **LowerLogoVersion** value for a system-defined [device setup class](#) is "5.1." This means that drivers that have a WHQL signature for Windows Server 2003 and Windows XP have the same signature score as a driver that is signed by Microsoft for Windows Vista and later versions of Windows.

For more information about driver ranking, see [How Windows Ranks Drivers \(Windows Vista and Later\)](#).

# Feature Score

11/2/2020 • 2 minutes to read • [Edit Online](#)

A driver rank is formatted as `0xSSGGTHHH`, where the value of `0xSS000000` is the [signature score](#), the value of `0x00GG0000` is the feature score, and the value of `0x0000THHH` is the [identifier score](#).

The feature score provides a way to rank drivers based on the features that a driver supports. For example, feature scores might be defined for a [device setup class](#) that distinguishes between drivers based on class-specific criteria. The feature score supplements the identifier score, making it possible for driver writers to more easily and precisely distinguish between different drivers for a device that is based on well-defined criteria.

Microsoft defines feature score usage for particular device classes. Feature score is not required, so many device classes will not have feature score usage specified. In this case, the default feature score (0xFF) is expected, and will be assigned in the absence of a feature score defined in the INF of a driver.

When Microsoft does not explicitly require feature score for a device class, the driver should:

- Not define a feature score in the driver INF (Windows will default to 0xFF)  
or
- Explicitly define a feature score of 0xFF in the driver INF

The feature score for a driver is set by the [INF FeatureScore Directive](#) in the [INF DDInstall section](#) of the INF file that installs a device. The feature score is set as follows:

```
[DDInstallSectionName]
...
FeatureScore=featurescore
```

where `DDInstallSectionName` is the name of the `DDInstall` section and `featurescore` is a single-byte hexadecimal number between 0x00 and 0xFF. Windows computes the feature score for a driver based on the `featurescore` value of the **FeatureScore** directive:

```
feature score = (featurescore * 0x10000)
```

If the [INF FeatureScore Directive](#) is not specified in the INF file, Windows uses a default feature score is 0x00FF0000 for the driver, which indicates that there is no preference based on the features of the driver. The lower the feature score, the better the rank, where the best feature score is 0x00000000.

For example, the following sets the feature score of a driver to 0x00FD0000:

```
[DDInstallSectionName]
...
FeatureScore=xFD
```

For more information about driver ranking, see [How Windows Ranks Drivers](#).

# Identifier Score

12/5/2018 • 4 minutes to read • [Edit Online](#)

A driver rank is formatted as  $0xSSGGTHHH$ , where the value of  $0xSS000000$  is the [signature score](#), the value of  $0x00GG0000$  is the [feature score](#), and the value of  $0x0000THHH$  is the identifier score.

The identifier score ranks a driver based on the type of match between a Plug and Play (PnP) [device identification string](#) that is reported by the bus driver of a device and a corresponding device identification string that is specified in an entry of an [INF Models section](#) of a driver INF file.

The identifier score is a sum of an identifier-match-type score and an identifier-list-position score. The identifier-match-type score ranks a driver according to whether a device [hardware ID](#) or a device [compatible ID](#) matches a hardware ID or a compatible ID in an entry of an INF *Models* section. A match between a device hardware ID and a hardware ID in an entry of an INF *Models* section is called a hardware ID match. A match where at least one of the matching identifiers is a compatible ID is called a compatible ID match.

For a given identifier-match type, the identifier-list-position score ranks a driver according to the position of the matching identifier in the hardware ID list or the compatible ID list for a device and the position of the matching identifier in an entry of an INF *Models* section. Specifically, each device has an ordered list of hardware IDs and an ordered list of compatible IDs that are reported by the bus driver for the device. The identifiers are ordered in the list from the most specific to the most generic functionality. In addition, each INF *Models* section entry has one hardware ID and an optional list of compatible IDs that are listed in order of the most specific to the most generic functionality, as follows:

```
device-description=install-section-name, hw-id, [compatible-id, ...] ...
```

The first identifier in a device identifier list has an identifier-list-position score of 0x0000, the second identifier has an identifier-list-position score of 0x0001, and so on. Because an INF *Models* section entry only has one hardware ID, the identifier-list-position score of this hardware ID is always 0x0000.

The following lists the identifier scores for the four types of identifier-match types, where the value of  $0x00007000$  is the identifier-match-type score and the value of  $0x00000HHH$  is the identifier-list-position score:

- A match between a device hardware ID and a hardware ID in an INF *Models* section entry is the best type of identifier match. A match of this type is called a *hardware ID match*.

The identifier-match-type score is 0x00000000 and the value of  $0x0000HHH$  is the identifier-list-position score of the matching hardware ID in the list of device hardware IDs.

Identifier scores for this match type range from 0x00000000 through 0x00000FFF.

For this type of match, the value 0x00000000 is the best identifier score and the value 0x00000FFF is the worst identifier score.

- A match between a device hardware ID and a compatible ID in an INF *Models* section entry is the second best type of identifier match. A match of this type is called a *compatible ID match*.

The identifier-match-type score is 0x00001000 and the value of  $0x0000HHH$  equals the identifier-list-position score of the matching hardware ID in the list of device hardware IDs.

Identifier scores for this match type range from 0x00001000 to 0x00001FFF.

For this type of match, the value 0x00001000 is the best identifier score and the value 0x00001FFF is the

worst identifier score.

- A match between a device compatible ID and a hardware ID in an INF *Models* section entry is the third best type of identifier match. A match of this type is also known as a *compatible ID match*.

The identifier-match-type score is 0x00002000 and the value of  $0x00000HHH$  equals the identifier-list-position score of the matching compatible ID in the list of device compatible IDs.

Identifier scores for this type of identifier match range from 0x00002000 to 0x00002FFF.

For this type of match, the value 0x00002000 is the best identifier score and the value 0x00002FFF is the worst identifier score.

- A match between a device compatible ID and a compatible ID in an INF *Models* section entry is the fourth best type of identifier match. A match of this type is also known as a *compatible ID match*. Identifier scores for this type of identifier match are in the range of 0x00003000 to 0x00003FFF, where:

- The identifier-match-type score is 0x3000.
- The value of  $0x0HHH$  equals  $(j + k * 0x100)$ , where  $j$  equals the identifier-list-position score of the matching device compatible ID and  $k$  equals the identifier-list-position score of the matching compatible ID in an INF *Models* section entry.

For this type of match, the value 0x00003000 is the best identifier score and the value 0x00003FFF is the worst identifier score.

For more information about driver ranking, see [How Windows Ranks Drivers](#).

# Driver Rank Ranges

12/5/2018 • 2 minutes to read • [Edit Online](#)

A driver rank is formatted as  $0xSSGGTHHH$ , where the value of  $0xSS000000$  is the [signature score](#), the value of  $0x00GG0000$  is the [feature score](#), and the value of  $0x0000THHH$  is the [identifier score](#).

The following table lists all the valid driver rank ranges, where  $0xSS000000$  represents a valid signature score,  $0x00GG0000$  represents a valid feature score, and the range of the identifier score for each type of identifier match is shown.

DRIVER RANK	IDENTIFIER SCORE	DESCRIPTION
$0xSSGG0000-0xSSGG0FFF$	$0x0000-0x0FFF$	A device hardware ID matched the hardware ID in an INF <i>Models</i> section entry.
$0xSSGG1000-0xSSGG1FFF$	$0x1000-0x1FFF$	A device hardware ID matched a compatible ID in an INF <i>Models</i> section entry.
$0xSSGG2000-0xSSGG2FFF$	$0x2000-0x2FFF$	A device compatible ID matched the hardware ID in an INF <i>Models</i> section entry.
$0xSSGG3000-0xSSGG3FFF$	$0x3000-0x3FFF$	A device compatible ID matched a compatible ID in an INF <i>Models</i> section entry.

# Driver Rank Example

12/5/2018 • 2 minutes to read • [Edit Online](#)

Consider a device that has the following lists of [device identification strings](#), where the HwID\_N and CID\_N names represent actual [hardware IDs](#) and [compatible IDs](#):

- List of hardware IDs

```
HwID_1, HwID_2
```

- List of compatible IDs

```
CID_1, and CID_2
```

The first hardware ID in a list of hardware IDs is the most specific identifier for the device. In this example, that is HwID\_1.

Also assume there is an INF file that has an [INF Models section](#) that has the following entry, where the INF\_XXX\_N names represent actual hardware IDs and compatible IDs:

```
DeviceDesc1 = InstallSection1, INF_HwID_1, INF_CID_1, INF_CID_2
```

In addition, assume that the INF *DD\Install* section named *InstallSection1* has the following **FeatureScore** directive, where the corresponding feature score of the driver is 0x00GG0000:

```
FeatureScore=0xGG
```

The following table lists the rank of each possible match between the identifiers that are reported by the device and the identifiers that are listed in the INF *Models* section entry. The rank is the sum of the [signature score](#), which is represented by 0xSS000000, the [feature score](#), which is represented by 0x00GG0000, and the [identifier score](#), which depends, in each case, on the two identifiers that matched.

DEVICE IDENTIFIERS	IDENTIFIERS IN THE INF MODELS SECTION ENTRY		
	INF_HwID_1	INF_CID_1	INF_CID_2
HwID_1	Rank 0xSSGG0000	Rank 0xSSGG1000	Rank 0xSSGG1000
HwID_2	Rank 0xSSGG0001	Rank 0xSSGG1001	Rank 0xSSGG1001
CID_1	Rank 0xSSGG2000	Rank 0xSSGG3000	Rank 0xSSGG3100
CID_2	Rank 0xSSGG2001	Rank 0xSSGG3001	Rank 0xSSGG3101

# Driver Rank Information in the SetupAPI Log

12/5/2018 • 2 minutes to read • [Edit Online](#)

Windows uses a signature indicator to represent the signature type. Windows saves this information in a [driver store](#) database for internal use.

When Windows installs a driver, SetupAPI adds a *Rank* entry that logs the driver rank in hexadecimal format and a Signer Score entry that logs the signature type. The following is an excerpt from a SetupAPI device log that shows, in bold font style, an example of a Rank entry and an example of a Signer Score entry. In this excerpt, the Rank entry indicates that the driver has a rank of "0x0dff0000" and the Signer Score entry indicates that the driver is an inbox driver.

```
dvi:      Created Driver Node:  
dvi:      HardwareID - ROOT\UPDATE  
dvi:      InfName -  
D:\Windows\System32\DriverStore\FileRepository\machine.inf_36c3d8da\machine.inf  
dvi:      DevDesc - Microcode Update Device  
dvi:      DrvDesc - Microcode Update Device  
dvi:      Provider - Microsoft  
dvi:      Mfg - (Standard system devices)  
dvi:      InstallSec - UPDATE_DRV  
dvi:      ActualSec - UPDATE_DRV  
dvi:      Rank - 0x0dff0000  
dvi:      Signer - Microsoft Windows Component Publisher  
dvi:      Signer Score - INBOX  
dvi:      DrvDate - 10/01/2002  
dvi:      Version - 6.0.5270.8
```

The following is a list of the Signer Score entries that Windows logs in the SetupAPI device log for each type of signature:

Premium WHQL signature

"Signer Score - WHQL Logo Gold"

Standard WHQL signature

"Signer Score - WHQL Logo Silver"

An unspecified Microsoft signature

"Signer Score - WHQL Unclassified"

A WHQL signature for a Windows version earlier than Windows Vista and equal to or later than the [LowerLogoVersion](#) value of the driver's device setup class.

"Signer Score - WHQL"

A Microsoft signature for an inbox driver

"Signer Score - INBOX"

An Authenticode signature or WHQL signature for a Windows version earlier than the version that is specified by [LowerLogoVersion](#) for the device setup class of the driver

"Signer Score - Authenticode"

Unsigned driver

"Signer Score - Not digitally signed"

For more information about driver ranking, see [How Windows Ranks Drivers \(Windows Vista and Later\)](#).

# AllSigningEqual Group Policy

11/2/2020 • 3 minutes to read • [Edit Online](#)

If the **AllSigningEqual** Group Policy is disabled, Windows ranks drivers signed by a Windows signing authority (Microsoft signature) better than drivers that have one of the following:

- An [Authenticode](#) signature.
- A Microsoft signature for a Windows version earlier than the [LowerLogoVersion](#) value of the driver's [device setup class](#).

If the **AllSigningEqual** Group Policy is disabled, Windows selects a driver signed by a Windows signing authority over an Authenticode-signed driver, even if the Authenticode-signed driver is otherwise a better match to a device.

Signatures from a Windows signing authority are ranked equally and include the following signature types:

- Premium WHQL signatures and standard WHQL signatures
- Signatures for inbox drivers
- Windows Sustained Engineering (SE) signatures
- A WHQL signature for a Windows version that is the same or later than the [LowerLogoVersion](#) value of the driver's [device setup class](#).

A network administrator can change this behavior by enabling the **AllSigningEqual** Group Policy. This configures Windows to treat all Microsoft signature types and Authenticode signatures as equal with respect to rank when selecting the driver that is the best match to a device.

**Note** Starting with Windows 7, the **AllSigningEqual** Group Policy is enabled by default.

For example, consider the situation where a network administrator has to configure client computers on a network to install drivers as follows:

- A client computer should install a new driver only if the driver has an Authenticode signature that is issued by an Enterprise Certificate Authority (CA) that is created for the network. An enterprise might want to do this in order to ensure that all drivers that are installed on client computers are signed and that the only trusted signing authority other than Microsoft is the CA managed by the enterprise.
- For signed drivers, the signature score should not be used to determine the driver that has the best rank. Only the sum of the feature score and identifier score is used to compare driver ranks. For example, if the sum of the feature score and identifier score of a new driver is lower than the corresponding sum of an inbox driver, Windows installs the new driver.

To accomplish this, a network administrator:

- Adds an Enterprise CA certificate to the Trusted Publisher Store of the client computers.
- Enables the **AllSigningEqual** Group Policy on the client computers.

After the network administrator configures the client computers in this manner, the administrator can sign the driver that has the Enterprise CA certificate and distribute the driver to the client computers. In this configuration, Windows on client computers will install the new driver for a device instead of a Microsoft-signed driver if the sum of the feature score and the identifier score is lower than the corresponding sum for the Microsoft-signed driver.

Follow these steps to configure the AllSigningEqual Group Policy on Windows Vista and later versions of Windows:

1. On the Start menu, click **Run**.
2. Enter **GPEdit.msc** to execute the Group Policy editor, and click **OK**.
3. In the left pane of the Group Policy editor, click **Computer Configuration**.
4. On the **Administrative Templates** page, double-click **System**.
5. On the **System** page, double-click **Device Installation**.
6. Select **Treat all digitally signed drivers equally in the driver ranking and selection process** and then click **Properties**.
7. On the **Settings** tab, select **Enabled** (to enable the AllSigningEqual Group Policy) or **Disabled** (to disable the AllSigningEqual Group Policy).

To ensure that the settings are updated on the target system, do the following:

1. Create a desktop shortcut to *Cmd.exe*, right-click the *Cmd.exe* shortcut, and select **Run as administrator**.
2. From the Command Prompt window, run the Group Policy update utility, *GPUpdate.exe*.

This configuration change is made one time and applies to all subsequent driver installations on the computer until AllSigningEqual is reconfigured.

For more information about driver ranking, see [How Windows Ranks Drivers](#).

# Driver Packages

10/7/2019 • 2 minutes to read • [Edit Online](#)

A *driver package* consists of all the software components that you must supply in order for your device to be supported under Windows.

Installing a device or driver involves system-supplied and vendor-supplied components. The system provides generic installation software for all device classes. Vendors must supply one or more device-specific components within the driver package.

This section provides information to help you determine which components to supply within your driver package.

## In this section

- [Components of a Driver Package](#)
- [Installation Component Overview](#)
- [Supplying an INF File](#)

# Components of a Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following components are necessary to install and support a device on a Windows operating system:

## The device itself

If you plan to design and build a new device, follow industry hardware standards. When you follow these standards, you are more likely to have a streamlined development process as well as lower your support costs. Not only do test suites exist for such devices, but, in many cases, generic drivers exist for standard types. Therefore, you might not have to write a new driver.

## The driver package for the device

A driver package includes all the software components that you must supply to ensure that your device is supported with Windows. Typically, a driver package contains the following components:

- Driver files
- Installation files
- Other files

A brief description of each component of a driver package follows.

### Driver Files

The driver is the part of the package that provides the I/O interface for a device. Typically, a driver is a dynamic-link library (DLL) with the .sys file name extension. Long file names are allowed, except for *boot-start drivers*. When a device is installed, Windows copies the .sys file to the %SystemRoot%\system32\drivers directory.

The software that is required to support a particular device depends on the features of the device and the bus or port to which it connects. Microsoft ships drivers for many common devices and nearly all buses with the operating system. If your device can be serviced by one of these drivers, you might have to write only a device-specific *minidriver*. A minidriver handles device-specific features on behalf of a system-supplied driver. For some types of devices, even a minidriver is not necessary. For example, modems can typically be supported with just installation files.

### Installation Files

In addition to the device and the driver, a driver package also contains one or more of the following files that provide driver installation:

- A device setup information (INF) file

An INF file contains information that the [system-provided device installation components](#) use to install support for the device. Windows copies this file to the %SystemRoot%\inf directory when it installs the device. Every device must have an INF file.

For more information, see [Supplying an INF File](#).

- A driver [catalog \(.cat\)](#) file

A driver catalog file contains a cryptographic hash of each file in the driver package. Windows uses these hashes to verify that the package was not altered after it was published. To ensure that the catalog file is not altered, it should be [digitally signed](#).

For information about how to sign drivers, see [Signing Drivers for Public Release](#) and [Signing Drivers during](#)

[Development and Test.](#)

## **Other Files**

A driver package can also contain other files, such as a device installation application, a device icon, device property pages, and so forth. For more information, see the following topics:

[Providing Icons for a Device](#)

[Installing a Boot-Start Driver](#)

# Installation Component Overview

12/5/2018 • 2 minutes to read • [Edit Online](#)

The [Device Installation Overview](#) section provides details on how the Microsoft Windows operating system finds and installs devices and drivers, and on the components involved in such an installation.

To install a device or a driver, the operating system requires the following information at a minimum:

- The name and version number of each operating system on which the device or drivers are supported
- The device's setup class GUID and setup class
- Driver version information
- The names of the driver files together with their source and destination locations
- Device-specific information, including [hardware ID](#) and [compatible IDs](#)
- The name of a [catalog \(.cat\) file](#)
- Information about how and when to load the services that are provided by each driver (Windows 2000 and later versions of Windows)

All this information can be supplied in an INF file for the device. For most device and driver combinations, an INF file is the only installation component that is required. All devices and drivers require an INF file. For more information, see [Supplying an INF File](#).

If your device is involved in booting the system, installation requirements differ. See [Installing a Boot Driver](#).

# Supplying an INF File

11/2/2020 • 2 minutes to read • [Edit Online](#)

Every driver package must include an INF file, which the [device installation components](#) read when installing the device. An INF file is not an installation script. It is an ASCII or Unicode text file that provides device and driver information, including the driver files, registry entries, device IDs, [catalog files](#), and version information that is required to install the device or driver. The INF is used not only when the device or driver is first installed, but also when the user requests a driver update through Device Manager.

The exact contents and format of the INF file depend on the [device setup classes](#). [Summary of INF Sections](#) describes the information that is required in each type of INF. In general, per-manufacturer information is located in an [INF Models section](#). Entries in the **Models** section refer to [INF DDInstall sections](#) that contain model-specific details.

The [InfVerif](#) tool, which is provided in the `|tools` directory of the Microsoft Windows Driver Kit (WDK), checks the syntax and structure of all cross-class INF sections and directives, together with the class-specific extensions for all setup classes except for Printers.

Starting with Windows 2000, you can use a single INF file for installation on all versions of the Windows operating system. For more information, see [Creating INF Files for Multiple Platforms and Operating Systems](#). If your device will be sold in the international market, you should [create an international INF file](#). Depending on the localities involved, an international INF file might have to be a Unicode file instead of ASCII.

A good way to create an INF file for your driver is to modify one of the samples that the WDK provides. Most of the WDK sample drivers include INF files in the same directory as the sample driver.

For more information about INF files, see [Creating an INF File](#), the documentation for [InfVerif](#), the device-specific documentation in the WDK, and the INF files that are supplied with sample drivers for devices similar to yours.

# Overview of INF Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

Windows uses setup information (INF) files to install the following components for a device:

- One or more drivers that support the device.
- Device-specific configuration or settings to bring the device online.

An INF file is a text file that contains all the information that [device installation components](#) used to install a driver. Windows installs drivers using INF files. This information includes the following:

- Driver name and location
- Driver version information
- Registry information

You can use an `/INX` file to automatically create an INF file. An INX file is an INF file that contains string variables that represent version information. The Build utility and the [Stampinf](#) tool replace the string variables in INX files with text strings that represent a specific hardware architecture or framework version. For more information about INX files, see [Using INX Files to Create INF Files](#).

See [INF File Sections](#) and [INF File Directives](#) for a complete description of INF file format.

# Looking at an INF File

12/5/2018 • 2 minutes to read • [Edit Online](#)

The following example shows selected fragments from a system-supplied class installer's INF file to show how any INF file is made up of sections, each of which contains zero or more lines, some of which are entries that reference additional INF-writer-defined sections:

```
[Version]
Signature="$Windows NT$"
Class=Mouse
ClassGUID={4D36E96F-E325-11CE-BFC1-08002BE10318}
Provider=%Provider% ; defined later in Strings section
DriverVer=09/28/1999,5.00.2136.1

[DestinationDirs]
DefaultDestDir=12 ; DIRID_DRIVERS

; ... [ControlFlags] section omitted here

[Manufacturer]
%StdMfg% =StdMfg ; (Standard types)
%MSMfg% =MSMfg ; Microsoft
; ... %otherMfg% entries omitted here

[StdMfg] ; per-Manufacturer Models section
; Std serial mouse
*pnp0f0c.DeviceDesc= Ser_Inst,*PNP0F0C,SERENUM\PNP0F0C,SERIAL_MOUSE
; Std InPort mouse
%*pnp0f0d.DeviceDesc% = Inp_Inst,*PNP0F0D
; ... more StdMfg entries and following
; MSMfg and xxMfg Models sections omitted here

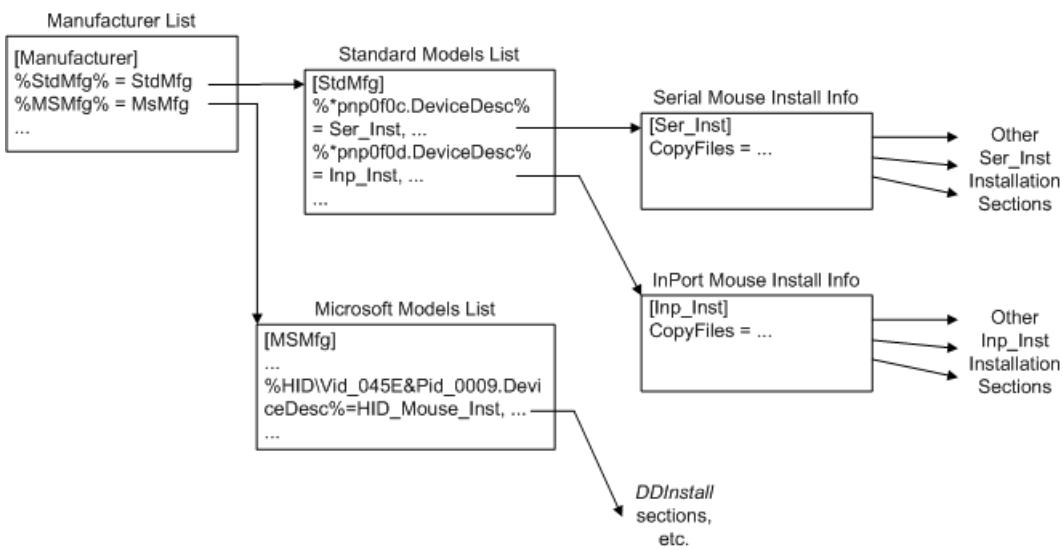
; per-Models DDInstall (Ser_Inst, Inp_Inst, etc.)
; sections also omitted here

[Strings]
; where INF %strkey% tokens are defined as user-visible (and
; possibly as locale-specific) strings.
Provider = "Microsoft"
; ...
StdMfg = "(Standard mouse types)"
MSMfg = "Microsoft"

; ...
*pnp0f0c.DeviceDesc = "Standard Serial Mouse"
*pnp0f0d.DeviceDesc = "InPort Adapter Mouse"
; ...
HID\Vid_045E&Pid_0009.DeviceDesc = "Microsoft USB Intellimouse"
; ...
```

A few sections within the previous INF file have system-defined names, such as **Version**, **DestinationDirs**, **Manufacturer**, and **Strings**. Some named sections like **Version**, **DestinationDirs**, and **Strings** have only simple entries. Others reference additional INF-writer-defined sections, as shown in the previous example of the **Manufacturer** section.

Note the implied hierarchy of related sections for mouse device driver installations starting with the **Manufacturer** section in the previous example. The following figure shows the hierarchy of some sections in the INF file.



Note the following about the implied hierarchy of an INF file:

- Each **%xxMfg%** entry in the **Manufacturer** section references a per-manufacturer *Models* section (StdMfg, MSMfg) elsewhere in the INF file.

The entries in the previous example use **%strkey%** tokens.

- Each *Models* section specifies some number of entries; in the example they are **%xxx.DeviceDesc%** tokens.

Each such **%xxx.DeviceDesc%** token references some number of per-models *DDInstall*/sections (Ser\_Inst and Inp\_Inst) for that manufacturer's product line, with each entry identifying a single device (\*PNP0F0C and \*PNP0F0D, hence the "DeviceDesc" shown here) or a set of compatible models of a device.

- Each such *DDInstall*-type *Xxx\_Inst* section, in turn, can have certain system-defined extensions appended and/or can contain directives that reference additional INF-writer-defined sections. For example, the full INF file that is shown as fragments in the previous example also has a **Ser\_Inst.Services** section, and its **Ser\_Inst** section has a **CopyFiles** directive that references a **Ser\_CopyFiles** section elsewhere in this INF file.

# Using a Universal INF File

11/2/2020 • 2 minutes to read • [Edit Online](#)

If you are building a [Windows Driver](#) package, you must use a universal INF file. If you are building a Windows Desktop Driver package, you don't have to use a universal INF file, but doing so is recommended because of the performance benefits.

A universal INF file uses a subset of the [INF syntax](#) that is available to a Windows driver. A universal INF file installs a driver and configures device hardware, but does not perform any other action, such as running a co-installer.

## Why is a universal INF file required on non-desktop editions of Windows?

Some editions of Windows, such as Windows 10X, use only a subset of the driver installation methods that are available on Windows 10 Desktop. An INF file for non-Desktop versions of Windows must perform only additive operations that do not depend on the runtime behavior of the system. An INF file with such restricted syntax is called a universal INF file.

A universal INF file installs predictably, with the same result each time. The results of the installation do not depend on the runtime behavior of the system. For example, co-installer references are not valid in a universal INF file because code in an additional DLL cannot be executed on an offline system.

As a result, a driver package with a universal INF file can be configured in advance and added to an offline system.

You can use the [InfVerif](#) tool to test if your driver's INF file is universal.

## Which INF sections are invalid in a universal INF file?

You can use any INF section in a universal INF file except for the following:

- [INF ClassInstall32 Section](#)
- [INF DDInstall.CoInstallers Section](#)
- [INF DDInstall.FactDef Section](#)
- [INF DDInstall.LogConfigOverride Section](#)

The [INF Manufacturer Section](#) is valid as long as the [TargetOSVersion](#) decoration does not contain a [ProductType](#) flag or [SuiteMask](#) flag.

The [INF DefaultInstall Section](#) is valid only if it has an architecture decoration, for example  
[DefaultInstall.NTAMD64].

## Which INF directives are invalid in a universal INF file?

You can use any INF directive in a universal INF file except for the following:

- [INF BitReg Directive](#)
- [INF DelFiles Directive](#)
- [INF DelProperty Directive](#)
- [INF DelReg Directive](#)

- [INF DelService Directive](#)
- [INF Ini2Reg Directive](#)
- [INF LogConfig Directive](#)
- [INF ProfileItems Directive](#)
- [INF RegisterDLLs Directive](#)
- [INF RenFiles Directive](#)
- [INF UnregisterDLLs Directive](#)
- [INF UpdateINIFields Directive](#)
- [INF UpdateINIS Directive](#)

The following directives are valid with some caveats:

- The [INF AddReg Directive](#) is valid if entries in the specified *add-registry-section* have a *reg-root* value of HKR, or in the following cases:
  - For registration of [Component Object Model \(COM\)](#) objects, a key may be written under:
    - HKCR
    - HKLM\SOFTWARE\Classes
  - For creation of [Hardware Media Foundation Transforms \(MFTs\)](#), a key may be written under:
    - HKLM\SOFTWARE\Microsoft\Windows Media Foundation
    - HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows Media Foundation
    - HKLM\SOFTWARE\WOW3232Node\Microsoft\Windows Media Foundation
- [INF CopyFiles Directive](#) is valid only if the [destination directory](#) is one of the following:
  - 11 (corresponds to %WINDIR%\System32)
  - 12 (corresponds to %WINDIR%\System32\Drivers)
  - 13 (corresponds to the directory under %WINDIR%\System32\DriverStore\FileRepository where the driver is stored)
 

**Note:** CopyFiles may not be used to rename a file for which [DestinationDirs](#) includes *dirid* 13. Also, *dirid* 13 is only valid on Windows 10 products for a limited subset of device installation scenarios. Please consult guidance and samples for your specific device class for more details.
  - 10,SysWOW64 (corresponds to %WINDIR%\SysWOW64)
  - 10,*vendor-specific subdirectory name*

**Note:** In Windows 10, version 1709, using *dirid* 10 with a vendor-specific subdirectory name is valid in a universal INF as measured using the [InfVerif](#) tool. In later releases, this value may not be supported. We recommend moving to *dirid* 13.

## See Also

- [Getting Started with Windows Drivers](#)
- [InfVerif](#)

# Using an Extension INF File

11/2/2020 • 8 minutes to read • [Edit Online](#)

Prior to Windows 10, Windows selected a single driver package to install for a given device. This resulted in large, complex driver packages that included code for all scenarios and configurations, and each minor update required an update to the entire driver package. Starting in Windows 10, you can split INF functionality into multiple components, each of which can be serviced independently.

To extend a driver package INF file's functionality, provide an extension INF in a separate driver package. An extension INF:

- Can be provided by a different company and updated independently from the base INF.
- Looks the same as a base INF, but can extend the base INF for customization or specialization.
- Enhances the value of the device, but is not necessary for the base driver to work.
- Must be a [universal INF file](#).

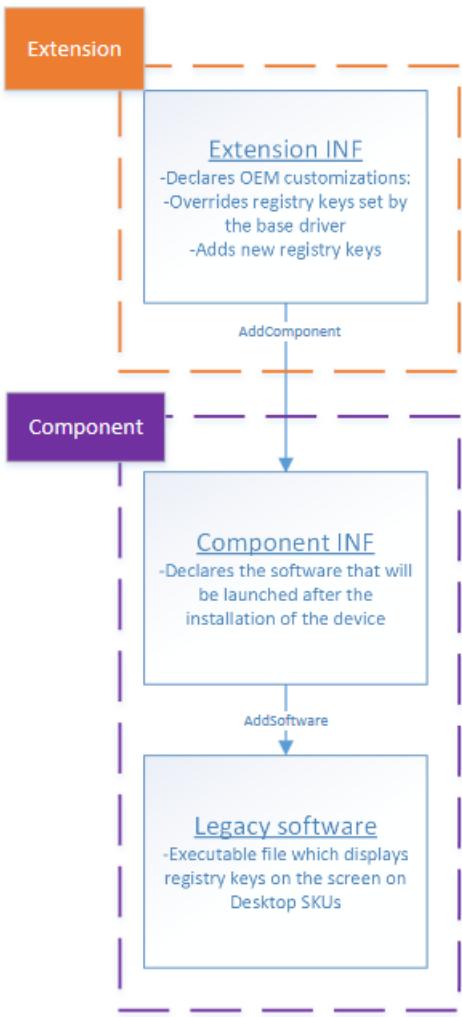
Every device must have one base INF, and can optionally have one or more extension INFs associated with it.

Typical scenarios where you might use an extension INF include:

- Modifying settings provided in a base INF, such as customizing the device friendly name or modifying a hardware configuration setting.
- Creating one or more software components by specifying the [INF AddComponent directive](#) and providing a [component INF file](#).

You can find sample code for these scenarios in the examples below. Also see [DCH-Compliant Driver Package Example](#), which describes how the [DCHU universal driver sample](#) uses extension INFs.

In the following diagram, two different companies have created separate driver packages for the same device, which are shown in the dotted lines. The first contains just an extension INF, and the second contains a component INF and a legacy software module. The diagram also shows how an extension INF can reference a component INF, which can in turn reference software modules to install.



## How extension INF and base INF work together

Settings in an extension INF are applied after settings in a base INF. As a result, if an extension INF and a base INF specify the same setting, the version in the extension INF is applied. Similarly, if the base INF changes, the extension INF remains and is applied over the new base INF.

It is helpful to include comments in the base INF describing which entries can be overridden, as well as applicable parameter value ranges and constraints.

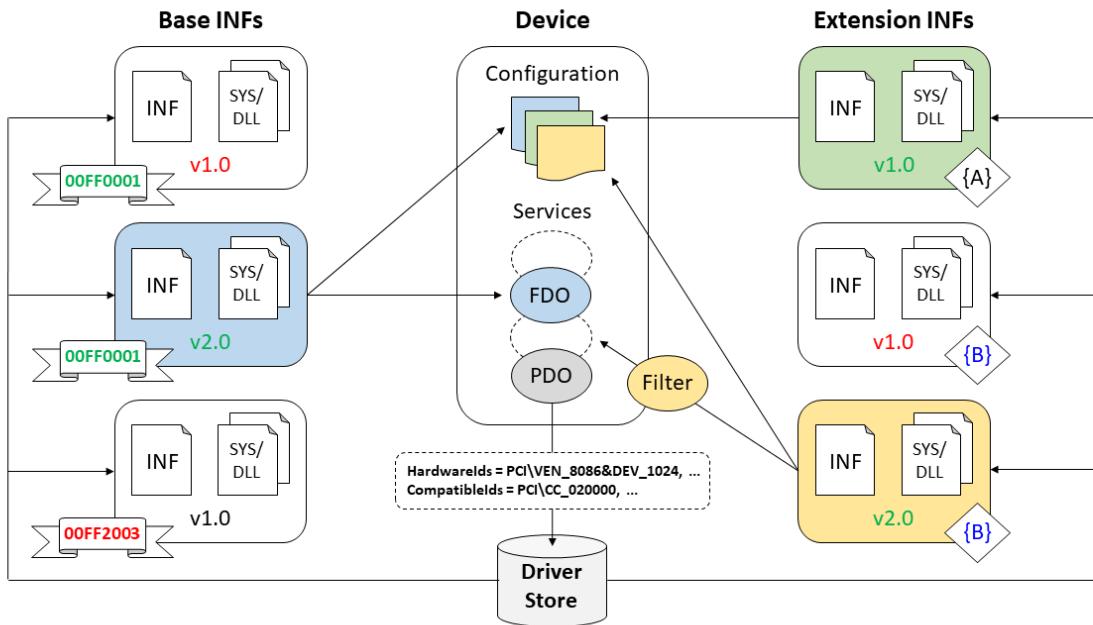
## Specifying ExtensionId

When you write an extension INF, you generate a special GUID called the **ExtensionId**, which is an entry in the INF's **[Version]** section.

The system identifies possible extension INFs for a specific device by matching the hardware ID and compatible IDs of the device to those specified in an extension INF in a **Models** section that applies to that system.

Among all possible extension INFs that specify the same **ExtensionId** value, the system selects only one to install and applies its settings over those of the base INF. The driver date and driver version specified in the INF are used, in that order, to choose the single INF between multiple extension INFs with the same **ExtensionId**.

To illustrate, consider the following scenario that includes a hypothetical device for which there are three extension INFs:



The **ExtensionId** values `{A}`, `{B}`, and `{C}` are shown in curly brackets, and each driver's **rank** is shown in the banner ribbons.

First, the system selects the driver with the most recent version and highest rank.

Next, the system processes the available extension INFs. Two have **ExtensionId** value `{B}`, and one has **ExtensionId** value `{A}`. From the first two, let's say that driver date is the same. The next tiebreaker is driver version, so the system selects the extension INF with v2.0.

The extension INF with the unique **ExtensionId** value is also selected. The system applies the base INF for the device, and then applies the two extension INFs for that device.

Note that extension INF files are always applied after the base INF, but that there is no determined order in which the extension INFs are applied.

## Creating an extension INF

Here are the entries you need to define an INF as an extension INF.

- Specify these values for **Class** and **ClassGuid** in the **Version** section. For more info on setup classes, see [System-Defined Device Setup Classes Available to Vendors](#).

```
[Version]
...
Class      = Extension
ClassGuid  = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}
```

- Provide an **ExtensionId** entry in the **[Version]** section. Generate a new GUID for the initial version of an extension INF, or reuse the last GUID for subsequent updates of the initial extension INF.

```
ExtensionId = {zzzzzzzz-zzzz-zzzz-zzzz-zzzzzzzzzz} ; replace with your own GUID
```

Note that an organization may only use an **ExtensionID** that it owns. For information on registering an Extension ID, see [Managing hardware submissions in the Windows Hardware Dev Center dashboard](#).

- If you are updating an extension INF, keep the **ExtensionId** the same and increment the version or date (or

both) specified by the [DriverVer](#) directive. For a given ExtensionId value, PnP selects the INF with the highest DriverVer.

#### NOTE

If your extension INF targets Windows 10 S, see [Windows 10 in S mode Driver Requirements](#) for information about driver installation on that version of Windows.

4. In the [INF Models section](#), specify one or more hardware and compatible IDs that match those of the target device. Note that these hardware and compatible IDs do not need to match those of the base INF. Typically, an extension INF lists a more specific hardware ID than the base INF, with the goal of further specializing a specific driver configuration. For example, the base INF might use a two-part PCI hardware ID, while the extension INF specifies a four-part PCI hardware ID, like the following:

```
[DeviceExtensions.NTamd64]  
%Device.ExtensionDesc% = DeviceExtension_Install, PCI\VEN_XXXX&DEV_XXXX&SUBSYS_XXXXXXXX&REV_XXXX
```

Alternatively, the extension INF might list the same hardware ID as the base INF, for instance if the device is already very narrowly targeted, or if the base INF already lists the most specific hardware ID.

In some cases, the extension INF might provide a less specific device ID, like a compatible ID, in order to customize a setting across a broader set of devices.

[CHID targeting](#) can be used if a four-part hardware ID is not possible or is not restrictive enough.

5. Do not define a service with `SPSVINST_ASSOCSERVICE`. However, an extension INF can define other services, such as a filter driver for the device. For more info about specifying services, see [INF AddService Directive](#).

In most cases, you'll submit an extension INF package to the Hardware Dev Center separately from the base driver package. For examples on how to package extension INFs, as well as links to sample code, see [DCH-Compliant Driver Package Example](#).

The driver validation and submission process is the same for extension INFs as for regular INFs. For more info, see [Windows HLK Getting Started](#).

## Uninstalling an extension driver

To uninstall an extension driver, use PnUtil's `delete` command with the `uninstall` flag.

To delete the driver package, use `pnputil /delete-driver /uninstall oem0.inf`. To force delete the driver package, use `pnputil /delete-driver /uninstall oem1.inf /force`.

This allows the extension driver to be uninstalled without removing the base driver.

## Example 1: Using an extension INF to set the device friendly name

In one common scenario, a device manufacturer (IHV) provides a base driver and a base INF, and then a system builder (OEM) provides an extension INF that supplements and in some cases overrides the configuration and settings of the base INF. The following snippet is a complete extension INF that shows how to set the device friendly name.

```

[Version]
Signature = "$WINDOWS NT$"
Class = Extension
ClassGuid = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}
Provider = %CONTOSO%
ExtensionId = {zzzzzzz-zzzz-zzzz-zzzz-zzzzzzzzzz} ; replace with your own GUID
DriverVer = 05/28/2013,1.0.0.0
CatalogFile = delta.cat

[Manufacturer]
%CONTOSO% = DeviceExtensions,NTamd64

[DeviceExtensions.NTamd64]
%Device.ExtensionDesc% = DeviceExtension_Install, PCI\VEN_XXXX&DEV_XXXX&SUBSYS_XXXXXXX&REV_XXXX

[DeviceExtension_Install]
; No changes

[DeviceExtension_Install.HW]
AddReg = FriendlyName_AddReg

[FriendlyName_AddReg]
HKR,,FriendlyName,, "New Device Friendly Name"

[Strings]
CONTOSO = "Contoso"
Device.ExtensionDesc = "Sample Device Extension"

```

## Example 2: Using an extension INF to install additional software

The following snippet is a complete extension INF that is included in the [Driver package installation toolkit for universal drivers](#). This example uses [INF AddComponent directive](#) to create components that install a service and an executable. For more info about what you can do in a component INF, see [Using a Component INF File](#).

To access this file online, see [osrfx2\\_DCHU\\_extension.inx](#).

```

;/**+
;
;Copyright (c) Microsoft Corporation. All rights reserved.
;
; THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
; KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR
; PURPOSE.
;
;Module Name:
;
;    osrfx2_DCHU_extension.INF
;
;Abstract:
;
;    Extension inf for the OSR FX2 Learning Kit
;
;--*/
;

[Version]
Signature = "$WINDOWS NT$"
Class = Extension
ClassGuid = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}
Provider = %ManufacturerName%
ExtensionId = {3846ad8c-dd27-433d-ab89-453654cd542a}
CatalogFile = osrfx2_DCHU_extension.cat
DriverVer = 05/16/2017,15.14.36.721

```

```

[Manufacturer]
%ManufacturerName% = Osrfx2Extension, NT$ARCH$

[Osrfx2Extension.NT$ARCH$]
%Osrfx2.ExtensionDesc% = Osrfx2Extension_Install, USB\Vid_045e&Pid_94aa&mi_00
%Osrfx2.ExtensionDesc% = Osrfx2Extension_Install, USB\Vid_0547&PID_1002

[Osrfx2Extension_Install.NT]
CopyInf=osrfx2_DCHU_usersvc.inf

[Osrfx2Extension_Install.NT.HW]
AddReg = Osrfx2Extension_AddReg
AddReg = Osrfx2Extension_COMAddReg

[Osrfx2Extension_AddReg]
HKR, OSR, "OperatingParams",, "-Extended"
HKR, OSR, "OperatingExceptions",, "x86"

; Add all registry keys to successfully register the
; In-Process ATL COM Server MSFT Sample.

[Osrfx2Extension_COMAddReg]
HKCR,AppID\ATLD11COMServer.DLL,AppID,, "{9DD18FED-55F6-4741-AF25-798B90C4AED5}"
HKCR,AppID\{9DD18FED-55F6-4741-AF25-798B90C4AED5},,"ATLD11COMServer"
HKCR,ATLD11COMServer.SimpleObject,,, "SimpleObject Class"
HKCR,ATLD11COMServer.SimpleObject\CLSID,,, "{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}"
HKCR,ATLD11COMServer.SimpleObject\CurVer,,, "ATLD11COMServer.SimpleObject.1"
HKCR,ATLD11COMServer.SimpleObject.1,,, "SimpleObject Class"
HKCR,ATLD11COMServer.SimpleObject.1\CLSID,,, "{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084},,"SimpleObject Class"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-
1A14BA118084}\InprocServer32,,%REG_EXPAND_SZ%, "%SystemRoot%\System32\ATLD11COMServer.dll"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}\InprocServer32,ThreadingModel,, "Apartment"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}\ProgID,,, "ATLD11COMServer.SimpleObject.1"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}\Programmable,,%FLG_ADDREG_KEYONLY%
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}\TypeLib,,, "{9B23EFED-A0C1-46B6-A903-218206447F3E}"
HKCR,CLSID\{92FCF37F-F6C7-4F8A-AA09-1A14BA118084}\VersionIndependentProgID,,, "ATLD11COMServer.SimpleObject"

[Osrfx2Extension_Install.NT.Components]
AddComponent = osrfx2_DCHU_component,,Osrfx2Extension_ComponentInstall
AddComponent = osrfx2_DCHU_usersvc,,Osrfx2Extension_ComponentInstall_UserSvc

[Osrfx2Extension_ComponentInstall]
ComponentIds=VID_045e&PID_94ab

[Osrfx2Extension_ComponentInstall_UserSvc]
ComponentIds=VID_045e&PID_94ac

[Strings]
ManufacturerName = "Contoso"
Osrfx2.ExtensionDesc = "Osrfx2 DCHU Device Extension"
REG_EXPAND_SZ = 0x00020000
FLG_ADDREG_KEYONLY = 0x00000010

```

For info on how to use an Extension INF to install a filter driver, see [Device filter driver ordering](#).

To improve extensibility, we recommend that an IHV put optional functionality in an [extension INF template](#).

## Backward compatibility

Any change to the base INF must be thoroughly tested to ensure that it does not break backward compatibility for existing extension INFs.

When managing a base INF, follow these best practices:

- Document parameter value ranges and constraints both in code comments and in a design document. Future changes must conform to the specified ranges.

- To support new ranges, add an optional parameter (no default value).

If the IHV puts all functionality in the base INF, here is one way to ensure that existing extension INFs continue to work:

1. The IHV provides a companion app that sets a registry flag to disable optional functionality by default.
2. The base driver checks if the flag is enabled before using optional functionality.
3. An OEM-supplied extension INF can opt in by setting the flag to enable.

## Submitting an extension INF for certification

For detailed information on how to work with Extension INFs on the Hardware Dev Center, please see [Working with Extension INFs in the Windows Hardware Dev Center Dashboard](#).

## Related topics

- [Working with Extension INFs in the Partner Center](#)
- [DCH-Compliant Driver Package Example](#)
- [Using a Universal INF File](#)
- [Getting Started with Windows Drivers](#)
- [Driver package installation toolkit for universal drivers](#)

# Using an Extension INF File Template

11/2/2020 • 2 minutes to read • [Edit Online](#)

This page describes how to use extension INF templates to improve extensibility.

An extension INF template is an extension INF with entries commented out that a device manufacturer (IHV) publishes in a separate driver package. Typically, the IHV separates out optional features from the base INF and puts them in an extension INF template. In the template, the IHV provides comments indicating entries that the system builder (OEM) can uncomment and change, as well as entries which can be uncommented but should not be changed. The OEM then uses the template as a starting point to create an extension INF.

To create an extension INF based on a template, follow the guidance in [Creating an extension INF](#) and refer to the examples at the bottom of that page.

To submit a new extension INF that is based on a template, use the [DUA process](#).

## NOTE

If an OEM uses the DUA process to modify an IHV-provided base INF, ownership of the base INF shifts to the OEM. Instead, the OEM should contact the IHV and request that appropriate extensibility be added to the base INF, or that the IHV provide an extension INF template.

An IHV might also use an extension INF template to add optional functionality to an already published driver package. By publishing a template rather than updating the base INF, the IHV helps ensure that existing extension INFs continue to work. The following sequence shows how this might work:

1. The IHV adds the new, optional value to an extension INF template, but not to the base INF.
2. The IHV adds code to the base driver to check for the existence of the new registry value:
  - If the updated base driver finds the new value, it uses the new functionality.
  - Otherwise, it uses the previous functionality.
3. The OEM uses the extension INF template to create a new extension INF that sets the new value.

If instead, the IHV decides to update the base INF, follow the guidelines described in [Using an Extension INF File](#).

## Related topics

- [Using an Extension INF file](#)
- [Working with Extension INFs in the Partner Center](#)

# Updating Device Firmware using Windows Update

11/2/2020 • 4 minutes to read • [Edit Online](#)

This topic describes how to update a removable or in-chassis device's firmware using the Windows Update (WU) service. For information about updating system firmware, see [Windows UEFI firmware update platform](#).

To do this, you'll provide an update mechanism, implemented as a device driver, that includes the firmware payload. If your device uses a vendor-supplied driver, you have the option of adding the firmware update logic and payload to your existing function driver, or providing a separate firmware update driver package. If your device uses a Microsoft-supplied driver, you must provide a separate firmware update driver package. In both cases, the firmware update driver package must be universal. For more info about universal drivers, see [Getting Started with Windows Drivers](#). The driver binary can use [KMDF](#), [UMDF 2](#) or the [Windows Driver Model](#).

Because WU cannot execute software, the firmware update driver must hand the firmware to Plug and Play (PnP) for installation.

## Firmware update driver actions

Typically, the firmware update driver is a lightweight device driver that does the following:

- At device start or in the driver's [\*EVT\\_WDF\\_DRIVER\\_DEVICE\\_ADD\*](#) callback function:
  1. Identify the device to which it is attached.
  2. Determine whether the driver has a firmware version that is more recent than the version on the firmware currently flashed on device hardware.
  3. If a firmware update is necessary, set an event timer to schedule the update.
  4. Otherwise, do nothing until the driver is started again.
- During system runtime:
  1. If an update is queued, wait for a set of conditions to be met.
  2. When conditions are met, perform the firmware update on the device.

## Firmware update driver contents

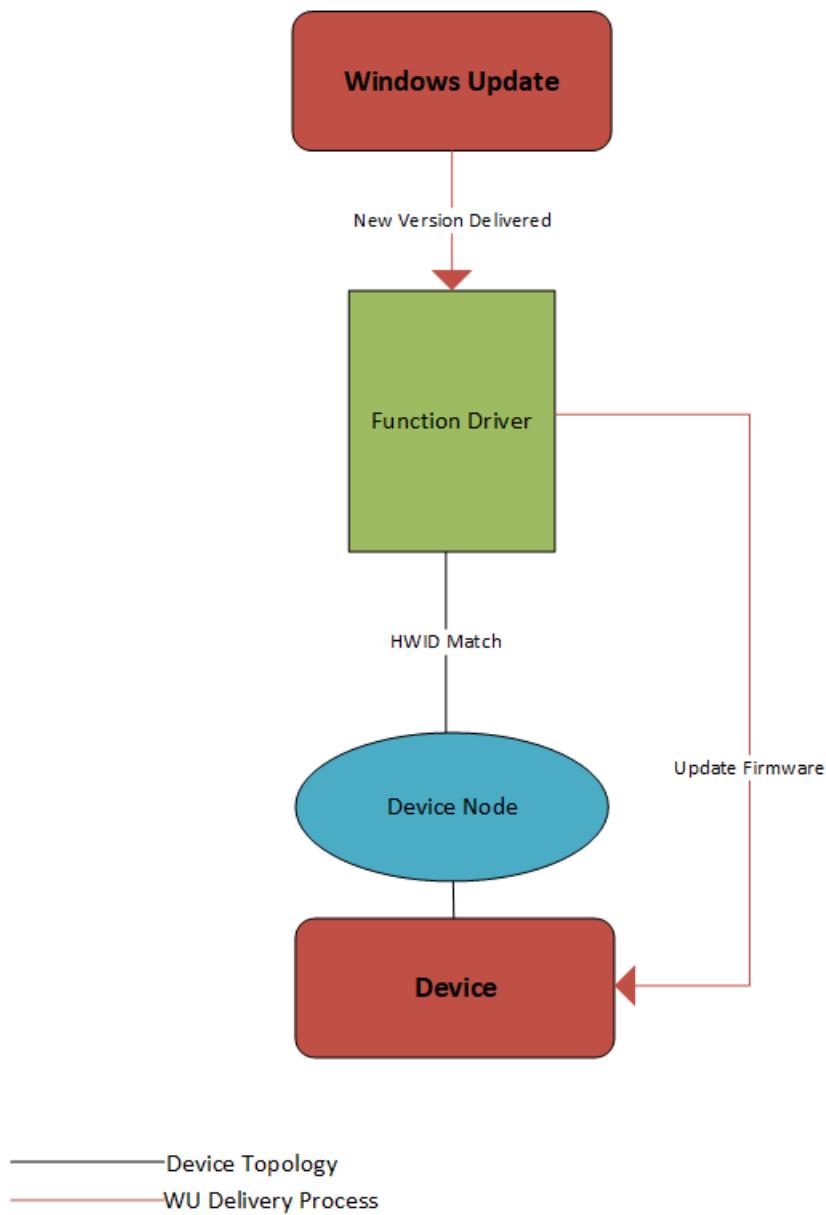
Typically, the firmware update driver package contains the following:

- [Universal Driver INF](#)
- Driver catalog
- Function driver (.sys or .dll)
- Firmware update payload binary

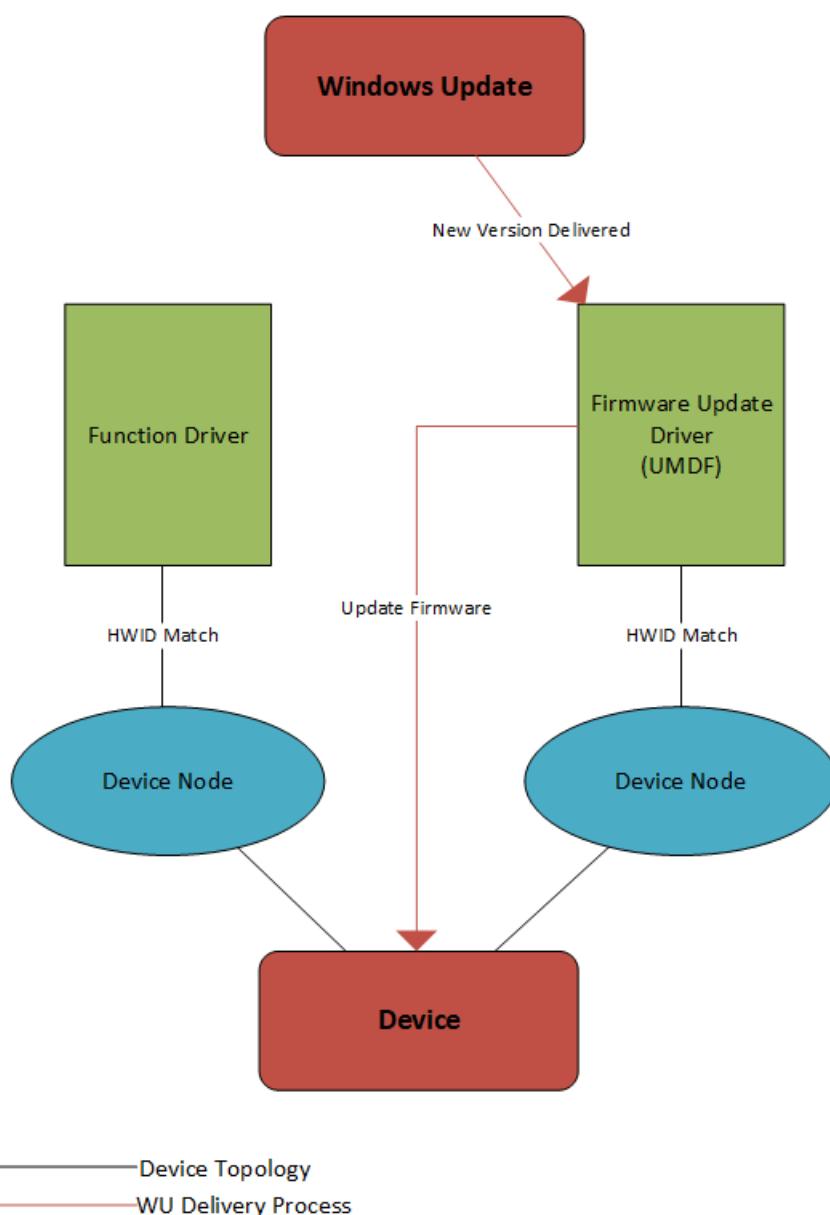
Submit your firmware update package as a separate driver submission.

## Adding firmware update logic to a vendor-supplied driver

The existing function driver can implement the firmware update mechanism, as shown in the following diagram:



Alternatively, if you want to update the function driver and the firmware update driver separately, create a second device node, on which you will install the firmware update driver. The following diagram shows how one device can have two separate device nodes:



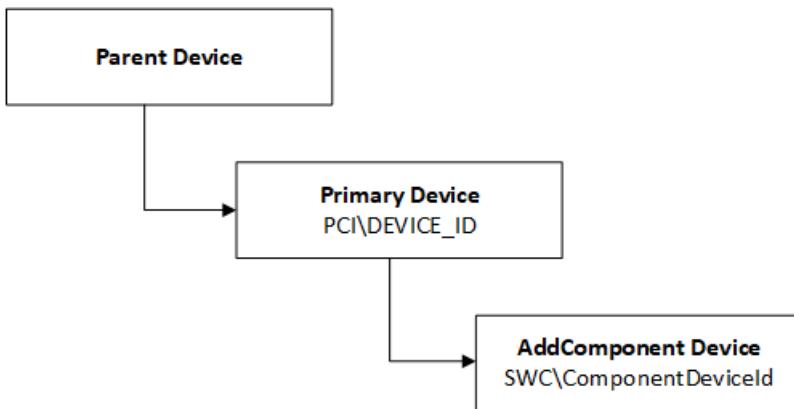
In this case, the function and firmware device nodes must have different hardware IDs in order to be targeted independently.

There are a couple ways to create a second device node. Certain device types have the ability to expose a second device node on one physical device, such as USB. You can use this functionality to create a device node targetable by WU, and install a firmware update driver on it. Many device types, however, do not allow a single physical device to enumerate more than one device node.

In this case, use an extension INF that specifies the [AddComponent](#) directive to create a device node that can be targeted by Windows Update and install the firmware update driver on it. The following snippet from an INF file shows how you can do this:

```
[Manufacturer]
%Contoso%=Standard,NTamd64
[Standard.NTamd64]
%DeviceName%=Device_Install, PCI\DEVICE_ID
[Device_Install.Components]
AddComponent=ComponentName,,AddComponentSection
[AddComponentSection]
ComponentIDs = ComponentDeviceId
```

In the above INF sample, `ComponentIDs = ComponentDeviceId` indicates that the child device will have a hardware ID of `SWC\ComponentDeviceId`. When installed, this INF creates the following device hierarchy:



For future firmware updates, update the INF and binary file containing the firmware payload.

## Adding firmware update logic to a Microsoft-supplied driver

To update firmware for devices that use a Microsoft-supplied driver, you need to create a second device node, as shown above.

### Best practices

- In your firmware update driver INF, specify [DIRID 13](#) to cause PnP to leave the files in the driver package in the DriverStore:

```

[Firmware_AddReg]
; Store location of firmware payload
HKR,,FirmwareFilename,,,"%13%\firmware_payload.bin"

```

PnP resolves this location when it installs the device. The driver can then open this registry key to determine the location of the payload.

- Firmware update drivers should specify the following INF entries:

```

Class=Firmware
ClassGuid={f2e7dd72-6468-4e36-b6f1-6488f42c1b52}

```

- To locate another device node, the firmware driver should walk the device tree relative to itself, not by enumerating all device nodes for a match. A user may have plugged in multiple instances of the device, and the firmware driver should only update the device with which it is associated. Typically, the device node to be located is the parent or sibling of the device node on which the firmware driver is installed. For example, in the diagram above with two device nodes, the firmware update driver can look for a sibling device to find the function driver. In the diagram immediately above, the firmware driver can look for the parent device to find the primary device with which it needs to communicate.
- The driver should be robust to multiple instances of the device being on the system, possibly with multiple different firmware versions. For example, there may be one instance of the device that has been connected and updated several times; a brand new device may then be plugged in which is several firmware versions old. This means that state (such as current version) must be stored against the device, and not in a global location.
- If there is an existing method to update the firmware (EXE or co-installer, for example), you can largely reuse the update code within a UMDF driver.

# Summary of INF Sections

11/2/2020 • 6 minutes to read • [Edit Online](#)

The following summarizes the system-defined sections that can be used in INF files. System-defined section names are case-insensitive. For example, **version**, **VERSION**, and **Version** are equally valid section-names within an INF file.

This section describes the INF file sections in the same order that they generally appear in most device INF files. However, these sections actually can be specified in any arbitrary order. Windows finds all sections within each INF file by section name, not by sequential order, whether system-defined or INF-writer-defined.

## **Version** Section

This is a required section for every INF file. For installation on Windows 2000 and later versions of Windows, this section must have a valid **Signature** entry.

## **SignatureAttributes** Section

This section of the INF defines a set of files to be embedded-signed as part of Hardware Certification. These additional signatures are required for devices with certain special needs. Examples are Protected Environment media playback, Early Launch Antimalware, and third party HAL extensions.

## **SourceDiskNames** Section

This section is required if the INF file has a corresponding **SourceDiskFiles** section. This section is required to install IHV/OEM-supplied devices and their drivers from distribution media included in packaged products. It is also required in such an INF file that installs either of the following:

- A co-installer DLL to supplement the operations of a system-supplied device class installer or co-installers (see also *DDInstall.CoInstallers* later in this list)
- A new class installer DLL to supplement the operations of the OS's device installer (see also *ClassInstall32*)

This section identifies the individual source distribution disks or CD-ROM discs for the installation. By contrast, the system-supplied INF files each specify a **LayoutFile** entry in their **Version** sections and provide at least one other INF file detailing the source distribution contents and layout of all software components to be installed.

## **SourceDiskFiles** Section

This section identifies the locations of files to be installed from the distribution media to the destinations on the target computer. An INF file that has this section must also have a **SourceDiskNames** section.

## **DestinationDirs** Section

Device/driver INF files have a **DestinationDirs** section to specify a default destination directory for INF-specified copies of the files supplied on the distribution media or listed in the INF layout files. This section is required unless the INF file installs a device, such as a modem or display monitor, that has no files except its INF to be installed with it.

## **ControlFlags** Section

This section controls whether an INF file is used only to transfer files from the distribution media.

Generally, most INF files for device drivers and for the system class installers have this section so they can exclude at least a subset of *Models* entries from the list of manually installable devices to be displayed to end-users. INF files that only install PnP devices suppress the display of all model-specific information.

## **Manufacturer** Section

This section is required in INF files for devices and their drivers.

The **Manufacturer** section of a system device class INF is sometimes called a "Table of Contents," because each of its entries references an INF-writer-defined *Models* section, which, in turn, references additional INF-writer-defined sections, such as a per-models-entry *DDInstall* section, *DDInstall.Services* section, and so forth.

#### **Models Section** (per **Manufacturer** entry)

This section is required to identify the devices for which the INF file installs drivers. It specifies a set of mappings between the generic name (string) for a device, the device ID, and the name of the *DDInstall* section, elsewhere in the INF file that contains the installation instructions for the device.

An INF file that installs one or more devices and drivers for a single provider would have only one *Models* section, but system INF files for device classes can have many INF-writer-defined *Models* sections.

#### **DDInstall Section** (per *Models* entry)

This section is required to actually install any devices that are listed in a *Models* section in the INF file, along with the drivers for each such device. A *DDInstall* section can be shared by more than one *Models* section.

#### **DDInstall.Services Section**

Starting with Microsoft Windows 2000, this section is required as an expansion of the *DDInstall* section for most kernel-mode device drivers. This includes any WDM drivers (exceptions are INF files for modems and display monitors). It controls how and when the services of a particular driver are started, its dependencies (if any) on underlying legacy, and so forth. This section also sets up event-logging services by a device driver if it supports event logging.

#### **DDInstall.HW Section**

This optional section adds device-specific (and typically, driver-independent) information to the registry or removes such information from the registry, possibly for a multifunction device or to install one or more PnP filter drivers.

#### **DDInstall.Coinstallers Section**

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

This optional section registers one or more device-specific co-installers supplied on the distribution media to supplement the operations of the system's device installer or of an existing device class installer.

A co-installer is an IHV/OEM-provided Win32 DLL that typically writes additional configuration information to the registry or performs other installation tasks that require dynamically generated, machine-specific information that is not available when the device's INF file is created. For more information, see [Writing a Co-installer](#).

#### **DDInstall.FactDef Section**

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

This section should be included in the INF file of any manually installed non-PnP device. It specifies the factory default hardware configuration settings, such as the bus-relative I/O ports, IRQ (if any), and so forth, for the card.

#### **DDInstall.LogConfigOverride Section**

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

This section is used to create an [override configuration](#), which overrides the hardware resource requirements that a Plug and Play device's bus driver reports.

#### **DDInstall.Interfaces Section**

If a driver exports the functionality of a device interface class, therefore creating a new instance of the interface class, such as kernel-streaming still-image capture or data decompression, its INF file can have this section.

#### **InterfaceInstall32 Section**

If a to-be-installed component, such as a new class driver, provides one or more new [device interface classes](#) to higher-level components, its INF file has this section. In effect, this section bootstraps a set of device interfaces for a new class by setting up whatever is needed to use the functionality that the interface class provides.

### **DefaultInstall Section**

An INF file's **DefaultInstall** section will be accessed if a user selects the "Install" menu item after selecting and holding (or right-clicking) on the INF file name.

### **DefaultInstall.Services Section**

This section is the same as the [INF DDInstall.Services section](#), and is used in association with an [INF DefaultInstall section](#).

### **Strings Section**

This section is required in every INF file to define each `%strkey%` token specified in the INF. By convention, the **Strings** section (or sections if the INF provides a set of locale-specific **Strings** sections) appears last in all system-supplied INF files for ease of maintenance and localization.

Some sections listed here, especially those with *Install*/in their names, can contain directives that reference additional INF-writer-defined sections. Each directive causes particular operations to be performed on the items listed under the appropriate type of INF-writer-defined section during the installation process.

The set of valid entries and directives for any particular section in the previous list is section-specific and shown in the formal syntax of the reference for each of these sections. Additionally, see [Summary of INF Directives](#) for a summary of the most commonly used directives.

Optional entries and directives within each such section are shown enclosed in unbolted brackets, as for example:

[Version] ... [Provider=%INF-creator%] ... The Provider entry in a [Version] section is optional in the sense that it is not a mandatory entry in every INF file.

# Summary of INF Directives

11/2/2020 • 4 minutes to read • [Edit Online](#)

The following list summarizes many (but not all) of the directives that can be used in INF files. INF directive names are case-insensitive. For example, **Addreg**, **addReg**, and **AddReg** are equally valid as directive specifications within an INF file.

This section lists the most commonly used directives first, together with their reciprocal or related directives. The most rarely used directives are toward the end of the list.

## [AddReg Directive](#)

This directive references one or more *add-registry-sections*, which are INF sections used to add or modify subkeys and value entries in the registry.

The particular INF section in which an **AddReg** (or **DelReg**) directive resides determines the default, relative registry location that will receive modifications specified in the referenced *add-registry-section* (or *delete-registry-section*). These default registry locations are typically device-specific or driver-specific subkeys somewhere under the HKEY\_LOCAL\_MACHINE registry tree. For more information, see [Registry Trees and Keys for Devices and Drivers](#).

Additional *add-registry-sections* can set up registry information for vendor-supplied co-installers, for system-defined device interfaces (such as kernel streaming interfaces) exported to higher level drivers, for new device interfaces exported by an installed component for a given class of devices, for driver services, or (rarely) for a new setup class of devices if the INF has a **ClassInstall32** section.

## [DelReg Directive](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This directive references one or more *del-registry-sections* used to remove obsolete subkeys and/or value entries from the registry. For example, such a section might appear in an INF that upgrades a previous installation.

## [CopyFiles Directive](#)

This directive references one or more *file-list-sections* specifying transfers of model/device-specific driver images and any other necessary files from the distribution media to the destination directory for each such file.

Alternatively, this directive can specify a single file to be copied from the distribution media to the default destination directory.

## [DelFiles Directive](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *file-list-sections* specifying files to be deleted from the target of the installation.

## [RenFiles Directive](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *file-list-sections* specifying INF-associated source files to be renamed on the destination.

## [AddService Directive](#)

This directive references at least a *service-install-section*, possibly with an additional *event-log-install-section*.

INF files for most kinds of devices (those that install drivers) have an INF-writer-defined *service-install-section* to specify any dependencies on system-supplied drivers or services, during which stage of the system initialization process the supplied drivers should be loaded, and so forth. Many INF files for device drivers also have an INF-writer-defined *event-log-install-section* that is referenced by the **AddService** directive to set up event logging by the device driver.

### **DelService Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive deletes a previously installed service.

### **AddInterface Directive**

This directive references an *add-interface-section* in which one or more **AddReg** directives are specified referencing sections that set up the registry entries for the device interfaces supported by this device/driver. Optionally, such an *add-interface-section* can reference one or more additional sections that specify delete-registry, file-transfer, file-delete, and/or file-rename operations.

### **BitReg Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *bit-registry-sections* specifying existing **REG\_BINARY**-type value entries in the registry for which particular bits in the values are to be modified.

### **LogConfig Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This directive references one or more *log-config-sections* that specify acceptable bus-relative and device-specific hardware configurations in an INF for devices that are detected (by PnP device enumerators) or manually installed. For example, INF files for non-PnP ISA, EISA, and MCA devices, which are manually installed, use this directive. (Also see [INF DDInstall.LogConfigOverride Section](#).)

### **UpdateInis Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *update-ini-sections* specifying parts of a supplied INI file to be read during installation and, possibly specifying line-by-line modifications to be made in that INI file.

### **UpdateIniFields Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *update-inifields-sections* specifying modifications to be made on fields within the lines of an INI file.

### **Ini2Reg Directive**

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

This rarely used directive references one or more *ini-to-registry-sections* specifying lines or sections of an INI file to be written into the registry.

The sections under which any of the directives in the previous list can be specified is system-determined. The basic form of each directive is shown in the formal syntax of the reference for each section, as for example:

```
[DDInstall] | [DDInstall.HW] | [DDInstall.CoInstallers] |
```

```
[ClassInstall32] | [ClassInstall32.ntx86] | [ClassInstall32.ntia64] | [ClassInstall32.ntamd64]
```

```
AddReg=add-registry-section[,add-registry-section] ...
```

The rest of this section describes the formal syntax and meaning for each system-defined named section, standard INF-writer-defined section, and directives that can be specified in an INF file.

# General Guidelines for INF Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

INF files have many common parts and follow a single set of syntax rules. However, they are also as different as the variety of devices that are supported by Microsoft Windows. When you write an INF file, refer to the following sources of information:

- This section and the [INF File 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](#).

- WDK tools for INF files.

For more information, see [Tools for INF Files](#). These tools are included in the \Tools subdirectory of the WDK.

- Sample INF files and INF files for similar devices.

The WDK includes INF files for its sample drivers. Look through the sample drivers to see whether there are INF files for devices similar to your device.

You can create or modify an INF file by 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.

INF files that ship with Windows 7 and earlier operating systems must have a file name of *xxxxxxxx.inf*, where "xxxxxxxx" does not exceed eight characters. The name of an INF file that ships separately from the operating system is not limited to eight characters.

Beginning with Windows 8, INF file names are not limited to eight characters, regardless if they ship with the operating system or not.

Do not arbitrarily modify the time stamps of your INF files, as a version control mechanism. Version control of INF files should be based on a date and version number that is specified in an [INF Version section](#).

## Best Practices for Naming and Versioning Your INF File

- INF names should be named in a way that reduces the chance of conflicts with INFs from other vendors. For example, the INF name could include in it, either as a prefix or a suffix, an abbreviation of your company name.
- If you have two different variants of the same driver package differing in aspects such as branding strings, settings, etc., those two driver packages should have unique names.
- Each time you update an INF or any file the INF references, you should update the date and version in the INF.

# General Syntax Rules for INF Files

11/2/2020 • 7 minutes to read • [Edit Online](#)

An INF file is a text file organized into named sections. Some sections have system-defined names and some sections have names determined by the writer of the INF file.

Each section contains section-specific entries that are interpreted by [device installation components](#) (class installers, co-installers, SetupAPI). Some entries begin with a predefined keyword. These entries are called *directives*.

Some INF file entries are basically pointers from one section to another, for a specific purpose. For example, an [INF AddReg directive](#) identifies a section that contains entries that instruct Windows to modify the registry. These entries sometimes include additional arguments (required or optional) for Windows to interpret during installation.

Other INF file entries do not point to other sections, but supply information that Windows uses during installation, such as file names, registry values, hardware configuration information, flags, and so on. For example, an [INF DriverVer directive](#) supplies driver version information.

When Windows begins an installation, it first looks for an [INF Version section](#) to verify the validity of the INF file and to determine where installation files are located. Then it starts the installation by finding an [INF Manufacturer section](#). This section contains directives to [INF Models sections](#), which in turn provide directives leading to various [INF DDInstall sections](#), based on the hardware ID of the device being installed.

The following syntax rules govern the required and optional contents of INF files, the format of section names by using string tokens, and line format, continuation, and comments.

## Case Sensitivity

- Section names, entries, and directives are case-insensitive. For example, `version`, `VERSION`, and `Version` are equally valid section name specifications within an INF file.

## Required and Optional Contents

- The set of required and optional sections, entries, and directives in any particular INF file depends on the type of device/driver or component (such as an application or device class installer DLL) to be installed.
- The set of sections, section-specific entries, and directives required to install any particular device and its drivers also depends somewhat on the corresponding class installer. For more information about how the system-supplied class installers handle device-type-specific INF files, see the device-type specific documentation in the WDK.
- Within syntax definitions, optional entries are delimited by *unbolded* brackets (`[.]`). On the other hand, *bold* brackets (`[,]`) are required elements of the entry in which they are contained. In the following example, the brackets around `Version` are required, while the brackets around `Class=class-name` indicate this entry is optional.

```
[Version]  
  
Signature="signature-name"  
[Class=class-name]  
...
```

## Section Names

- Sections can be specified in any order. Most INF files list sections in a particular order, by convention, but Windows finds sections by name, not by location within the INF file.
- Each section in an INF file begins with the section name enclosed in brackets ([ ]). The section name can be system-defined or INF-writer-defined.

For example, [Manufacturer] specifies the start of the system-named **Manufacturer** section, while [Std.Mfg] represents a particular INF-writer-defined *Models* section name.

A section name has a maximum length of 255 characters on Windows 2000 and later versions of Windows.

Each section ends at the beginning of a new [*section-name*] or at the end-of-file mark.

- If more than one section in an INF file has the same name, the system merges their entries and directives into a single section.
- Unless it is enclosed in double quotation marks characters ("), an INF-writer-defined section name must be a unique-to-the-INF unquoted string of explicitly visible characters, excluding certain characters with INF-specific meanings. In particular, an unquoted section name referenced by a section entry or directive cannot have leading or trailing spaces, a linefeed character, a return character, or any invisible control character, and it should not contain tabs. In addition, it cannot contain either of the bracket ([ ]) characters, a single percent (%) character, a semicolon (;), or any internal double quotation marks (") characters, and it cannot have a backslash (\) as its last character.

For example, Std.Mfg and Std\_Mfg are unique and valid section names when referenced by an INF file entry or directive, but Std;Mfg (with its internal semicolon) is invalid unless it is enclosed by double quotation marks (").

Specifying an INF-writer-defined section name as a "*quoted string*" overrides most of the restrictions that were previously described on characters in referenced section names. Such a delimited section name can contain almost any explicitly or implicitly visible characters except the closing bracket () as long as the corresponding section in the INF file matches this "*quoted string*" exactly.

For example, ";" Std Mfg " is a valid section-name reference if the corresponding section declaration in the INF file exactly matches the name inside the double quotation marks with respect to its space and semicolon characters as [;; Std Mfg ].

## Using String Tokens

- Many values in an INF file, including INF-writer-defined section names, can be expressed as string key tokens of the form %*strkey*%. In the **INF Strings** section of the INF file, each string key must be associated with a string value that consists of a sequence of explicitly visible characters. If necessary, the setup code converts the string value, into Unicode.

For more information about how to define %*strkey*% tokens and their respective values, see the description of the [INF Strings section](#).

## Line Format, Continuation, and Comments

- Each entry and directive in a section ends with a return or linefeed character. Therefore, the text editor used to create an INF file must not insert return or linefeed characters after some arbitrary, editor-determined number of characters.
- The backslash character (\) can be used as an explicit line continuator in an entry or directive. However, backslash characters are used also in path specifications. To ensure that a backslash character that appears in a path specification is not misinterpreted as a line continuator, use the following strategy:
  - For a directive that spans two lines, one of which is an entry that contains a backslash, use quotation marks to delimit the entry that contains the backslash.

```
CopyFiles = "SomeDirectory\"\  
,SomeFile
```

- Avoid using the backslash character in the manner shown in the following example. Windows ignores the first backslash and interprets the second backslash as a line continuator.

```
CopyFiles = SomeDirectory\\  
,SomeFile
```

- The following syntax is valid and is equivalent to

```
CopyFiles = "SomeDirectory\"",SomeFile ; comment .
```

```
CopyFiles = "SomeDirectory"\ ; comment  
,SomeFile
```

Because text after a semicolon is ignored, `CopyFiles = "SomeDirectory\" ; comment ,SomeFile` does not work.

- Comments begin with a semicolon (;) character. When parsing and interpreting an INF file, the system assumes that the following have no relevance to the installation process:
  - Any characters following a semicolon on the same line, unless the semicolon appears within a "*quoted string*" or `%strkey%` token
  - Any empty line that contains nothing except a linefeed or return character
- Commas separate the values supplied in section entries and directives.

An INF file entry or directive can omit an optional value in the middle of a list of values, but the commas must remain. INF files can omit trailing commas.

For example, consider the syntax for a **SourceDisk\Files** section entry:

```
filename= diskid[,subdir][,size]
```

An entry that omits the *subdir* value but supplies the *size* value must specify the comma delimiters for both values, as shown in the following example:

```
filename= diskid,,size
```

An entry in an INF file that omits the two optional values can have this format:

```
filename= diskid
```

- In order to include a percent (%) character in values supplied in section entries and directives, escape the percent character with another percent character.

For example, consider this statement in an *[add-registry-section]* section:

```
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\IoLogMsg.dll"
```

The registry value will be set with the following value:

```
%SystemRoot%\System32\IoLogMsg.dll
```

- In order to include a double quote ("") character in values supplied in section entries and directives, escape the double quote character with another double quote character. Note that the string must be within a "*quoted string*".

For example, consider this statement in an *[add-registry-section]* section:

*HKR,,Example,, "Display an ""example"" string"*

The registry value will be set with the following value:

*Display an "example" string*

#### **INF Size Limits**

- The maximum length, in characters, of an INF file field, before string substitution and including a terminating NULL character, is 4096.
- After string substitution, the maximum length, in characters, of an INF file string is 4096, which includes a terminating NULL character.
- However, be aware that Plug and Play (PnP) may impose a more restrictive limit for certain INF file fields that it recognizes or uses, such as device description, driver provider, and device manufacturer.

# Specifying the Source and Target Locations for Device Files

12/5/2018 • 2 minutes to read • [Edit Online](#)

When Windows 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**, **Include**, **Needs**, and **DestinationDirs**.

This section describes how Windows 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 INF Files](#)

[Target Media for INF Files](#)

[Copying INF Files](#)

# Source Media for INF Files

1/11/2019 • 3 minutes to read • [Edit Online](#)

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

## Source Media for INF Files

INF files for drivers specify where the files are located by using **SourceDiskNames** and **SourceDiskFiles** sections. If such an INF contains **Include** and **Needs** entries in the **DDInstall** section to reference other INF files and sections, those files and sections may specify additional possible source locations.

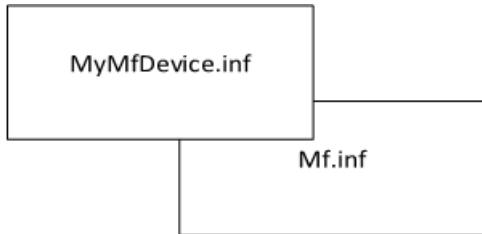
If an INF has **SourceDiskNames** and **SourceDiskFiles** sections and no **Include** entries, the **SourceDiskNames** and **SourceDiskFiles** sections must list all the source media and source files in the driver package except for the catalog and INF files.

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 and INF Files that Contain Include and Needs Entries

If an INF has **SourceDiskNames** and **SourceDiskFiles** sections and **Include** and **Needs** entries, Windows 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 INF files that are shown in the following figure:



This figure shows an INF file (*MyMfDevice.inf*) for a multifunction device. This INF file includes the system-supplied *Mf.inf* file. When Windows searches for source media from which to copy a file referenced in *MyMfDevice.inf*, it looks for a **SourceDiskFiles** section in *MyMfDevice.inf* and in any included INF files that reference the file to be copied. Windows searches *MyMfDevice.inf* first, but it does not guarantee the order in which it searches the included INF files.

Decorated **SourceDiskFiles** sections take precedence over undecorated sections, even if the *decorated INF section* is in an included file. For example, for the INF files that are shown in the previous figure, if *Mf.inf* contains a **[SourceDiskFiles.x86]** section and *MyMfDevice.inf* contains only an undecorated **[SourceDiskFiles]** section, Windows uses the decorated section from *Mf.inf* first when it installs on an x86 computer. Therefore, an INF that includes other INF files 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 Windows to search included INF files for file locations. In other words, a vendor INF that copies files should specify both a **SourceDiskNames** and a **SourceDiskFiles** section, those sections should be decorated with platform extensions, and those sections should contain information for all the files directly copied by the INF.

Vendor file names should be as vendor-specific as possible to avoid potential filename conflicts.

Please note that **Includes** entries can only be used to specify system-supplied INF files.

An INF file that uses a section in another INF file by using the **Include** and **Needs** entries might have to use an accompanying section to maintain consistency. For example, 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
```

Also note that each section that specifies a **Needs** directive must also specify an **Include** directive, even if the same INF file was specified in an **Include** directive elsewhere in the INF.

# Target Media for INF Files

12/5/2018 • 2 minutes to read • [Edit Online](#)

An INF file specifies the target location for device files that have 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 INF files, Windows searches the included INF files for target location information. However, the order in which Windows searches the included files is not predictable. Therefore, there is a risk that Windows 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.

# Copying INF Files

12/5/2018 • 2 minutes to read • [Edit Online](#)

It is sometimes necessary to copy INF files during device installation so that Windows 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 Windows can find these INF files without prompting the user each time it installs one of the device's functions.

To copy INF files, an INF file can use the [INF CopyINF directive](#).

Doing so 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 of the source medium from which device files are to be copied.
- Ensure compatibility with future versions of Windows.

Installation software must never copy INF files directly into a system's `%SystemRoot%/inf` directory. Copying INF files by using techniques not described in this section will invalidate a driver's digital signature.

# Creating INF Files for Multiple Platforms and Operating Systems

11/2/2020 • 2 minutes to read • [Edit Online](#)

By using system-defined platform extensions to [INF file sections and directives](#), you can create a single INF file for cross-platform installations. The extensions enable you to create *decorated* section names, which specify which sections and directives are relevant to each target platform and operating system. For example, you can create an INF file that installs a device only on x64-based systems, only on Itanium-based systems, only on x86-based systems, or on all systems that are supported by Windows 2000 and later versions of Windows.

The following table summarizes the system-supported platform extensions that can be added to the names of sections that support extensions.

PLATFORM EXTENSION	USE
.ntamd64	The section contains instructions for installing a device or set of device-compatible models on x64-based systems that are supported by Windows XP and later.
.ntia64	The section contains instructions for installing a device or set of device-compatible models on Itanium-based systems that are supported by Windows XP and later.
.ntx86	The section contains instructions for installing a device or set of device-compatible models on x86-based systems that are supported by Windows XP and later.
.ntarm	The section contains instructions for installing a device or set of device-compatible models on ARM-based systems that are supported by Windows 8 and later.
.ntarm64	The section contains instructions for installing a device or set of device-compatible models on ARM64-based systems that are supported by Windows 10 version 1709 and later.
.nt	In versions of Windows earlier than Windows Server 2003 SP1, the section contains instructions for installing a device or set of device-compatible models on all systems that are supported by the operating system. Starting with Windows Server 2003 SP1, the section contains instructions for installing a device or set of device-compatible models on x86-based systems that are supported by the operating system.

PLATFORM EXTENSION	USE
(no platform extension)	<p>In versions of Windows earlier than Windows Server 2003 SP1, the section contains instructions for installing a device or set of device-compatible models on all systems that are supported by the operating system.</p> <p>Starting with Windows Server 2003 SP1, the section contains instructions for installing a device or set of device-compatible models on x86-based systems that are supported by the operating system.</p>

**Important** Starting with Windows Server 2003 SP1, INF files must decorate entries in the [INF Models section](#) with `.ntia64`, `.ntarm`, `.ntarm64` or `.ntamd64` platform extensions to specify non-x86 target operating system versions. These platform extensions are not required in INF files for x86-based target operating system versions or non-PnP driver INF files (such as file system driver INF files for x64-based architectures).

**Tip** We highly recommend that you always decorate entries in the [INF Models section](#) with platform extensions for target operating systems of Windows XP and later versions of Windows. For x86-based hardware platforms, you should avoid the use of the `.nt` platform extension and use `.ntx86` instead.

## In this section

- [INF File Platform Extensions and x64-Based Systems](#)
- [INF File Platform Extensions and x86-Based Systems](#)
- [Cross-Platform INF Files](#)
- [Combining Platform Extensions with Other Section Name Extensions](#)

For an example of how to use INF file platform extensions to support cross-platform installations, see [Cross-Platform INF Files](#).

For information about how to use platform extensions in combination with section name extensions, see [Combining Platform Extensions With Other Section Name Extensions](#).

For information about how to specify target operating systems through platform extensions, see [Combining Platform Extensions with Operating System Versions](#).

For information about a sample INF file that can be used to install drivers in multiple operating system versions, see [Sample INF File for Device Installation on Multiple Versions of Windows](#).

# INF File Platform Extensions and x64-Based Systems

4/29/2020 • 3 minutes to read • [Edit Online](#)

## Platform Extensions and x64-Based Systems (Windows XP and later)

On Windows Server 2003 Service Pack 1 (SP1) and later, an **.ntamd64** platform extension is required on an **INF Models section**. An **.ntamd64** or **.nt** platform extension is optional on all other sections that support platform extensions.

For sections that support optional platform extensions, Windows selects which section to process, as follows:

1. Windows checks for a *section-name.ntamd64* section and, if one exists, processes it. Windows checks for the **.ntamd64** extension in the INF file that is being processed and in any included INF files (that is, any INF files that are included with **Include** entries).
2. If a *section-name.ntamd64* section does not exist, Windows checks for a *section-name.nt* section in the INF file or any included INF files. If one exists, Windows processes the *section-name.nt* section.
3. If a *section-name.nt* section does not exist, Windows processes a *section-name* section that does not include a platform extension.

## Testing Installation on x64-Based Systems (Windows Server 2003 SP1 and Later)

For testing purposes only, the requirement that an **INF Models section** name include an **.ntamd64** extension can be suppressed. To suppress this requirement, create the following registry value entry under the subkey **HKLM\Software\Microsoft\Windows\CurrentVersion\Setup** and set this value entry to one:

```
DisableDecoratedModelsRequirement:REG_DWORD
```

After setting the **DisableDecoratedModelsRequirement** value entry to one, restart the system and then install the device.

To restore the platform extension requirement, delete the **DisableDecoratedModelsRequirement** value entry or set it to zero, and then restart the system.

## Creating INF Files for x64-Based Systems (Windows XP and later)

In general, you cannot use a single INF file that differentiates between device installations that are based on the operating system version. For example, if the files or registry settings that support a device differ between versions of x64-based operating systems, you must create an operating system-specific INF file for each version.

However, if a device does not require an operating system-specific installation, you can create a single cross-operating system INF file for x64-based systems that run Windows XP and later.

Because Windows Server 2003 SP1 and later require an **.ntamd64** platform extension on an **INF Models section** name, but do not require this extension on other section names, the simplest approach to create and to maintain a cross-operating system INF file for x64-based systems is to include the **.ntamd64** extension only on the names of **Models** sections.

To create such a cross-operating system INF file, do the following:

1. Create a valid INF file that contains the generic entries that are required in all INF files, as described in [General Guidelines for INF Files](#).
2. Include an **INF Manufacturer** section that includes a *manufacturer-identifier* that specifies the **INF Models section** name for the device and that specifies the **.ntamd64** platform extension. For example, the

following **Manufacturer** section specifies an INF *Models* section name of "AbcModelSection" for an Abc device and the **.ntamd64** platform extension.

```
[Manufacturer]
; The manufacturer-identifier for the Abc device.
%ManufacturerName%=AbcModelSection,ntamd64
```

3. Include a *Models.ntamd64* section whose name matches the *Models* section name that is specified by the *manufacturer-identifier* in the **Manufacturer** section. For example, the following AbcModelSection.**.ntamd64** section for an Abc device includes a *device-description* that specifies an *install-section-name* of "AbcInstallSection."

```
[AbcModelSection.ntamd64]
%AbcDeviceName%=AbcInstallSection,Abc-hw-id
```

4. Include a *DD\Install* section whose name matches the *install-section-name* that is specified by the *Models* section. For example, the *device-description* in an AbcModelSection section specifies the following AbcInstallSection section for an Abc device.

```
[AbcInstallSection]
; Install section entries go here.
...
```

5. Include other device-specific sections that are required to install the device, but do not include an **.ntamd64** platform extension on names of these sections. For more information about INF file sections and directives, see [Summary of INF Sections](#) and [Summary of INF Directives](#).

For information about how to create a single cross-operating system INF for all platform types, see [Cross-Platform INF Files](#).

# INF File Platform Extensions and x86-Based Systems

4/29/2020 • 3 minutes to read • [Edit Online](#)

The following table summarizes Windows support for platform extensions for x86-based systems that run Windows 2000 and later versions of Windows.

PLATFORM EXTENSION	PLATFORM EXTENSION SUPPORT (WINDOWS 2000 AND LATER)
.ntamd64	Not supported.
.ntia64	Not supported.
.ntarm	Not supported.
.ntarm64	Not supported.
.ntx86	Optional on sections that support extensions.
.nt	Optional on sections that support extensions.
(No platform extension)	Supported by default on all sections.

For more information about how to use platform extensions with x86-based systems, see the following sections:

[Platform Extensions and x86-Based Systems \(Windows 2000 and Later\)](#)

[Creating INF Files for x86-Based Systems \(Windows 2000 and Later\)](#)

## Platform Extensions and x86-Based Systems

Windows XP and later versions of Windows support an optional .nt or .ntx86 platform extension on a *Models* section name and the names of other sections that support platform extensions.

Windows 2000 does not support platform extensions on an **INF Models section** name, but does support an optional .nt or .ntx86 platform extension on the names of other sections that support platform extensions.

For sections that support optional platform extensions, Windows selects which section to process, as follows:

1. Windows checks for a *section-name.ntx86* section and, if one exists, processes it. Windows checks for the .ntx86 extension in the INF file that is being processed and in any included INF files (that is, any INF files that are included with **Include** entries).
2. If a *section-name.ntx86* section does not exist, Windows checks for a *section-name.nt* section in the INF file or any included INF files. If one exists, Windows processes the *section-name.nt* section.
3. If a *section-name.nt* section does not exist, Windows processes a *section-name* section that does not include a platform extension.

## Creating INF Files for x86-Based Systems

In general, you cannot use a single INF file that differentiates between device installations that are based on the operating system version. For example, if the files or registry settings that support a device differ between versions of x86-based operating systems, you must create an operating system-specific INF file for each version.

However, if a device does not require operating system-specific installation, you can create a single cross-operating system INF file for x86-based systems that run Windows 2000 and later versions of Windows.

Because .nt and .ntx86 platform extensions are optional, the simplest approach to create and to maintain a cross-operating system INF file for x86-based systems is not to use platform extensions on section names.

To create a single cross-operating system INF file for x86-based systems that run Windows 2000 and later versions of Windows, do the following:

1. Create a valid INF file that contains the generic entries that are required in all INF files, as described in [General Guidelines for INF Files](#).
2. Include an **INF Manufacturer** section that includes a *manufacturer-identifier* that specifies the *Models* section name for the device, but does not specify an optional .nt or .ntx86 platform extension. For example, the following **Manufacturer** section specifies a *Models* section name of "AbcModelSection" for an Abc device.

```
[Manufacturer]
; The manufacturer-identifier for the Abc device.
%ManufacturerName%=AbcModelSection
```

3. Include a *Models* section whose name matches the *Models* section name that is specified by the *manufacturer-identifier* in the **Manufacturer** section. For example, the following AbcModelSection section for an Abc device includes a *device-description* that specifies an *install-section-name* of "AbcInstallSection."

```
[AbcModelSection]
%AbcDeviceName%=AbcInstallSection,Abc-hw-id
```

4. Include a *DD\Install* section whose name matches the *install-section-name* that is specified by the *Models* section. For example, the *device-description* in the AbcModelSection section specifies the following AbcInstallSection section for an Abc device. Windows processes this section to install the Abc device on x86-based systems that run Windows 2000 and later versions of Windows.

```
[AbcInstallSection]
; Install section entries go here.
...
```

5. Include other device-specific sections that are required to install the device, but do not include an .nt or .ntx86 platform extension on the names of these sections. Windows processes these sections to install the Abc device on x86-based systems that run Windows 2000 and later versions of Windows.

For more information about INF file sections and directives, see [Summary of INF Sections](#) and [Summary of INF Directives](#).

For information about how to create a single cross-platform INF file for all platform types, see [Cross-Platform INF Files](#).

# Cross-Platform INF Files

12/5/2018 • 4 minutes to read • [Edit Online](#)

The simplest strategy for cross-platform INF files is to create a separate INF file for each platform type because this approach is the easiest to create and maintain. For more information about how to create platform-specific INF files, see the following topics:

[Creating INF Files for x64-Based Systems \(Windows XP and later\)](#)

[Creating INF Files for x86-Based Systems \(Windows 2000 and Later\)](#)

You can create a single cross-operating system and cross-platform INF file for a device if the device does not have operating system-specific installation requirements. For example, if the files or registry settings that support a device differ between operating system versions for a given platform, you cannot, in general, create a single INF file for that platform type that is supported by all operating system versions.

To create a single cross-operating system and cross-platform INF file for Windows 2000 and later versions of Windows, the simplest approach is as follows:

- Use **.ntia64** platform extensions on the names of sections that are required to install components on Itanium-based systems, and use **.ntamd64** platform extensions on the names of sections that are required to install components on x64-based systems.
- Because **.nt** and **.ntx86** platform extensions are optional on all sections that support platform extensions, do not use an **.nt** or **.ntx86** platform extension on the names of sections that install components on x86-based systems.

To create a single cross-operating system and cross-platform INF file for Microsoft Windows 2000 and later versions of Windows, use the following process:

- Use **.ntia64** platform extensions on the names of sections that are required to install components on Itanium-based systems, and use **.ntamd64** platform extensions on the names of sections that are required to install components on x64-based systems.

To create a single cross-operating system and cross-platform INF file for a device that does not have operating system-specific requirements, supports all platform types, and that supports Windows 2000 and later versions of Windows, do the following:

1. Create a valid INF file that contains the generic entries that are required in all INF files, as described in [General Guidelines for INF Files](#).
2. Include an **INF Manufacturer** section that includes a *manufacturer-identifier* that specifies the *Models* section name for a device and a platform extension entry for each platform that the device supports. For example, the following Manufacturer section specifies a *Models* section name of "AbcModelSection" and the platform extensions **.ntia64** and **.ntamd64**. (Do not specify the **.ntx86** platform extension.)

```
[Manufacturer]
; The manufacturer-identifier for the Abc device.
%ManufacturerName%=AbcModelSection,ntia64,ntamd64
```

3. Include a *Models* section whose name does not include a platform extension. Starting with Windows 2000, the operating system processes this section for x86-based systems. For example, the following AbcModelSection section specifies an *install-section-name* of "AbcInstallSection" for an Abc device.

```
[AbcModelSection]
%AbcDeviceName%=AbcInstallSection,Abc-hw-id
```

4. Include a *Models.ntia64* section. Windows Server 2003 SP1 and later versions require a *Models.ntia64* section for Itanium-based systems. If a *Models.ntia64* section exists, Windows Server 2003 and Windows XP also use this section for Itanium-based systems. For example, the following AbcModelSection.ntia64 section specifies an *install-section-name* of "AbcInstallSection.ntia64" for an Abc device.

```
[AbcModelSection.ntia64]
%AbcDeviceName%=AbcInstallSection.ntia64,Abc-hw-id
```

5. Include a *Models.ntamd64* section. Windows Server 2003 SP1 and later versions require a *Models.ntamd64* section for x64-based systems. If a *Models.ntamd64* section exists, Windows Server 2003 and Windows XP also use this section for x64-based systems. For example, the following AbcModelSection.ntamd64 section specifies an *install-section-name* of "AbcInstallSection.ntamd64" for an Abc device.

```
AbcModelSectionName.ntamd64
%AbcDeviceName%=AbcInstallSection.ntamd64,Abc-hw-id
```

6. Include a *DD\Install* section whose name is the same as the *install-section-name* that is specified by the *Models* section that does not include a platform extension. For example, the AbcModelSection section specifies the following AbcInstallSection section. Windows processes this section to install the Abc device on x86-based systems that run Windows 2000 or later versions of Windows.

```
[AbcInstallSection]
; Install section entries go here.
...
```

7. Include a *DD\Install.ntia64* section whose name is the same as the *install-section-name* that is specified by the *Models.ntia64* section. For example, the AbcModelSection.ntia64 section specifies the following AbcInstallSection.ntia64 section. Windows processes this section to install the Abc device on Itanium-based systems that run Windows XP or later versions of Windows.

```
[AbcInstallSection.ntia64]
; Install section entries go here.
...
```

8. Include a *DD\Install.ntamd64* section whose name is the same as the *install-section-name* that is specified by the *Models.ntamd64* section. For example, the AbcModelSection.ntamd64 section specifies the following AbcInstallSection.ntamd64 section. Windows processes this section to install the Abc device on x64-based systems that run Windows XP or later versions of Windows.

```
[AbcInstallSection.ntamd64]
; Install section entries go here.
...
```

9. Include additional device-specific sections that are required for an x86-based installation. Do not include an .ntx86 platform extension on the names of these sections. Windows processes these sections by default to install the device on x86-based systems that run Windows 2000 or later versions of Windows.

10. Include additional device-specific sections that are required for Itanium-based systems that run Windows

XP or later versions of Windows. Include the **.ntia64** extension on these section names.

11. Include additional device-specific sections that are required for x64-based systems that run Windows XP or later versions of Windows. Include the **.ntamd64** extension on these section names.

For more information about INF file sections and directives, see [Summary of INF Sections](#) and [Summary of INF Directives](#).

# Combining Platform Extensions with Other Section Name Extensions

11/2/2020 • 2 minutes to read • [Edit Online](#)

An INF file that contains 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 *DDInstall.Interfaces* sections. In cross-operating system and cross-platform INF files, you should specify the appropriate platform extension immediately after the INF-writer-defined section name (for example, *install-section-name.ntx86.HW*).

If a cross-operating system INF file contains decorated *install-section-name.nt*, *install-section-name.ntx86*, *install-section-name.ntia64*, or *install-section-name.ntamd64* 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 must also have a parallel *install-section-name.HW* section (if the device or driver requires a .HW section for Windows 2000 and later versions of Windows).

The topics in the [INF File Sections and Directives](#) section show the **ntxxx.HW** extensions as part of the formal syntax statements in the appropriate section references, such as in the following example:

```
[install-section-name.HW] |
[install-section-name.nt.HW] |
[install-section-name.ntx86.HW] |
[install-section-name.ntia64.HW] |
[install-section-name.ntamd64.HW]
```

Such a formal syntax statement indicates that these extensions are valid alternatives to the basic section. This type of statement does *not* indicate that any INF file that has an *install-section-name.nt.HW* section must also include every other platform-specific *install-section-name.ntxxx.HW* section. You can use any subset of these extensions to specify the decorated sections that are required for a particular cross-platform 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.

# Combining Platform Extensions with Operating System Versions

12/5/2018 • 5 minutes to read • [Edit Online](#)

Within the **INF Manufacturer section** of an INF file, you can supply **INF Models sections** that are specific to various versions of the Windows operating system. These version-specific *Models* sections are identified by using the *TargetOSVersion* decoration.

Within the same INF file, different **INF Models sections** can be specified for different versions of the operating system. The specified versions indicate target operating system versions with which the INF *Models* sections will be used. If no versions are specified, Windows uses a *Models* section without the *TargetOSVersion* decoration for all versions of all operating systems.

## **TargetOSVersion** decoration format

The following example shows the correct format of the *TargetOSVersion* decoration for Windows XP through Windows 10, version 1511:

```
nt[Architecture][.[OSMajorVersion].[OSMinorVersion].[.ProductType].[.SuiteMask]]]
```

Starting with Windows 10, version 1607 (Build 14310 and later), the correct format of the *TargetOSVersion* decoration includes *BuildNumber*:

```
nt[Architecture][.[OSMajorVersion].[.OSMinorVersion].[.ProductType].[.SuiteMask].[.BuildNumber]]]
```

Each field is defined as follows:

**nt**

Specifies that the target operating system is NT-based. Windows 2000 and later versions of Windows are all NT-based.

**Architecture**

Identifies the hardware platform. If specified, this must be **x86**, **ia64**, or **amd64**. If not specified, the associated INF *Models* section can be used with any hardware platform.

**OSMajorVersion**

A number that represents the major version number for the operating system. The following table defines the major version for the Windows operating systems.

WINDOWS VERSION	MAJOR VERSION
Windows 10	10
Windows Server 2012 R2	6
Windows 8.1	6
Windows Server 2012	6

WINDOWS VERSION	MAJOR VERSION
Windows 8	6
Windows Server 2008 R2	6
Windows 7	6
Windows Server 2008	6
Windows Vista	6
Windows Server 2003	5
Windows XP	5

#### *OSMinorVersion*

A number that represents the minor version number for the operating system. The following table defines the minor version for the Windows operating systems.

WINDOWS VERSION	MINOR VERSION
Windows 10	0
Windows Server 2012 R2	3
Windows 8.1	3
Windows Server 2012	2
Windows 8	2
Windows Server 2008 R2	1
Windows 7	1
Windows Server 2008	0
Windows Vista	0
Windows Server 2003	2

WINDOWS VERSION	MINOR VERSION
Windows XP	1

#### *ProductType*

A number that represents *one* of the VER\_NT\_XXXX flags defined in *Winnt.h*, such as the following:

0x00000001 (VER\_NT\_WORKSTATION)

0x00000002 (VER\_NT\_DOMAIN\_CONTROLLER)

0x00000003 (VER\_NT\_SERVER)

If a product type is specified, the INF file will be used only if the operating system matches the specified product type. If the INF file supports multiple product types for a single operating system version, multiple *TargetOSVersion* entries are required.

#### *SuiteMask*

A number that represents a *combination of* one or more of the VER\_SUITE\_XXXX flags defined in *Winnt.h*. These flags include the following:

0x00000001 (VER\_SUITE\_SMALLBUSINESS)

0x00000002 (VER\_SUITE\_ENTERPRISE)

0x00000004 (VER\_SUITE\_BACKOFFICE)

0x00000008 (VER\_SUITE\_COMMUNICATIONS)

0x00000010 (VER\_SUITE\_TERMINAL)

0x00000020 (VER\_SUITE\_SMALLBUSINESS\_RESTRICTED)

0x00000040 (VER\_SUITE\_EMBEDDEDNT)

0x00000080 (VER\_SUITE\_DATACENTER)

0x00000100 (VER\_SUITE\_SINGLEUSERTS)

0x00000200 (VER\_SUITE\_PERSONAL)

0x00000400 (VER\_SUITE\_SERVERAPPLIANCE)

If one or more suite mask values are specified, the INF file will be used only if the operating system matches all the specified product suites. If the INF file supports multiple product suite combinations for a single operating system version, multiple *TargetOSVersion* entries are required.

#### *BuildNumber*

Specifies the minimum OS build number of the Windows 10 release to which the section applies, starting with build 14310 or later.

The build number is assumed to be relative to some specific OS major/minor version only, and may be reset for some future OS major/minor version.

Any build number specified by the *TargetOSVersion* decoration is evaluated only when the OS major/minor version of the *TargetOSVersion* matches the current OS (or AltPlatformInfo) version exactly. If the current OS version is greater than the OS version specified by the *TargetOSVersion* decoration (OSMajorVersion, OSMinorVersion), the section is considered applicable regardless of the build number specified. Likewise, if the current OS version is less than the OS version specified by the *TargetOSVersion* decoration, the section is not applicable.

If build number is supplied, the OS version and BuildNumber of the *TargetOSVersion* decoration must both be greater than the OS version and build number of the Windows 10 build 14310 where this decoration was first introduced. Earlier versions of the operating system without these changes (for example, Windows 10 build

10240) will not parse unknown decorations, so an attempt to target these earlier builds will actually prevent that OS from considering the decoration valid at all.

## How Windows processes *TargetOSVersion* decorations

When you install a device or driver on a host operating system, Windows follows these steps to process the **INF Models sections** within an INF file:

1. If one or more **INF Models sections** have the *TargetOS* decoration, Windows selects the **INF Models** section that is closest to the attributes for the host operating system.

For example, if an **INF Models** section has a *TargetOS* decoration of **ntx86.5.1**, Windows selects that section if the host operating system is running Windows XP or later version of Windows on an x86-based system.

Similarly, if an **INF Models section** has a *TargetOS* decoration of **nt.6.0**, Windows selects that section if the host operating system is Windows Vista or later version of Windows on any supported hardware platform.

If an **INF Models** section has a *TargetOS* decoration of **nt.10.0...14393**, Windows selects that section if the host operating system is running a Windows 10 build equal to or greater than 14393 on any supported hardware platform.

2. If none of the **INF Models sections** have a *TargetOS* decoration that matches the host operating system, Windows selects the *Models* section that has either a matching platform extension or no platform extension.

For example, if an **INF Models** section has a platform extension of **ntx86**, Windows selects that section if the host operating system is Microsoft Windows 2000 or later version of Windows on an x86-based system.

Similarly, if an **INF Models sections** has no platform extension, Windows selects that section if the host operating system is Windows 2000 or later version of Windows on any supported hardware platform.

3. If Windows cannot find a matching **INF Models section**, it will not use the INF file to install the device or driver.

## Sample **INF Models** sections with *TargetOSVersion* decorations

The following topics provide samples of how to decorate platform extensions for target operating systems within an **INF Models section**:

[Sample \*\*INF Models\*\* Sections for One or More Target Operating Systems](#)

[Sample \*\*INF Models\*\* Sections for Only One Target Operating System](#)

[Sample INF File for Device Installation on Multiple Versions of Windows](#)

# Sample INF Models Sections for One or More Target Operating Systems

12/5/2018 • 2 minutes to read • [Edit Online](#)

This topic shows a sample INF file that installs a driver package on various operating systems and platforms. This INF file has the following INF sections and directives:

- A decorated **INF Manufacturer section** with various **INF Models sections** for device installation on x86-based systems that are running:
  - Microsoft Windows 2000
  - Windows XP
  - Windows Vista or later versions of Windows
- A decorated **INF Manufacturer section** with various **INF Models sections** for device installation on x86- and AMD64-based systems that are running Windows Vista or later versions of Windows.
- An **INF DDInstall** and **DDInstall.Services** that creates the service and registry entries on the target x86- and AMD64-based systems.
- **INF CopyFiles directives** that copies platform-specific files to the target x86- and AMD64-based systems.

```
[Version]
Signature      = "$Windows NT$"
Class          = Net
ClassGUID     = {4d36e972-e325-11ce-bfc1-08002be10318}
Provider       = %Msft%
DriverVer     = 10/01/2002,6.0.5019.0

[Manufacturer]
%Msft%         = Msft, NTx86.6.0, NTamd64.6.0

; -----
; Models Section
; -----


;For Windows 2000 and Windows XP

[Msft]
%NetVMini.DeviceDesc%  = NetVMini.NTx86.ndi, root\NetVMini ; Root enumerated
%NetVMini.DeviceDesc%  = NetVMini.NTx86.ndi, {b85b7c50-6a01-11d2-b841-00c04fad5171}\NetVMini ; Toaster Bus
enumerated

;For Windows Vista and later

[Msft.NTx86.6.0]
%NetVMini.DeviceDesc%  = NetVMini.NTx86.ndi, root\NetVMini ; Root enumerated
%NetVMini.DeviceDesc%  = NetVMini.NTx86.ndi, {b85b7c50-6a01-11d2-b841-00c04fad5171}\NetVMini ; Toaster Bus
enumerated

[Msft.NTamd64.6.0]
%NetVMini.DeviceDesc%  = NetVMini.NTamd64.ndi, root\NetVMini ; Root enumerated
%NetVMini.DeviceDesc%  = NetVMini.NTamd64.ndi, {b85b7c50-6a01-11d2-b841-00c04fad5171}\NetVMini ; Toaster Bus
enumerated

; -----
; NT x86 DDInstall and DDInstall.Service Sections
```

```

; -----
[NetVMini.NTx86.ndi]
Characteristics = 0x1 ; NCF_VIRTUAL
AddReg          = NetVMini.Reg
CopyFiles       = NetVMini.NTx86.CopyFiles

[NetVMini.NTx86.ndi.Services]
AddService      = NetVMini, 2, NetVMini.NTx86.Service

[NetVMini.NTx86.Service]
DisplayName     = %NetVMini.Service.DispName%
ServiceType    = 1 ;SERVICE_KERNEL_DRIVER
StartType      = 3 ;SERVICE_DEMAND_START
ErrorControl   = 1 ;SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\netvmini_32.sys
LoadOrderGroup = NDIS
AddReg          = TextModeFlags.Reg

; -----
; NT x64 DDInstall and DDInstall.Service Sections
; -----
[NetVMini.NTamd64.ndi]
Characteristics = 0x1 ; NCF_VIRTUAL
AddReg          = NetVMini.Reg
CopyFiles       = NetVMini.NTamd64.CopyFiles

[NetVMini.NTamd64.ndi.Services]
AddService      = NetVMini, 2, NetVMini.NTamd64.Service

[NetVMini.NTamd64.Service]
DisplayName     = %NetVMini.Service.DispName%
ServiceType    = 1 ;SERVICE_KERNEL_DRIVER
StartType      = 3 ;SERVICE_DEMAND_START
ErrorControl   = 1 ;SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\netvmini_64.sys
LoadOrderGroup = NDIS
AddReg          = TextModeFlags.Reg

; -----
; Registry Section
; -----
[NetVMini.Reg]
HKR, , BusNumber, 0, "0"
HKR, Ndi, Service, 0, "NetVMini"
HKR, Ndi\Interfaces, UpperRange, 0, "ndis5"
HKR, Ndi\Interfaces, LowerRange, 0, "Ethernet"

[TextModeFlags.Reg]
HKR, , TextModeFlags, 0x00010001, 0x0001

; -----
; Disk/FIle Sections
; -----
[SourceDisksNames]
1 = %DiskId1%"",,

[SourceDisksFiles]
netvmini_32.sys = 1,,
netvmini_64.sys = 1,,

[DestinationDirs]
DefaultDestDir      = 12
NetVMini.NTx86.CopyFiles = 12
NetVMini.NTamd64.CopyFiles = 12

[NetVMini.NTx86.CopyFiles]
netvmini_32.sys,,,2

[NetVMini.NTamd64.CopyFiles]

```

```
netvmini_64.sys,,,2

; -----
; Strings Section
; -----
[Strings]
Msft           = "Microsoft"

NetVMini.DeviceDesc      = "Microsoft Virtual Ethernet Adapter"
NetVMini.Service.DispName = "Microsoft Virtual Miniport"
DiskId1                = "Microsoft Virtual Miniport Device Installation Disk #1"
```

The sample INF file (*MultiOS.inf*), which is included in the Windows Driver Kit (WDK), shows how a single INF file can be used to install a device on multiple versions of Windows. This file is in the *src\print\infs\MultiOS* directory of the WDK.

# Sample INF Models Sections for Only One Target Operating System

12/5/2018 • 2 minutes to read • [Edit Online](#)

It is possible to use decorated **INF Models sections** to limit the range of applicable target operating systems.

The following example shows an **INF Manufacturer section** with various **INF Models sections** that will prevent Windows from installing a device on x86-based systems not running Windows Vista.

```
[Manufacturer]
%Msft% = Msft, NTx86.6.0, NT.6.1

;For Windows Vista only

[Msft.NTx86.6.0]
%NetVMini.DeviceDesc%      = NetVMini.ndi, root\NetVMini ; Root enumerated
%NetVMini.DeviceDesc%      = NetVMini.ndi, {b85b7c50-6a01-11d2-b841-00c04fad5171}\NetVMini ; Toaster Bus
enumerated

;For Windows 7 and later

[Msft.NT.6.1]
```

In this example, the **INF Manufacturer section** has the following **INF Models sections**:

- A complete INF *Models* section for Windows Vista on x86-based systems that include device descriptions and hardware identifiers (IDs). Windows will select and use this section when it installs the device on x86-based systems that are running Windows Vista.
- An empty INF *Models* section for Windows 7 and later versions of Windows on any hardware platform. Windows will select this section for device installation on these platforms. However, because the section contains no specific device descriptions or hardware IDs, Windows will not install any devices through this INF file.

# Sample INF File for Device Installation on Multiple Versions of Windows

2/14/2019 • 2 minutes to read • [Edit Online](#)

The sample INF file (*MultiOS.inf*) shows how a single INF file can be used to install a device on multiple versions of Windows. As of March 2015, this sample is no longer available in the WDK. It is still available as part of the [Windows 8 driver samples](#) and the [Windows 8.1 driver samples](#).

# Creating International INF Files

12/5/2018 • 2 minutes to read • [Edit Online](#)

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, by following these guidelines:

- 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 Windows 2000 and later versions of Windows, this is the recommended method for supporting international markets.

- To create a separate INF file for each locale, start with a main INF file that contains 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 2000 and later versions of Windows, 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 operating system version, on a single installation medium because Windows cannot determine which INF file to use.

## Locale-Specific Versions of Driver Files

If you have to provide locale-specific versions of driver files for Windows 2000 and later versions of Windows, mark each version of each file with its locale. Be sure to mark files that are not locale-specific as language-neutral. You can do this by adding the following macro definition to your resource file:

```
#define VER_LANGNEUTRAL
```

This definition must appear before the preprocessor directive that includes *common.ver*.

After compiling your files, you can verify that each is marked as language-neutral by doing the following:

1. Right-click the file in Windows Explorer.
2. Click **Properties**.
3. Click the **Version** tab.

The **Language** selection in the **Other version information** pane contains a value that identifies the file as Language Neutral, or as intended for a specific locale.

Put the locale-specific 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 by 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.

For example:

```
[SourceDisksNames]
1=%DiskName%,,%LocaleSubDir%

[SourceDisksFiles]
mysftwre.exe=1

[Strings]           ; No language ID implies English
DiskName="My Excellent Software"
LocaleSubDir="English"
[Strings.0407]      ; 0407 is the language ID for German
DiskName="Meine ausgezeichnete Software"
LocaleSubDir="German"
```

### Creating Unicode INF Files

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

Some applications, such as Notepad, allow you to create a Unicode file in either little-endian or big-endian format. Windows supports INF files that use either format.

# Specifying Driver Load Order

12/1/2020 • 6 minutes to read • [Edit Online](#)

For most devices, the physical hierarchy of the devices on a computer determines the order in which Windows and the PnP manager load drivers. Windows 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**, **BootFlags**, **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 when the PnP manager finds a device that the driver services.

- Driver for a device required to start the computer

If a device is required to start the computer, the drivers for the device should have a start type of SERVICE\_BOOT\_START (0x0).

- Non-boot-start 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 Windows will load the driver during system initialization.

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

- Non-PnP 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 when Windows configures a device that the driver services. Conversely, a driver should be able to be unloaded any time that 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 as follows:

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

For example, the function driver can be certain that any lower-filter drivers are loaded.

However, be aware that a driver in the device stack cannot depend on being loaded sequentially after a device's lower drivers, because the driver might be loaded previously when another device was configured.

Filter drivers in 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 Windows and the PnP manager use the **StartType** entries in INF files:

1. On system startup, 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 device node (*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 (except when **StartType** is **SERVICE\_DISABLED**) before proceeding to start the devices. Many of these drivers are **SERVICE\_DEMAND\_START** drivers.

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 the devices are 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 non-PnP 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 because 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. Be aware that the **Dependencies** information is only processed for non-PnP 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 Microsoft Windows SDK documentation for more information about the service control manager.

### Using **BootFlags** to Promote a Driver's **StartType** at Boot Depending on Boot Scenario

The operating system can promote a driver's **StartType** to be a boot start driver depending on the **BootFlags** value specified in the driver's INF. You can specify one or more (ORed) of the following numeric values in the INF file, expressed as a hexadecimal value:

- If a driver should be promoted to be a boot start driver on network boot, specify **0x1** (CM\_SERVICE\_NETWORK\_BOOT\_LOAD).
- If a driver should be promoted on booting from a VHD, specify **0x2** (CM\_SERVICE\_VIRTUAL\_DISK\_BOOT\_LOAD).
- If a driver should be promoted while booting from a USB disk, specify **0x4** (CM\_SERVICE\_USB\_DISK\_BOOT\_LOAD).
- If a driver should be promoted while booting from SD storage, specify **0x8** (CM\_SERVICE\_SD\_DISK\_BOOT\_LOAD).
- If a driver should be promoted while booting from a disk on a USB 3.0 controller, specify **0x10** (CM\_SERVICE\_USB3\_DISK\_BOOT\_LOAD).
- If a driver should be promoted while booting with measured boot enabled, specify **0x20** (CM\_SERVICE\_MEASURED\_BOOT\_LOAD).
- If a driver should be promoted while booting with verifier boot enabled, specify **0x40** (CM\_SERVICE\_VERIFIER\_BOOT\_LOAD).
- If a driver should be promoted on WinPE boot, specify **0x80** (CM\_SERVICE\_WINPE\_BOOT\_LOAD).

For more information about promoting a driver's **StartType** at boot, depending on the boot scenario, see [INF AddService directive](#).

# Using Dirids

11/2/2020 • 4 minutes to read • [Edit Online](#)

Many of the directories that appear in INF files can be expressed by using directory identifiers (*dirids*), which are numbers that identify specific directories. Applications can use, but cannot reassign the system-defined directories that are associated with *dirids* whose values are from -1 through 32767.

To create *dirids* with user-defined values from 32768 through 65534, or 65536 and up, use the **SetupSetDirectoryId** function (described in the Microsoft Windows SDK documentation).

Be aware that a *dirid* with a value of 65535 is considered synonymous with a *dirid* with a value of -1, although the latter (*dirid*-1) is preferred.

If you intend to use *dirids* in your INF file, consider the following two guidelines:

1. When the syntax for an INF file entry explicitly specifies a *dirid* value (the **INF DestinationDirs section**, for example), express that value as a number.

The following example demonstrates this syntax:

```
[DestinationDirs]
DefaultDestDir = 11 ; \system32 directory on Windows 2000 and later versions
```

2. When the syntax for an INF file entry specifies a file path, you can use a system-supplied string substitution to represent part or all of this path. This substitution has the following form:

**%dirid%**

This form consists of a percent (%) character, followed by the *dirid* for the directory that you want to specify, followed by another percent (%) character. A terminating backslash () character separates this expression from a following file name or additional directories in the path.

The following example demonstrates this syntax:

```
[aic78xx_Service_Inst]
ServiceBinary = %12%\aic78xx.sys
```

When fully expanded, the path shown in the previous example becomes *c\windows\system32\drivers\aic78xx.sys* (assuming that Windows was installed in the *c\windows* directory). Be aware that the string substitution, or **%dirid%** form, can be used anywhere a string is expected, with the exception of the **INF Strings section** of the INF file.

The two following examples show how string substitution should *not* be used.

```
[DestinationDirs]
DefaultDestDir = %11% ; Error! - number expected

[aic78xx_Service_Inst]
ServiceBinary = 12\aic78xx.sys ; Error! - unknown directory name
```

In the first example, the syntax for the **DefaultDestDir** entry requires its value to be a number. However, the **%11%** expression expands to a string. In the second example, the INF writer apparently intended to set the value for the **ServiceBinary** entry to a file in the directory that contains drivers (see the following

table for more information). The error occurs because Windows looks for the specified file in a directory named "12", which probably does not exist on the computer.

The following table shows several commonly used *dirids*, and the directories they represent. The values most commonly specified by device INF files and driver INF files are listed toward the top of the table.

VALUE	DESTINATION DIRECTORY
01	<i>SourceDrive\pathname</i> (the directory from which the INF file was installed)
10	Windows directory. This is equivalent to %SystemRoot%.
11	System directory. This is equivalent to %SystemRoot%\system32 for Windows 2000 and later versions of Windows..
12	Drivers directory. This is equivalent to %SystemRoot%\system32\drivers for Windows 2000 and later versions of Windows.
13	Driver package's <a href="#">Driver Store</a> directory. For Windows 8.1 and later versions of Windows, specifies the path to the Driver Store directory where the driver package was imported. Don't use <a href="#">DelFiles</a> on a file for which <b>DestinationDirs</b> includes <i>dirid</i> 13. The optional subdirectory in the <b>SourceDiskFiles</b> section for a file must match the subdirectory in the <b>DestinationDirs</b> section for the entry that applies to this file. Don't use <a href="#">CopyFiles</a> to rename a file for which <b>DestinationDirs</b> includes <i>dirid</i> 13.
17	INF file directory
18	Help directory
20	Fonts directory
21	Viewers directory
23	Color directory (ICM) ( <i>not</i> used for installing printer drivers)

VALUE	DESTINATION DIRECTORY
24	<p>Root directory of the system disk.</p> <p>This is the root directory of the disk on which Windows files are installed. For example, if <i>dirid</i> 10 is "C:\winnt", then <i>dirid</i> 24 is "C:\".</p>
25	Shared directory
30	Root directory of the boot disk, also known as "ARC system partition". (This might or might not be the same directory as the one represented by <i>dirid</i> 24.)
50	<p>System directory</p> <p>This is equivalent to %SystemRoot%\system.</p>
51	Spool directory ( <i>not</i> used for installing printer drivers – see <a href="#">Printer Dirids</a> )
52	Spool drivers directory ( <i>not</i> used for installing printer drivers)
53	User profile directory
54	Directory where <i>Ntldr.exe</i> and <i>Osloader.exe</i> are located
55	Print processors directory ( <i>not</i> used for installing printer drivers)
-1	Absolute path

*Dirid* values from 16384 through 32767 are reserved for special shell folders. The following table shows *dirid* values for these folders.

VALUE	SHELL SPECIAL FOLDER
16406	<i>All Users\Start Menu</i>
16407	<i>All Users\Start Menu\Programs</i>
16408	<i>All Users\Start Menu\Programs\Startup</i>
16409	<i>All Users\Desktop</i>

VALUE	SHELL SPECIAL FOLDER
16415	<i>All Users\Favorites</i>
16419	<i>All Users\Application Data</i>
16422	<i>Program Files</i>
16425	<i>%SystemRoot%\SysWOW64</i>
16426	<i>%ProgramFiles(x86)%</i>
16427	<i>Program Files\Common</i>
16428	<i>%ProgramFiles(x86)%\Common</i>
16429	<i>All Users\Templates</i>
16430	<i>All Users\Documents</i>

In addition to the values in this table that are defined in *Setupapi.h*, you can use any of the CSIDL\_Xxx values that are defined in *Shlobj.h*. To define a *dirid* value for a folder not listed in this table, add 16384 (0x4000) to the CSIDL\_Xxx value. For more information about CSIDL\_Xxx values, see the Windows SDK documentation.

# Accessing INF Files from a Device Installation Application

11/2/2020 • 2 minutes to read • [Edit Online](#)

Most INF files are used by system class installers and other *device installation applications*. This section describes common operations that vendor-supplied device installation applications can perform on INF files by using the [general Setup functions](#). For complete information about how to use these functions, see the Windows SDK documentation.

This section contains the following topics:

[Opening and Closing an INF File](#)

[Retrieving Information from an INF File](#)

# Opening and Closing an INF File

11/2/2020 • 2 minutes to read • [Edit Online](#)

Before a *device installation 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 have 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 that uses [\*\*SetupOpenAppendInfFile\*\*](#). When subsequent **SetupAPI** functions reference an open INF file, they will also be able to access any information that is 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\*\*](#).

# Retrieving Information from an INF File

11/2/2020 • 2 minutes to read • [Edit Online](#)

Once you have a handle to an INF file, you can retrieve information from it in a variety of ways. Functions such as [SetupGetInflInformation](#), [SetupQueryInfFileInformation](#), and [SetupQueryInfVersionInformation](#) retrieve information about the specified INF file.

Other functions, such as [SetupGetSourceInfo](#) and [SetupGetTargetPath](#), obtain information about the source files and target directories.

Functions such as [SetupGetLineText](#) and [SetupGetStringField](#) enable you to directly access information that is stored in a line or field of an INF file. These functions are used internally by the higher-level [SetupAPI](#) functions but are available if you have to directly access information at the line or field level.

# Providing Icons for a Device

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic describes how you can provide custom icons for a device by referencing them in a driver's INF file. You can provide icons that appear in Device Manager, Windows Explorer, or both, as appropriate.

## Adding icons for Device Manager

You can either embed a custom icon in a DLL or provide a standalone .ico file. If your driver is already a DLL file, the first is the easiest option because it does not require copying any additional files.

To embed the icon in a DLL, use an entry like this:

```
[<DDInstall>]
AddProperty = DeviceIconProperty

[DeviceIconProperty]
DeviceIcon,,,,"%13%\UmdfDriver.dll,-100"
```

The above example uses DIRID 13 to copy the file to the Driver Store, which avoids needing to copy it anywhere else. The entry follows the format `<Resource.dll>,-<IconResourceID>`, so the 100 signifies the resource ID of the icon in the resource table of the DLL. For more on DIRID 13, see [Using a Universal INF File](#).

To reference a standalone .ico file, use an entry like this:

```
[<DDInstall>]
AddProperty = DeviceIconProperty

[DeviceIconProperty]
DeviceIcon,,,,"%13%\vendor.ico"
```

## Adding icons for storage volumes in Explorer

The shell uses **Icons** and **NoMediaIcons** registry values to represent the device in AutoPlay, My Computer, and file Open dialog boxes.

To add these, include an [INF AddReg directive](#) under an [INF DDInstall.HW section](#) for the device. In the **AddReg** section, specify **Icons** and **NoMediaIcons** value entries, as shown in the following example:

```
[DDInstall.NT.HW]
AddReg = IconInformation

[IconInformation]
HKR, , Icons, 0x10000, "media-inserted-icon-file"
HKR, , NoMediaIcons, 0x10000, "no-media-inserted-icon-file"
```

Then include an [INF SourceDisksFiles section](#) that lists the icon files and a corresponding [INF CopyFiles directive](#) that copies them to the system.

The **Icons** and **NoMediaIcons** value entries are stored under the **Device Parameters** key under the device's **hardware key**. For example, `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\<Hardware ID>\Device Parameters` would contain entries like the following:

- Icons [REG\_MULTI\_SZ] = %SystemRoot%\system32\icon.ico
- NoMediaIcons [REG\_MULTI\_SZ] = %SystemRoot%\system32\noicon.ico

To modify the **Device Parameters** key from user mode, use [SetupDiCreateDevRegKey](#) or [SetupDiOpenDevRegKey](#).

From kernel mode, use [IoOpenDeviceRegistryKey](#).

## Resources

When you create icons, follow the guidelines that are provided in [Icons](#). These guidelines describe how to create icons that have the appearance and behavior of Windows graphical elements.

# Driver Signing

11/2/2020 • 2 minutes to read • [Edit Online](#)

Driver signing associates a [digital signature](#) with a [driver package](#).

Windows device installation uses [digital signatures](#) to verify the integrity of driver packages and to verify the identity of the vendor (software publisher) who provides the driver packages. In addition, the [kernel-mode code signing policy](#) for 64-bit versions of Windows Vista and later versions of Windows specifies that a kernel-mode driver must be signed for the driver to load.

**Note** Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) and Windows Server 2016 kernel-mode drivers must be signed by the Windows Hardware Dev Center Dashboard, which requires an EV certificate. For details, see [Driver Signing Policy](#).

All drivers for Windows 10 (starting with version 1507, Threshold 1) signed by the Hardware Dev Center are SHA2 signed. For details specific to operating system versions, see [Signing requirements by version](#).

Kernel-mode driver binaries embed signed with dual (SHA1 and SHA2) certificates from a third party certificate vendor for operating systems earlier than Windows 10 may not load, or may cause a system crash on Windows 10. To fix this problem, install [KB 3081436](#).

## In this section

- [Windows 10 in S mode Driver Requirements](#)
- [Managing the Signing Process](#)
- [Signing Drivers during Development and Test](#)
- [Signing Drivers for Public Release](#)
- [Troubleshooting Install and Load Problems with Signed Driver Packages](#)
- [Microsoft Security Advisory 2880823](#)

For general information about driver signing on Windows Vista and later versions of Windows, see the white paper [Digital Signatures for Kernel Modules on Systems Running Windows Vista](#).

# Digital Signatures

11/2/2020 • 2 minutes to read • [Edit Online](#)

Digital signatures are based on Microsoft public key infrastructure technology, which is based on Microsoft [Authenticode](#) combined with an infrastructure of trusted certification authorities (CAs). Authenticode, which is based on industry standards, allows vendors, or *software publishers*, to sign either a file or a collection of files (such as a [driver package](#)) by using a code-signing [digital certificate](#) that is issued by a CA.

Windows uses a valid digital signature to verify the following:

- The file, or the collection of files, is signed.
- The signer is trusted.
- The certification authority that authenticated the signer is trusted.
- The collection of files was not altered after it was published.

For example, this signing process for a [driver package](#) involves the following:

- A publisher obtains an *X.509 digital certificate* from a CA. An Authenticode certificate is also referred to as a *signing certificate*. A signing certificate is a set of data that identifies a publisher, and is issued by a CA only after the CA has verified the identity of the publisher. A CA can be a Microsoft CA, a third-party commercial CA, or an Enterprise CA.

The signing certificate is used to sign the [catalog file](#) of a driver package or to [embed a signature](#) in a driver file. Certificates that identify trusted publishers and trusted CAs are installed in [certificate stores](#) that are maintained by Windows.

- The signing certificate includes a private key and a public key, which is known as the *key pair*. The private key is used to sign the catalog file of a [driver package](#) or to embed a signature in a driver file. The public key is used to verify the signature of a driver package's catalog file or a signature that is embedded in a driver file.
- To sign a catalog file or to embed a signature in a file, the signing process first generates a cryptographic hash, or *thumbprint*, of the file. The signing process then encrypts the file thumbprint with a private key and adds the thumbprint to the file.

The signing process also adds information about the publisher and the CA that issued the signing certificate. The digital signature is added to the file in a section of the file that is not processed when the file thumbprint is generated.

- To verify the digital signature of a file, Windows extracts the information about the publisher and the CA and uses the public key to decrypt the encrypted file thumbprint.

Windows accepts the integrity of the file and the authenticity of the publisher only if the following are true:

- The decrypted thumbprint matches the thumbprint of the file.
- The certificate of the publisher is installed in the [Trusted Publishers certificate store](#).
- The root certificate of the CA that issued the publisher's certificate is installed in the [Trusted Root Certification Authorities certificate store](#).

For more information about how the Plug and Play (PnP) device installation uses the digital signature of a [driver package's catalog file](#), see [Digital Signatures and PnP Device Installation](#).

For more information about Microsoft public key infrastructure technology, code signing, and digital signatures, see [Introduction to Code Signing](#) and [Code Signing Best Practices](#).

# Authenticode Digital Signatures

11/2/2020 • 2 minutes to read • [Edit Online](#)

Authenticode is a Microsoft code-signing technology that identifies the publisher of Authenticode-signed software. Authenticode also verifies that the software has not been tampered with since it was signed and published.

Authenticode uses cryptographic techniques to verify publisher identity and code integrity. It combines digital signatures with an infrastructure of trusted entities, including certificate authorities (CAs), to assure users that a driver originates from the stated publisher. Authenticode allows users to verify the identity of the software publisher by chaining the certificate in the digital signature up to a trusted root certificate.

Using Authenticode, the software publisher signs the driver or [driver package](#), tagging it with a [digital certificate](#) that verifies the identity of the publisher and also provides the recipient of the code with the ability to verify the integrity of the code. A certificate is a set of data that identifies the software publisher. It is issued by a CA only after that authority has verified the software publisher's identity. The certificate data includes the publisher's public cryptographic key. The certificate is typically part of a chain of such certificates, ultimately referenced to a well-known CA such as VeriSign.

Authenticode code signing does not alter the executable portions of a driver. Instead, it does the following:

- With embedded signatures, the signing process embeds a digital signature within a nonexecution portion of the driver file. For more information about this process, see [Embedded Signatures in a Driver File](#).
- With digitally-signed [catalog files \(.cat\)](#), the signing process requires generating a file hash value from the contents of each file within a [driver package](#). This hash value is included in a catalog file. The catalog file is then signed with an embedded signature. In this way, catalog files are a type of detached signature.

**Note** The [Hardware Certification Kit \(HCK\)](#) has test categories for a variety of device types. The list of test categories can be found at [HLK API Reference](#). If a test category for the device type is included in this list, the software publisher should obtain a [WHQL release signature](#) for the [driver package](#). However, if the HCK does not have a test program for the device type, the software publisher can sign the driver package by using the Microsoft Authenticode technology. For more information about this process, see [Signing Drivers for Public Release](#).

# Catalog Files and Digital Signatures

11/2/2020 • 2 minutes to read • [Edit Online](#)

A digitally-signed catalog file (*.cat*) can be used as a digital signature for an arbitrary collection of files. A catalog file contains a collection of cryptographic hashes, or *thumbprints*. Each thumbprint corresponds to a file that is included in the collection.

Plug and Play (PnP) device installation recognizes the signed catalog file of a [driver package](#) as the [digital signature](#) for the driver package, where each thumbprint in the catalog file corresponds to a file that is installed by the driver package. Regardless of the intended operating system, cryptographic technology is used to digitally-sign the catalog file.

PnP device installation considers the digital signature of a [driver package](#) to be invalid if any file in the driver package is altered after the driver package was signed. Such files include the INF file, the catalog file, and all files that are copied by [INF CopyFiles directives](#). For example, even a single-byte change to correct a misspelling invalidates the digital signature. If the digital signature is invalid, you must either resubmit the driver package to the [Windows Hardware Quality Labs \(WHQL\)](#) for a new signature or generate a new [Authenticode](#) signature for the driver package.

Similarly, changes to a device's hardware or firmware require a revised [device ID](#) value so that the system can detect the updated device and install the correct driver. Because the revised device ID value must appear in the INF file, you must either resubmit the package to WHQL for a new signature or generate a new [Authenticode](#) signature for the driver package. You must do this even if the driver binaries do not change.

The [CatalogFile](#) directive in the [INF Version section](#) of the driver's [INF file](#) specifies the name of the catalog file for the driver package. During driver installation, the operating system uses the [CatalogFile](#) directive to identify and validate the catalog file. The system copies the catalog file to the `%SystemRoot%\CatRoot` directory and the INF file to the `%SystemRoot%\Inf` directory.

## Guidelines for Catalog Files

Starting with Windows 2000, if the [driver package](#) installs the same binaries on all versions of Windows, the INF file can contain a single, undecorated [CatalogFile](#) directive. However, if the package installs different binaries for different versions of Windows, the INF file should contain decorated [CatalogFile](#) directives. For more information about the [CatalogFile](#) directive, see [INF Version Section](#).

If you have more than one driver package, you should create a separate catalog file for each driver package and give each catalog file a unique file name. Two unrelated driver packages cannot share a single catalog file. However, a single driver package that serves multiple devices requires only one catalog file.

# Embedded Signatures in a Driver File

12/1/2020 • 2 minutes to read • [Edit Online](#)

In 64-bit versions of Windows Vista and later versions of Windows, the kernel-mode code signing requirements state that a released kernel-mode *boot-start driver* must have an embedded [Software Publisher Certificate \(SPC\)](#) signature. An embedded signature is not required for drivers that are not boot-start drivers.

## NOTE

Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) and Windows Server 2016 kernel-mode drivers must be signed by the Windows Hardware Dev Center Dashboard, which requires an EV certificate. For more info about these changes, see [Driver Signing Changes in Windows 10](#).

Having an embedded signature saves significant time during system startup because there is no need for the system loader to locate the [catalog file](#) for the driver at system startup. A typical computer might have many different catalog files in the catalog root store (`%System%\CatRoot`). Locating the correct catalog file to verify the thumbprint of a driver file can require a substantial amount of time.

In addition to the load-time signature requirement that is enforced by the kernel-mode code signing policy, Plug and Play (PnP) device installation also enforces an install-time signing requirement. To comply with the [PnP device installation signing requirements](#) of Windows Vista and later versions of Windows, a [driver package](#) for a PnP device must have a signed catalog file.

Embedded signatures do not interfere with the signature of a catalog file because the thumbprints that are contained in a catalog file and the thumbprint in an embedded signature selectively exclude the signature part of the driver file.

Driver files are signed by using the [SignTool](#) tool.

# Digital Signatures and PnP Device Installation (Windows Vista and Later)

11/2/2020 • 6 minutes to read • [Edit Online](#)

On Windows Vista and later versions of Windows, Plug and Play (PnP) device installation uses the [digital signature](#) of a [driver package's catalog file](#) to do the following:

- Verify the identity of the publisher of the driver package. Windows uses the identity to allow users to choose whether to trust a driver's publisher.
- Determine whether the driver package was altered after it was published.

PnP device installation on Windows Vista and later versions of Windows support the following types of digital signatures for [driver packages](#):

- Signature types that can be used for drivers that are released to the general public:
  - Signatures generated by a Windows signing authority for:
    1. Inbox drivers
    2. Drivers certified and signed through the Windows Hardware Quality Labs (WHQL)
    3. Windows Sustained Engineering (SE) updates.
  - Signatures that are not generated by a Windows signing authority, but do comply with the following:
    1. The [kernel-mode code signing policy](#) for 64-bit versions of Windows Vista and later versions of Windows.
    2. The [PnP device installation signing requirements](#) for 32-bit and 64-bit versions of Windows Vista and later versions of Windows.

This type of signature is generated by using a [Software Publisher Certificate \(SPC\)](#) that is obtained from a third-party certification authority (CA) that is authorized by Microsoft to issue such certificates.

- Signatures that are not generated by a Windows signing authority and do not comply with the [kernel-mode code signing policy](#), but do comply with the [PnP device installation signing requirements](#). This type of signature can be used to sign kernel-mode drivers for 32-bit versions of Windows Vista and later versions of Windows. This type of signature is generated by using a [commercial release certificate](#) that is obtained from a CA that is a member of the Microsoft Root Certificate Program.
- Signatures for deploying drivers only within corporate network environments, which are created by a digital certificate that is created and managed by Enterprise CA. Detailed information about how to configure an Enterprise CA is outside the scope of this documentation.

For information about how to create an Enterprise CA, see the "Code Signing Best Practices" white paper on the [Driver Signing Requirements for Windows](#) website.

- Signature types that can be used in-house during the development and test of drivers:
  - Signatures generated by the [WHQL test signature program](#)
  - Signatures generated by a [MakeCert test certificate](#)
  - Signatures created by a [commercial test certificate](#) that is obtained from a CA that is a member of the Microsoft Root Certificate Program

- Signatures generated by an [Enterprise CA test certificate](#)

Windows Vista and later versions of Windows include the following features that provide support for signatures that are generated by third parties:

- Administrators can control which driver publishers are trusted. Windows Vista and later versions of Windows installs drivers from trusted publishers without prompting. It never installs drivers from publishers that the administrator has chosen not to trust.
- The driver-signing policy is always set to *Warn*. This eliminates the *Ignore* and *Block* options that were available in Windows Server 2003, Windows XP, and Windows 2000. An administrator must always authorize the installation of unsigned drivers or a driver from a publisher that is not yet trusted.
- All [device setup classes](#) are treated equally. On Windows Server 2003, Windows XP, and Windows 2000, driver packages that were signed by WHQL must have an INF file that specifies a device setup class that is defined in `%SystemRoot%/inf/Certclas.inf`. Otherwise, Windows treats the driver package as unsigned.
- Starting with Windows Vista, when there are several compatible drivers to choose from, the ranking algorithm that the operating system uses to select the best driver includes drivers that have third-party signatures.

This algorithm ranks drivers in the following way:

- If the [AllSignersEqual group policy](#) is disabled, the operating system ranks drivers that are signed with a Microsoft signature higher than drivers that are signed with a third-party signature. This ranking occurs even if a driver that is signed with a third-party signature is, in all other ways, a better match for a device.
- If the [AllSignersEqual group policy](#) is enabled, the operating system ranks all digitally signed drivers equally.

**Note** Starting with Windows 7, the [AllSignersEqual group policy](#) is enabled by default. In Windows Vista and Windows Server 2008, the [AllSignersEqual](#) group policy is disabled by default. IT departments can override the default ranking behavior by enabling or disabling the [AllSignersEqual](#) group policy.

Before installing a driver, Windows analyzes the [driver package's](#) digital signature. If a signature is present, Windows uses the signature to validate the files in the driver package. Based on the results of this analysis, Windows categorizes the digital signature as follows:

- **Signed by a Windows signing authority.** These drivers are either included in the default installation of Windows (*inbox drivers*), signed for release by WHQL, or signed by Windows SE.
- **Signed by a trusted publisher.** These drivers have been signed by a third-party, and user has explicitly chosen to always trust signed drivers from this publisher.
- **Signed by an untrusted publisher.** These drivers have been signed by a third-party, and the user has explicitly chosen to never trust drivers from this publisher.
- **Signed by a publisher of unknown trust.** These drivers have been signed by a third-party, and the user has not indicated whether to trust this publisher.
- **Altered.** These drivers are signed, but Windows has detected that at least one file in the [driver package](#) has been altered after the package was signed.
- **Unsigned.** These drivers are either unsigned or have an invalid signature. Valid signatures must be created by using a certificate that was issued by a trusted CA.

Starting with Windows Vista, when the operating system installs a driver on a computer for the first time, it preinstalls, or stages, the driver in the [driver store](#). To preinstall a driver, Windows copies the driver package to the driver store and saves a copy of the driver package's INF file in the system INF directory. Windows subsequently will silently install a driver for a matching device by using the copy of the driver package in the driver store. User

interaction is not required when Windows installs a preinstalled driver for a device.

Whether Windows will preinstall a [driver package](#) depends on the signature category, user credentials, and user interaction, as follows:

- **Signed by a Windows signing authority or a trusted publisher.** Windows silently preinstalls the driver package for system administrators and standard users (users without administrator credentials). Windows does not display any user dialog boxes.
- **Signed by an untrusted publisher.** Windows does not preinstall the driver package.
- **Signed by a publisher of unknown trust.** Windows displays a dialog box to a system administrator that informs the administrator that the publisher of the driver package is not yet trusted. The dialog box provides the administrator the option to install the driver package and the option to always trust the publisher. Windows does not display a dialog box to a standard user and does not preinstall the driver package for the standard user.
- **Altered or unsigned.** Windows displays a dialog box that appropriately warns a system administrator that the signature could not be verified. The dialog box provides the administrator the option to install or not to install the driver package. Windows does not display a dialog box to a standard user and does not preinstall the driver package for a standard user.

For more information about driver signatures and installation, see [Signature Categories and Driver Installation](#).

# Digital Certificates

12/5/2018 • 2 minutes to read • [Edit Online](#)

Digital certificates bind an entity, such as an individual, organization, or system, to a specific pair of public and private keys. Digital certificates can be thought of as electronic credentials that verify the identity of an individual, system, or organization.

Various types of digital certificates are used for a variety of purposes, such as the following:

- Secure Multipurpose Internet Mail Extensions (S/MIME) digital certificates for signing email messages.
- Secure Sockets Layer (SSL) and Internet Protocol security (IPSec) digital certificates for authenticating network connections.
- Smart card digital certificates for logging on to personal computers.

Windows code-signing technologies use X.509 code-signing certificates, a standard that is owned by the Internet Engineering Task Force (IETF). Code-signing certificates allow software publishers or distributors to digitally sign software.

A certificate is contained in a digital signature and verifies the origin of the signature. The certificate owner's public key is in the certificate and is used to verify the digital signature. This practice avoids having to set up a central facility for distributing the certificates. The certificate owner's private key is kept separately and is known only to the certificate owner.

Software publishers must obtain a certificate from a certification authority (CA), which vouches for the integrity of the certificate. Typically, a CA requires the software publisher to provide unique identifying information. The CA uses this information to authenticate the identity of the requester before issuing the certificate. Software publishers must also agree to abide by the policies that are set by the CA. If they fail to do so, the CA can revoke the certificate.

Once a certificate is obtained from the CA, software publishers must store the certificate locally in the computer. For more information about this process, see [Certificate Stores](#).

# Certificate Stores

12/5/2018 • 2 minutes to read • [Edit Online](#)

On a computer that has the Windows operating system installed, the operating system stores a certificate locally on the computer in a storage location called the *certificate store*. A certificate store often has numerous certificates, possibly issued from a number of different certification authorities (CAs).

This section includes the following topics:

[Local Machine and Current User Certificate Stores](#)

[Trusted Root Certification Authorities Certificate Store](#)

[Trusted Publishers Certificate Store](#)

# Local Machine and Current User Certificate Stores

11/2/2020 • 2 minutes to read • [Edit Online](#)

Each of the system certificate stores has the following types:

- Local machine certificate store

This type of certificate store is local to the computer and is global to all users on the computer. This certificate store is located in the registry under the HKEY\_LOCAL\_MACHINE root.

- Current user certificate store

This type of certificate store is local to a user account on the computer. This certificate store is located in the registry under the HKEY\_CURRENT\_USER root.

For specific registry locations of certificate stores, see [System Store Locations](#).

Be aware that all current user certificate stores *except the Current User/Personal store* inherit the contents of the local machine certificate stores. For example, if a certificate is added to the local machine [Trusted Root Certification Authorities certificate store](#), all current user Trusted Root Certification Authorities certificate stores (with the above caveat) also contain the certificate.

## NOTE

The driver signing verification during Plug and Play (PnP) installation requires that root and Authenticode certificates, including [test certificates](#), are located in a local machine certificate store.

For more information about how to add or delete certificates from the system certificate stores, see [CertMgr](#).

# Trusted Root Certification Authorities Certificate Store

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, the Plug and Play (PnP) manager performs driver signature verification during device and driver installation. However, the PnP manager can successfully verify a digital signature only if the following statements are true:

- The signing certificate that was used to create the signature was issued by a certification authority (CA).
- The corresponding root certificate for the CA is installed in the *Trusted Root Certification Authorities certificate store*. Therefore, the Trusted Root Certification Authorities certificate store contains the root certificates of all CAs that Windows trusts.

By default, the Trusted Root Certification Authorities certificate store is configured with a set of public CAs that has met the requirements of the Microsoft Root Certificate Program. Administrators can configure the default set of trusted CAs and install their own private CA for verifying software.

**Note** A private CA is unlikely to be trusted outside the network environment.

Having a valid digital signature ensures the authenticity and integrity of a [driver package](#). However, it does not mean that the end-user or a system administrator implicitly trusts the software publisher. A user or administrator must decide whether to install or run an application on a case-by-case basis, based on their knowledge of the software publisher and application. By default, a publisher is trusted only if its certificate is installed in the [Trusted Publishers certificate store](#).

The name of the Trusted Root Certification Authorities certificate store is *root*. You can manually install the root certificate of a private CA into the Trusted Root Certification Authorities certificate store on a computer by using the [CertMgr](#) tool.

**Note** The driver signing verification policy that is used by the PnP manager requires that the root certificate of a private CA has been previously installed in the local machine version of the Root Certification Authorities certificate store. For more information, see [Local Machine and Current User Certificate Stores](#).

For more information about driver signing, see [Driver Signing Policy](#).

# Trusted Publishers Certificate Store

11/2/2020 • 2 minutes to read • [Edit Online](#)

The *Trusted Publishers certificate store* contains information about the Authenticode (signing) certificates of trusted publishers that are installed on a computer. In order to test and debug your [driver packages](#) within your organization, your company should install the Authenticode certificates that are used to sign driver packages in the Trusted Publishers certificate store. Install the Authenticode certificates on each computer in the workgroup or organizational unit that runs signed code. The name of the Trusted Publishers certificate store is *trustedpublisher*.

If a publisher's Authenticode certificate is in the Trusted Publishers certificate store, Windows installs a [driver package](#) that was digitally signed by the certificate without prompting the user (*silent install*). By installing the Authenticode certificates in the Trusted Publishers certificate store, you can automate the installation of your driver package on various systems that are used for internal testing and debugging.

**Important** This practice of automating the installation of driver packages is only suggested for your internal systems. This practice should never be followed for any driver package that is distributed outside your organization.

The Trusted Publishers certificate store differs from the [Trusted Root Certification Authorities certificate store](#) in that only *end-entity* certificates can be trusted. For example, if an Authenticode certificate from a CA was used to [test-sign](#) a driver package, adding that certificate to the Trusted Publishers certificate store does not configure all certificates that this CA issued as trusted. Each certificate must be added separately to the Trusted Publishers certificate store.

Use a Group Policy to distribute certificates to an organizational unit on a network. In this situation, the administrator adds a Certificate Rule to a Group Policy to establish trust in a publisher. Certificate Rules are part of the software restriction policies that are supported on the following Windows versions:

- Windows Vista and later versions of Windows.
- Windows Server 2003.

You can manually install the Authenticode certificates into the Trusted Publishers certificate store on a computer by using the [CertMgr](#) tool.

**Note** The driver signing verification policy used by Plug and Play requires that the Authenticode certificate of a CA has been previously installed in the local machine version of the Trusted Publishers certificate store. For more information, see [Local Machine and Current User Certificate Stores](#).

For more information about software restriction policies and using Certificate Rules, see the information in the Windows Help and Support Center.

For more information about how to deploy Authenticode certificates in an enterprise by using Group Policy, see the readme file *Selfsign\_readme.htm*, which is located in the *src\general\build\driversigning* directory of the WDK.

For more information about certificate stores, see the [Code Signing Best Practices](#) website.

# Driver Signing Policy

11/2/2020 • 3 minutes to read • [Edit Online](#)

## NOTE

Starting with Windows 10, version 1607, Windows will not load any new kernel-mode drivers which are not signed by the Dev Portal. To get your driver signed, first [Register for the Windows Hardware Dev Center program](#). Note that an [EV code signing certificate](#) is required to establish a dashboard account.

There are many different ways to submit drivers to the portal. For production drivers, you should submit HLK/HCK test logs, as described below. For testing on Windows 10 client only systems, you can submit your drivers for [attestation signing](#), which does not require HLK testing. Or, you can submit your driver for Test signing as described on the [Create a new hardware submission](#) page.

## Exceptions

Cross-signed drivers are still permitted if any of the following are true:

- The PC was upgraded from an earlier release of Windows to Windows 10, version 1607.
- Secure Boot is off in the BIOS.
- Drivers were signed with an end-entity certificate issued prior to July 29th 2015 that chains to a supported cross-signed CA.

To prevent systems from failing to boot properly, boot drivers will not be blocked, but they will be removed by the Program Compatibility Assistant.

## Signing a driver for client versions of Windows

To sign a driver for Windows 10, follow these steps:

1. For each version of Windows 10 that you want to certify on, download the Windows HLK (Hardware Lab Kit) for that version and run a full cert pass against the client for that version. You'll get one log per version.
2. If you have multiple logs, merge them into a single log using the most recent HLK.
3. Submit your driver and the merged HLK test results to the [Windows Hardware Developer Center Dashboard portal](#).

For version-specific details, please review the [WHCP \(Windows Hardware Compatibility Program\) policy](#) for the Windows versions you want to target.

To sign a driver for Windows 7, Windows 8, or Windows 8.1, use the appropriate HCK (Hardware Certification Kit). For more information, see the [Windows Hardware Certification Kit User's Guide](#).

## Signing a driver for earlier versions of Windows

Before Windows 10, version 1607, the following types of drivers require an Authenticode certificate used together with Microsoft's cross-certificate for cross-signing:

- Kernel-mode device drivers
- User-mode device drivers
- Drivers that stream protected content. This includes audio drivers that use Protected User Mode Audio

(PUMA) and Protected Audio Path (PAP), and video device drivers that handle protected video path-output protection management (PVP-OPM) commands. For more information, see [Code-signing for Protected Media Components](#).

## Signing requirements by version

The following table shows signing policies for client operating system versions.

Note that Secure Boot does not apply to Windows Vista and Windows 7.

APPLIES TO:	WINDOWS VISTA, WINDOWS 7; WINDOWS 8+ WITH SECURE BOOT OFF	WINDOWS 8, WINDOWS 8.1, WINDOWS 10, VERSIONS 1507, 1511 WITH SECURE BOOT ON	WINDOWS 10, VERSIONS 1607, 1703, 1709 WITH SECURE BOOT ON	WINDOWS 10, VERSION 1803+ WITH SECURE BOOT ON
Architectures:	64-bit only, no signature required for 32-bit	64-bit, 32-bit	64-bit, 32-bit	64-bit, 32-bit
Signature required:	Embedded or catalog file	Embedded or catalog file	Embedded or catalog file	Embedded or catalog file
Signature algorithm:	SHA2	SHA2	SHA2	SHA2
Certificate:	Standard roots trusted by Code Integrity	Standard roots trusted by Code Integrity	Microsoft Root Authority 2010, Microsoft Root Certificate Authority, Microsoft Root Authority	Microsoft Root Authority 2010, Microsoft Root Certificate Authority, Microsoft Root Authority

In addition to driver code signing, you also need to meet the PnP device installation signing requirements for installing a driver. For more info, see [Plug and Play \(PnP\) device installation signing requirements](#).

For info about signing an ELAM driver, see [Early launch antimalware](#).

## Signing a driver for internal distribution only

In some cases, you may want to distribute a driver internally within a company rather than via Windows Update. To do this without requiring that computers running it are in test mode, use the following procedure:

1. [Register for the Hardware Dev Center](#).
2. Review the [Hardware dashboard FAQ](#) and sign appropriate agreements.
3. Upload codesign certificates.
4. Sign drivers locally using a non-EV codesign certificate.
5. Package drivers in a CAB and sign the CAB using the above codesign certificate.
6. Submit the CAB to the Hardware Dev Center for signing.
7. If the submission is approved, the Hardware Dev Center returns the driver with a Microsoft signature.
8. Distribute the driver internally.

## See Also

- [Installing an Unsigned Driver Package during Development and Test](#)
- [Signing Drivers for Public Release](#)

- Signing Drivers during Development and Test
- Digital Signatures
- Troubleshooting Install and Load Problems with Signed Driver Packages

# Windows 10 in S mode Driver Requirements

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section describes driver installation requirements and blocked components on Windows 10 S.

## Driver Requirements

To install on Windows 10 in S mode, driver packages must meet the following requirements:

- Driver packages must be digitally signed with a **Windows, WHQL, ELAM, or Store** certificate from the [Windows Hardware Developer Center Dashboard](#).
- Companion software must be signed with a [Microsoft Store Certificate](#).
- Does not include an \*.exe, \*.zip, \*.msi or \*.cab in the driver package that extracts unsigned binaries.
- Driver installs using only INF directives.
- Driver does not call [blocked inbox components](#).
- Drivers does not include any user interface components, apps, or settings. Instead, use Universal applications from the Microsoft Store, for example:
  - [Hardware Support Apps](#)
  - [UWP device apps](#)
  - [Centennial Apps](#)
- Driver and firmware servicing uses Windows Update and not an updater app.

Finally, we recommend using a Universal Windows driver where possible. For more info, see:

- [Getting Started with Universal Drivers](#)
- [Validating Universal Drivers](#)

## Installation

- If you check the S compliance checkboxes when submitting a driver in the dashboard, the driver is delivered to both Windows 10 in S mode as well as desktop versions of Windows 10 that have the same HW ID. For more info about these dashboard options, see [Publish a driver to Windows Update](#).
- If different driver packages are required for Windows 10 in S mode and desktop versions of Windows 10 that target the same HWID, set a greater **DriverVer** entry in the [INF Version Section](#) for the package that targets desktop versions of Windows 10. For example, you might set a **DriverVer** of `05/24/2019,10.0.1.0` for the package targeting Windows 10 in S mode, and `05/24/2019,10.1.1.0` for the package targeting desktop versions of Windows 10.

## Troubleshooting installation

If you are targeting Windows 10 in S mode for both a base INF and an extension INF, but only the extension INF is installing on desktop versions of Windows 10, then either your installed driver is of greater rank, or your base driver was not published with the correct targeting. (CHID may be different). Check and compare your Shipping Label of the BASE driver and Extension driver.

## Blocked inbox components

The following components are blocked from executing on Windows 10 S:

- bash.exe
- cdb.exe
- cmd.exe
- cscript.exe
- csi.exe
- dnx.exe
- fsi.exe
- hh.exe
- infdefaultinstall.exe (new addition for Windows 10, version 1709)
- kd.exe
- lxssmanager.exe
- msbuild.exe
- mshta.exe
- ntsd.exe
- powershell.exe
- powershell\_ise.exe
- rcsi.exe
- reg.exe
- regedit.exe
- regedt32.exe
- regini.exe
- syskey.exe
- wbemtest.exe
- windbg.exe
- wmic.exe
- wscript.exe
- wsl.exe

[!NEXT] To ensure your Windows app will operate correctly on devices that run Windows 10 in S mode, please review the [test guidance](#) for apps.

# Managing the Signing Process

11/2/2020 • 2 minutes to read • [Edit Online](#)

The process of signing a [driver package's catalog file](#) or embedding a signature in a driver file involves the following activities.

## Driver Signing Administration

This is typically handled by the publisher's program management and software release services and includes the following topics:

- [Selecting the appropriate signature type](#)
- [Obtaining the necessary release certificate or test certificate](#)
- [Managing the digital signature or code signing keys](#)

## Driver Development

This is typically handled by the publisher's development team and includes the following:

- Implementing the driver.
- Creating a signed driver package for internal testing. For information about test signing, see [Signing Drivers during Development and Test](#).
- Creating a signed driver package for release. For information about how to sign drivers for release, see [Signing Drivers for Public Release](#).

# Kernel-Mode Code Signing Requirements

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, the kernel-mode code signing policy controls whether a kernel-mode driver will be loaded. The signing requirements depend on the version of the Windows operating system and on whether the driver is being signed for public release or by a development team during the development and test of a driver. There are also [signing requirements](#) that pertain to the installation of a PnP device and driver.

Virtual drivers have the same requirements as actual hardware drivers. In other words, they must comply with the requirements for the OS version for which they are targeted.

For info about signing and dashboard submission, see [Get drivers signed by Microsoft for multiple Windows versions](#).

## Kernel-Mode Code Signing Requirements for Public Release of a Driver

### NOTE

Starting with Windows 10, version 1607, Windows will not load any new kernel mode drivers which are not signed by the Microsoft through the [Hardware Dev Center](#). Valid signatures can be obtained by either [Hardware Certification](#) or [Attestation](#).

### 64-bit versions of Windows starting with Windows Vista

The kernel-mode code signing policy requires that a kernel-mode driver be signed as follows:

- A kernel-mode boot-start driver must have an embedded [Software Publisher Certificate \(SPC\)](#) signature. This applies to any type of PnP or non-PnP kernel-mode boot-start driver.
- A non-PnP kernel-mode driver that is not a boot-start driver must have either a [catalog file](#) with an SPC signature or the driver file must include an embedded SPC signature.
- A PnP kernel-mode driver that is not a boot-start driver must have either an embedded SPC signature, a catalog file with a [WHQL release signature](#), or a catalog file with an SPC signature. Although the kernel-mode code signing policy does not require that the catalog file of a PnP driver be signed, PnP device installation treats a driver as signed only if the catalog file of the driver is also signed.

### 32-bit versions of Windows

Windows Vista and later versions of Windows enforce the kernel-mode driver signing policy only for the following drivers:

- Drivers that stream protected media. For more information about these requirements, see [Code-signing for Protected Media Components \(Windows Vista and Later\)](#)
- Kernel-mode *boot-start drivers*.

## Kernel-Mode Code Signing Requirements during Development and Test

### 64-bit versions of Windows starting with Windows Vista

The kernel-mode code signing policy requires that a kernel-mode driver be [test-signed](#) and that test-signing is [enabled](#). A test signature can be a [WHQL test signature](#) or generated in-house by a [test certificate](#). Drivers must be test-signed as follows:

- A kernel-mode *boot-start driver* must have an embedded test signature. This applies to any type of PnP or non-PnP kernel-mode driver.

- A kernel-mode driver that is not a *boot-start driver* must have either a test-signed [catalog file](#) or the driver file must include an embedded test signature. This applies to any type of PnP or non-PnP kernel-mode driver.

### 32-bit versions of Windows

Windows Vista and later versions of Windows enforce the kernel-mode driver signing policy only for the following drivers:

- Drivers that stream protected media. For more information about these requirements, see [Code-signing for Protected Media Components \(Windows Vista and Later\)](#)
- Kernel-mode *boot-start drivers*.

# PnP Device Installation Signing Requirements

11/2/2020 • 2 minutes to read • [Edit Online](#)

The driver signing requirements for Plug and Play (PnP) device installation depend on the version of Windows and on whether the driver is being signed for public release or by a development team during the development and test of the driver. All 64-bit versions of Windows enforce [kernel-mode code signing requirements](#) that determine whether a kernel-mode driver can be loaded.

## PnP Signing Requirements for Public Release of a Driver

The [Windows Hardware Lab Kit \(Windows HLK\)](#) has [test categories](#) for a variety of device types. If a test category for the device type is included in this list, you should obtain a [WHQL release signature](#).

A valid WHQL release signature verifies that the driver complies with the requirements of the HCK, verifies the identity of the publisher, and verifies that the driver has not been altered.

To be considered signed by PnP device installation, the [catalog file](#) of the [driver package](#) must be signed by WHQL or signed by a third-party [release certificate](#) (a [Software Publisher Certificate \(SPC\)](#) or a commercial release certificate). A WHQL release signature should be used if one can be obtained. A third-party release signature verifies the identity of the publisher and that the driver has not been altered. However, unlike a WHQL release signature, a third-party release signature does not verify driver functionality.

Also be aware that for 64-bit versions of Windows Vista and later versions of Windows, the kernel-mode code signing policy further requires that a kernel-mode driver be signed by WHQL or by an SPC.

For more information about release-signing, see [Signing Drivers for Public Release](#).

## PnP Signing Requirements for Development and Test of a Driver

In 64-bit versions of Windows Vista and later versions of Windows, a driver must have a [WHQL test signature](#) or must be signed by a [test certificate](#). For more information about test-signing drivers, see [Signing Drivers during Development and Test](#).

# Authenticode Signing of Third-party CSPs

11/2/2020 • 2 minutes to read • [Edit Online](#)

Third-party Authenticode signing for custom Cryptographic Service Providers (CSPs) has been available beginning with Windows Vista, and has been back ported to Windows XP SP3 and Windows Server 2003 SP2 as of May, 2013 via [this download](#).

Consequently, Microsoft will no longer sign CSPs, and the manual CSP signing service has been retired. Emails and CSPs sent to [cspsign@microsoft.com](mailto:cspsign@microsoft.com) or [cecpsig@microsoft.com](mailto:cecpsig@microsoft.com) to be signed will no longer be processed by Microsoft.

Instead, all third-party CSPs can now be self-signed by following this procedure:

1. Purchase a code-signing certificate from a Certificate Authority (CA) for which Microsoft also issues a cross-certificate. The [Cross-Certificates for Kernel Mode Code Signing](#) topic provides a list of CAs for which Microsoft also provides cross-certificates and the corresponding cross-certificates. Note that these are the only cross-certificates that chain up to the "Microsoft Code Verification Root" issued by Microsoft, which will enable Windows to run third-party CSPs.
2. After you have a certificate from the CA and the matching cross-certificate, you can use [SignTool](#) to sign all your CSP binaries.
3. SignTool is included in the latest versions of Visual Studio. It is also included in the WDK version 7.0 and newer versions. Note that the SignTool that ships with earlier versions of the WDK is not compatible with cross-certificates and cannot be used to sign your binaries.

## NOTE

Starting with Windows 8, it is no longer a requirement that CSPs must be signed.

You can sign binaries from a command-line, or sign as an integrated build step in Visual Studio 2012 and newer.

The command to [SignTool](#) is:

```
signtool.exe sign /ac <cross-certificate_from_ms> /sha1 <sha1_hash> /t <timestamp_server> /d  
<"optional_description_in_double_quotes"> <binary_file.ext>
```

- <cross-certificate\_from\_ca> is the cross-certificate file you downloaded from Microsoft
- <sha1\_hash> is the SHA1 thumbprint that corresponds to the code signing certificate
- <timestamp\_server> is the server used to timestamp the signing operation
- <"optional\_description\_in\_double\_quotes"> is an optional friendly-name description
- <binary\_file.ext> is the file to sign

For example:

```
signtool.exe sign /ac certificate.cer /sha1 553e39af9e0ea8c9edcd802abbf103166f81fa50 /t  
"http://timestamp.digicert.com" /d "My Cryptographic Service Provider" csp.dll
```

**NOTE**

It is unnecessary to include resource ID #666 in the CSP DLL, or the signature in the registry, as was required for older CSP signatures.

## Additional Help and Support

Consult the [Troubleshooting and support](#) page for additional help and support.

You can also check the [Application Security for Windows Desktop](#) forum for assistance.

# Managing the Digital Signature or Code Signing Keys

12/5/2018 • 2 minutes to read • [Edit Online](#)

The cryptographic keys that are at the heart of the Authenticode signing process must be well protected and treated with the same care as the publisher's most valuable assets. These keys represent an organization's identity. Any code that is signed with these keys appears to Windows as if it contains a valid digital signature that can be traced to the organization. If the keys are stolen, they could be used to fraudulently sign malicious code and possibly result in the delivery of code that contains Trojan or a virus that appears to come from a legitimate publisher.

# Cross-Certificates for Kernel Mode Code Signing

11/2/2020 • 6 minutes to read • [Edit Online](#)

This information describes how to obtain and use cross-certificates for code-signing kernel-mode binary files for Microsoft Windows.

## NOTE

Please review Microsoft Security Advisory ([2880823](#)) "Deprecation of SHA-1 Hashing Algorithm for Microsoft Root Certificate Program" which describes a policy change wherein Microsoft will no longer allow root certificate authorities to issue X.509 certificates using the SHA-1 hashing algorithm for the purposes of SSL and code signing after January 1, 2016.

## NOTE

The [Microsoft Trusted Root Program](#) no longer supports root certificates that have kernel mode signing capabilities. For more info, see [Deprecation of Software Publisher Certificates, Commercial Release Certificates, and Commercial Test Certificates](#).

## Cross-Certificates Overview

A cross-certificate is a digital certificate issued by one Certificate Authority (CA) that is used to sign the public key for the root certificate of another Certificate Authority. Cross-certificates provide a means to create a chain of trust from a single, trusted, root CA to multiple other CAs.

In Windows, cross-certificates:

- Allow the operating system kernel to have a single trusted Microsoft root authority.
- Extend the chain of trust to multiple commercial CAs that issue Software Publisher Certificates (SPCs), which are used for code-signing software for distribution, installation, and loading on Windows

The cross-certificates that are provided here are used with the Windows Driver Kit (WDK) code-signing tools for properly signing kernel-mode software. Digitally signing kernel-mode software is similar to code-signing any software that is published for Windows. Cross-certificates are added to the digital signature by the developer or software publisher when signing the kernel-mode software. The cross-certificate itself is added by the code-signing tools to the digital signature of the binary file or catalog.

See [Authenticode Signing of Third-party CSPs](#) for more information about how to use cross-certificates to sign third-party Cryptographic Service Providers (CSPs).

## Selecting the Correct Cross-certificate

Microsoft provides a specific cross-certificate for each CA that issues SPCs for code-signing kernel-mode code. The list below has a link to the correct cross-certificate for the root authority that issued your SPC.

Follow the steps below to identify your CA, and then download the related cross-certificate.

1. Open the Microsoft Management Console (MMC) and add the Certificates snap-in:
  - a. Select the Start button, type "mmc" in the search box, and select mmc from the search results. If a User Account Control dialog box appears, select Yes.
  - b. From the MMC File menu, select Add/Remove Snap-in, ...

- c. Select the Certificates snap-in and select Add.
  - d. Select My user account and select Finish.
  - e. Select the Certificates snap-in again and select Add.
  - f. Select Computer account and select Next.
  - g. Select Local computer and select Finish.
2. Locate your SPC in the certificate store, and then double-click it. Your certificate is listed in one of the following two locations, depending on how the certificate was installed.
- The Current User, Personal, Certificates store, or
  - The Local Computer, Personal, Certificates store
3. In the **Certificate** dialog box, select the **Certification Path** tab, and then select the top-most certificate in the certification path. This is the CA that is the issuing root authority for your SPC.
4. View the root authority certificate by selecting the **View Certificate** button, and then select the **Details** tab of the new **Certificate** dialog box.
5. Find the **Issuer** and **Thumbprint** for this certificate. Then locate the corresponding entry for this CA in the list below.
6. Download the related cross-certificate for the CA and use this cross-certificate together with your SPC when digitally signing kernel-mode code

## Cross-Certificate List

The following list contains all of the CAs that are currently supported by Microsoft for issuing SPCs for code-signing kernel-mode code.

CA	ROOT CERTIFICATE THUMBPRINT	EXPIRATION DATE	DOWNLOAD LINK
Certum Trusted Network CA	55 43 55 15 fd d2 48 65 75 fd c5 cf 3b ad 00 c9 13 12 3d 03	2021/04/15	<a href="#">Download</a>
DigiCert Assured ID Root CA	ba 3e a5 4d 72 c1 45 d3 7c 25 5e 1e a4 0a fb c6 33 48 b9 6e	2021/04/15	<a href="#">Download</a>
DigiCert Global Root CA	c9 83 39 19 f1 f3 6a 63 48 11 1e 93 02 6f d4 0e b9 6f bc 34	2021/04/15	<a href="#">Download</a>
DigiCert High Assurance EV Root CA	2f 25 13 af 39 92 db 0a 3f 79 70 9f f8 14 3b 3f 7b d2 d1 43	2021/04/15	<a href="#">Download</a>
Entrust.net Certification Authority (2048)	00 a3 e6 00 9e aa 73 9b 3d ee f4 b5 06 64 9d 8a 1a 7a d3 3a	2021/04/15	<a href="#">Download</a>
Entrust Root Certification Authority – G2	d8 fc 24 87 48 58 5e 17 3e fb fb 30 75 c4 b4 d6 0f 9d 8d 08	2025/07/07	<a href="#">Download</a>

CA	ROOT CERTIFICATE THUMPRINT	EXPIRATION DATE	DOWNLOAD LINK
GeoTrust Primary Certification Authority	e8 6e 80 82 99 0e 3d fa ed 81 6d 9e b1 72 0f 91 a4 f1 a1 85	2021/02/22	<a href="#">Download</a>
GeoTrust Primary Certification Authority – G3	b2 bb bd fa c8 f1 a8 ad 58 95 cd 49 38 4b 22 ca 19 db 2d 1f	2021/02/22	<a href="#">Download</a>
GlobalSign Root CA	cc 1d ee bf 6d 55 c2 c9 06 1b a1 6f 10 a0 bf a6 97 9a 4a 32	2021/04/15	<a href="#">Download</a>
Go Daddy Root Certificate Authority – G2	84 2c 5c b3 4b 73 bb c5 ed 85 64 bd ed a7 86 96 7d 7b 42 ef	2021/04/15	<a href="#">Download</a>
NetLock Arany (Class Gold)	89 4f 1d 28 97 aa 4c 07 4d cd 85 c5 fc 09 ee 73 b9 51 04 d8	2021/04/15	<a href="#">Download</a>
NetLock Platina (Class Platinum)	97 dd 74 97 16 20 57 29 41 dc 80 0c 2f d8 0a 48 07 7d 10 b0	2021/04/15	<a href="#">Download</a>
Security Communication RootCA1	41 f2 8c e5 6f d8 b9 cb 46 7f b5 03 2a 3c ae 1c dc 9d 86 48	2021/04/15	<a href="#">Download</a>
Starfield Root Certificate Authority – G2	40 c2 0a 9a 33 fa d0 36 ac bf e8 2d 6c bb ee 1b 42 9b 86 de	2021/04/15	<a href="#">Download</a>
StartCom Certification Authority	e6 06 9e 04 8d ea 8d 81 7a fc 41 88 b1 be f1 d8 88 d0 af 17	2021/04/15	<a href="#">Download</a>
TC TrustCenter Class 2 CA II	42 62 ff 7d 89 70 66 aa e7 75 80 d3 3a d2 88 03 f9 a1 1a 62	2021/04/11	<a href="#">Download</a>
Thawte Primary Root CA	55 38 e9 fe c1 40 30 b7 40 15 23 49 e1 15 a1 16 5d 29 07 4a	2021/02/22	<a href="#">Download</a>
Thawte Primary Root CA – G3	ba 57 ca 5e 78 dd 2d 1d 74 76 ae be e9 95 3e 39 6f d0 55 46	2021/02/22	<a href="#">Download</a>
VeriSign Class 3 Public Primary Certification Authority – G5	57 53 4c cc 33 91 4c 41 f7 0e 2c bb 21 03 a1 db 18 81 7d 8b	2021/02/22	<a href="#">Download</a>
VeriSign Universal Root Certification Authority	9e d8 cd 56 01 f0 10 56 51 eb bb 3f 57 f0 31 82 e5 fa 7e 01	2021/02/22	<a href="#">Download</a>

## New Cross-Certificate List

The following list contains several new CAs that are currently supported by Microsoft for issuing SPCs for code-signing kernel-mode code.

CA	ROOT CERTIFICATE THUMPRINT	EXPIRATION DATE	DOWNLOAD LINK
AddTrust External CA Root	a7 5a c6 57 aa 7a 4c df e5 f9 de 39 3e 69 ef ca b6 59 d2 50	2023/08/15	<a href="#">Download</a>
GoDaddy Class 2 Certification Authority	d9 61 24 72 ef 0f 27 87 e2 b2 d9 e0 63 a0 6b 32 fa 5e 33 3d	2023/08/27	<a href="#">Download</a>
Starfield Class 2 Certification Authority	f8 fc 7f 3c dd 51 76 ad d2 7c f9 7f 73 96 59 09 46 6d 9a 22	2023/08/27	<a href="#">Download</a>
UTN-USERFirst-Object	ae 1e 25 26 01 30 a3 0b 1b c2 20 29 35 65 3b e5 a7 23 be f5	2023/08/15	<a href="#">Download</a>

# Introduction to Test-Signing

11/2/2020 • 2 minutes to read • [Edit Online](#)

Drivers should be test-signed with a [digital signature](#) during development and test for the following reasons:

- To facilitate and automate installation.

If a driver is not signed, the [Plug and Play \(PnP\) driver installation policy](#) of Windows Vista and later versions of Windows require that a system administrator manually authorize the installation of an unsigned driver, adding an extra step to the installation process. This extra step can adversely affect the productivity of developers and testers. This requirement cannot be overridden.

- To be able to load kernel-mode drivers on 64-bit versions of Windows Vista and later versions of Windows.

By default, the [kernel-mode code signing policy](#) for 64-bit versions of Windows Vista and later versions of Windows require that a kernel-mode driver be signed in order for the driver to be loaded. This requirement can be temporarily overridden to facilitate the development or debugging of a driver.

- To play back certain types of next-generation premium content, all kernel-mode components in Windows Vista and later versions of Windows must be signed. In addition, all the user-mode and kernel-mode components in the Protected Media Path (PMP) must comply with PMP signing policy. For information about PMP signing policy, see the white paper [Code-signing for Protected Media Components in Windows Vista](#).

For these reasons, drivers for Windows Vista and later versions of Windows should be test-signed with a digital certificate that is created by using Microsoft Authenticode. Such a digital certificate is referred to as a *test certificate* and a signature generated with a test certificate is referred to as a *test signature*.

**Note** Windows Vista and later versions of Windows support test-signed drivers only for development and testing purposes. Test signatures must not be used for production purposes or released to customers.

A development and test team can participate in the [WHQL Test Signature program](#), where the Windows Hardware Quality Labs (WHQL) will sign PnP [driver packages](#) for testing purposes. Alternatively, a development and test team can manage their own in-house signing process and use the following types of [test certificates](#) to test-sign drivers:

- [MakeCert test certificate](#), which is a digital certificate created by the [MakeCert](#) tool.
- [Commercial test certificate](#), which is a digital certificate that is issued by a CA that is a member of the Microsoft Root Certificate Program.
- [Enterprise CA test certificate](#), which is a digital certificate that is deployed by an Enterprise CA.

For information about how a test team signs a driver package after the team creates, obtains, or is provided a test certificate, see [Test-Signing Driver Packages](#).

For information about how to install driver packages that are test-signed, see [Installing Test-Signed Driver Packages](#).

To facilitate early driver development and debugging, you can temporarily disable the kernel-mode code signing requirement to load and test an unsigned kernel-mode driver. However, you cannot disable the PnP driver installation policy that requires a system administrator to authorize the installation of an unsigned driver. For more information about how to install an unsigned driver, see [Installing an Unsigned Driver during Development](#)

and Test.

For information about the most appropriate tools to use to test-sign driver packages, see [Tools for Signing Drivers](#).

**Note** To get a better understanding of the steps that are involved in test-signing driver packages, see [How to Test-Sign a Driver Package](#). This topic provides a summary of the test-signing process, and steps through many examples of test-signing by using the *ToastPkg* sample [driver package](#) within the Windows Driver Kit (WDK).

# How to Test-Sign a Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section provides information about the basic steps that you have to follow when you test-sign a [driver package](#).

Test-signing refers to using a test certificate to sign a prerelease version of a [driver package](#) for use on test computers. In particular, this allows developers to sign kernel-mode binaries by using self-signed certificates, such as those the [MakeCert](#) tool generates. This capability allows developers to test kernel-mode binaries on Windows with driver signature verification enabled.

Windows supports test-signed drivers only for development and testing purposes. Test-signed drivers must not be used for production purposes or released to customers.

This section includes topics that describe these steps and provide examples, such as the following:

- Creating a test certificate that is used to sign a driver package. In this section, steps are described to create and use a self-signed test certificate named *Contoso.com/Test*. This certificate is used in many examples that are discussed in this section.
- Preparing a [driver package](#) for test-signing. This includes creating a [catalog file](#) that contains the digital signature.
- Test-signing the driver package's catalog file by using the *Contoso.com/Test* certificate.
- Test-signing a driver through an embedded signature by using the *Contoso.com/Test* certificate.

**Note** You have to embed a digital signature within the driver if the driver is a *boot-start driver*.

Each topic in this section describes a separate procedure in the test-signing process, and provides the general information that you need to understand the procedure. In addition, each topic points you to other topics that provide detailed information about the procedure.

Throughout this section, separate computers are used for the various processes involved in test-signing a driver. These computers are referred to as follows:

## Signing computer

This is the computer that is used to test-sign a driver package for Windows Vista and later versions of Windows. This computer must be running Windows XP SP2 or later versions of Windows. In order to use the [driver signing tools](#), this computer must have the Windows Vista and later versions of the Windows Driver Kit (WDK) installed.

## Test computer

This is the computer that is used to install and test the test-signed driver package. This computer must be running Windows Vista or later versions of Windows.

The topics of this section use the *ToastPkg* sample driver package to introduce the test-signing process. Within the WDK installation directory, the *ToastPkg* driver package is located in the *src\general\toaster\toastpkg* directory.

**Note** The WDK contains a sample command script that shows the step-by-step procedure to correctly test-sign the *ToastPkg* sample [driver package](#). You can modify this script to test-sign your own driver package. Within the WDK installation directory, the example is located at *src\general\build\driversigning\selfsign\_example.cmd*. Additional instructions for test-signing are described in *src\general\build\driversigning\selfsign\_readme.htm*.

This section includes the following topics:

[Creating Test Certificates](#)

[Viewing Test Certificates](#)

[Creating a Catalog File for Test-Signing a Driver Package](#)

[Test-Signing a Driver Package's Catalog File](#)

[Test-Signing a Driver through an Embedded Signature](#)

[Configuring the Test Computer to Support Test-Signing](#)

[Installing Test Certificates](#)

[Verifying the Test Signature](#)

[Installing a Test-Signed Driver Package on the Test Computer](#)

# How to Install a Test-signed Driver Required for Windows Setup and Boot

11/2/2020 • 3 minutes to read • [Edit Online](#)

This page describes how to install a test-signed driver on a computer running Windows Server 2019 (or Windows Server 2016), or on a computer starting for the first time after Windows Setup. You should only use a test-signed driver in a test environment.

For more info, see [Introduction to Test-Signing](#).

Before you begin, ensure that you have:

- [Windows Assessment and Deployment Kit \(ADK\)](#) and Windows PE add-on for the ADK
- Windows Server 2019 or 2016 Installation Media ISO file

## Creating the ISO file

Use the following steps to create an ISO file and install Windows from it:

1. In the ADK Start Menu options, choose **Deployment and Imaging Tools Environment**, right-click, and select **Run as administrator**.
2. Run **copype** to create a working copy of the Windows PE files: `copype amd64 C:\WinPE_amd64`
3. Enable **testsigning**. On a non-UEFI (legacy) computer, use:

```
cd C:\WinPE_amd64\media\Boot  
bcdedit /store .\BCD /enum all  
bcdedit /store .\BCD /set {default} testsigning on
```

On a UEFI platform, use:

```
cd C:\WinPE_amd64\media\EFI\Microsoft\Boot  
bcdedit /store .\BCD /enum all  
bcdedit /store .\BCD /set {default} testsigning on
```

4. To verify that `testsigning Yes` now appears for the {default} identifier, under Windows Boot Loader, run `bcdedit /store .\BCD /enum all` a second time.
5. Mount the Windows Server 2016 Installation Media ISO file to a drive, for example, `G:`, and manually copy all files under the sources folder, for example `G:\sources`, to the sources folder of the WinPE system files, for example `C:\WinPE_amd64\media\sources`.

### NOTE

Do not overwrite the existing `boot.wim` file in the folder `C:\WinPE_amd64\media\sources`. We'll use the original WinPE environment later.

Now we have all the files including WinPE and Windows Server 2016.

6. Optionally copy a test-signed driver to the folder `C:\WinPE_amd64\media`. Files copied might include the driver's .cat, .cer, .inf, and .sys files. Use the following commands to import the test-signed driver to the WIM file:

```
Dism /Get-WimInfo /wimfile:C:\WinPE_amd64\media\sources\install.wim
Dism /Mount-Image /imagefile:C:\WinPE_amd64\media\sources\install.wim /index:4 /mountdir:C:\WinPE_amd64\mount
Dism /image:C:\WinPE_amd64\mount /Add-Driver /driver:C:\WinPE_amd64\media\DriverSample
Dism /unmount-image /mountdir:C:\WinPE_amd64\mount /commit
```

7. Create a new ISO file: `Makewinpemedia /iso C:\winpe_amd64 C:\WS2016_amd64.iso`. While the default application in the ISO file is the cmd.exe, you'll launch the setup.exe manually to configure boot settings after installation.
8. Install Windows Server 2016 from `WS2016_amd64.iso`. Optionally, customize the installation source to import more drivers.

## Installing the driver

Use these steps to install the driver:

1. Turn off **Secure Boot** on the test computer and then start the WinPE system.
2. After the machine boots with the ISO file, a command prompt appears.
3. To identify the letter of the drive with the mounted ISO file, use `diskpart`, then `list volume`. Find the volume with **Type** of `DVD-ROM`. Type `exit`.
4. Navigate to the ISO drive and switch to the driver sample directory, for example `D:\DriverSample`.
5. Use the following commands to install the test driver:

```
certmgr.exe -add DriverSample.cer -s -r localmachine root
certmgr.exe -add DriverSample.cer -s -r localmachine trustedpublisher
devcon.exe install DriverSample.inf Root\DriverSample
```

6. Optionally, confirm the installation by reviewing the `setupapi.dev` log.
7. Run `setup.exe /NoReboot`, for example from `D:\sources`.
8. After installation, a message appears indicating that the setup application can be closed. Exit the application to return to the WinPE command prompt.
9. Type `diskpart`. Identify the OS boot partition and the drive letter for that boot partition (The only FAT32 partition and the size is about 100MB)
10. Navigate to the boot partition drive and switch directory to the location of the BCD file, for example `E:\EFI\Microsoft\Boot`.
11. Turn on **testsigning**: `bcdedit /store BCD /set {default} testsigning on` and reboot the computer.
12. To confirm that the computer is in test mode, look for a **Test Mode** watermark in the lower right of the desktop.

The computer must be in Test Mode to load a test-signed driver. If there is a boot device requiring the test-signed driver, the test-signed driver must be imported to the WIM file (use the optional Dism steps above) to avoid PnP installation later. If you turn off the **testsigning** setting, the machine may fail to boot.

# Creating Test Certificates

11/2/2020 • 2 minutes to read • [Edit Online](#)

Test-signing requires a test certificate. After a test certificate is generated, it can be used to test-sign multiple drivers or [driver packages](#). For more information, see [Test Certificates](#).

This topic describes how to use the [MakeCert](#) tool to create test certificates. In most development environments, test certificates generated through MakeCert should be sufficient to test the installation and loading of test-signed drivers or driver packages. For more information about this type of test certificate, see [MakeCert Test Certificate](#).

The following command-line example uses MakeCert to complete the following tasks:

- Create a self-signed test certificate named *Contoso.com/Test*. This certificate uses the same name for the subject name and the certificate authority (CA).
- Put a copy of the certificate in an output file that is named *ContosoTest.cer*.
- Put a copy of the certificate in a certificate store that is named *PrivateCertStore*. Putting the test certificate in *PrivateCertStore* keeps it separate from other certificates that may be on the system.

Use the following MakeCert command to create the *Contoso.com/Test* certificate:

```
makecert -r -pe -ss PrivateCertStore -n CN=Contoso.com(Test) -eku 1.3.6.1.5.5.7.3.3 ContosoTest.cer
```

Where:

- The **-r** option creates a self-signed certificate with the same issuer and subject name.
- The **-pe** option specifies that the private key that is associated with the certificate can be exported.
- The **-ss** option specifies the name of the certificate store that contains the test certificate (*PrivateCertStore*).
- The **-n CN=** option specifies the name of the certificate, *Contoso.com/Test*. This name is used with the [SignTool](#) tool to identify the certificate.
- The EKU option inserts a list of one or more comma-separated, [enhanced key usage](#) object identifiers (OIDs) into the certificate. For example, `-eku 1.3.6.1.5.5.7.3.2` inserts the client authentication OID. For definitions of allowable OIDs, see the *Wincrypt.h* file in CryptoAPI 2.0.
- *ContosoTest.cer* is the file name that contains a copy of the test certificate, *Contoso.com/Test*. The certificate file is used to add the certificate to the Trusted Root Certification Authorities certificate store and the Trusted Publishers certificate store.

The certificate store that contains the test certificate is added to the list of certificate stores that Windows manages for the user account on the development computer on which the certificate store was created.

A developer has to create only one MakeCert test certificate to sign all [driver packages](#) on a development computer.

For more information about the MakeCert tool and its command-line arguments, see [MakeCert](#).

**NOTE**

After creating a test certificate, use the CertMgr tool to add it to the Trusted Root Certification Authorities certificate store. For more info, see [Installing Test Certificates](#).

Also refer to the readme file `Selfsign_readme.htm` in the `bin\selfsign` directory of the Windows Driver Kit (WDK).

# Installing Test Certificates

11/2/2020 • 2 minutes to read • [Edit Online](#)

To successfully install a test-signed [driver package](#) on a test computer, the computer must be able to verify the signature. To do that, the test computer must have the certificate for the certificate authority (CA) that issued the package's test certificate installed in the computer's Trusted Root Certification Authorities certificate store.

The CA certificate must be added to the Trusted Root Certification Authorities certificate store only once. Once added, it can then be used to verify the signature of all drivers or driver packages, which were digitally signed with the certificate, before the driver package is installed on the computer.

The simplest way to add a test certificate to the Trusted Root Certification Authorities certificate store is through the [CertMgr](#) tool. This topic will describe the procedure for installing the test certificate, Contoso.com(test). This certificate is stored within the *ContosoTest.cer* file. For more information about how this certificate was created, see [Creating Test Certificates](#).

The following command-line uses Certmgr.exe to install, or add, the Contoso.com(test) certificate to the test computer's Trusted Root Certification Authorities certificate store:

```
certmgr /add ContosoTest.cer /s /r localMachine root
```

Where:

- The /add option specifies that the certificate in the *ContosoTest.cer* file is to be added to the specified certificate store.
- The /s option specifies that the certificate is to be added to a system store.
- The /r option specifies the system store location, which is either *currentUser* or *localMachine*.
- *Root* specifies the name of the destination store for the local computer, which is either *root* to specify the Trusted Root Certification Authorities certificate store or *trustedpublisher* to specify the Trusted Publishers certificate store.

A successful run produces the following output:

```
certmgr /add ContosoTest.cer /s /r localMachine root
CertMgr Succeeded
```

After the certificate is copied to the Trusted Root Certification Authorities certificate store (the local machine's root store, *not* the user store), you can view it through the Microsoft Management Console (MMC) Certificates snap-in, as described in [Viewing Test Certificates](#).

The following screenshot shows the Contoso.com(Test) certificate in the Trusted Root Certification Authorities certificate store.

Actions
Certificates
More ...
Contoso.co... More ...

Issued To	Issued By	Expiration Date
Class 3 Public Primary Certificate...	Class 3 Public Primary Certificate...	8/1/2028
Class 3 Public Primary Certificate...	Class 3 Public Primary Certificate...	1/7/2004
Contoso.com\Test	Contoso.com\Test	12/31/2039
Copyright (c) 1997 Microsoft C...	Copyright (c) 1997 Microsoft Corp.	12/30/1999
GTE CyberTrust Global Root	GTE CyberTrust Global Root	8/13/2018
GTE CyberTrust Root	GTE CyberTrust Root	2/23/2006
GTE CyberTrust Root	GTE CyberTrust Root	2/23/2006
http://www.valicert.com/	http://www.valicert.com/	6/25/2019
Microsoft Authenticode(tm) Ro...	Microsoft Authenticode(tm) Root...	12/31/1999
Microsoft Corporate Root Auth...	Microsoft Corporate Root Authority	12/14/2017
Microsoft Corporate Root Auth...	Microsoft Corporate Root Authority	2/24/2008
Microsoft Corporate Root Auth...	Microsoft Corporate Root Authority	12/14/2017
Microsoft Corporate Root CA	Microsoft Corporate Root CA	9/19/2019
Microsoft Root Authority	Microsoft Root Authority	12/31/2020
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...	5/9/2021

You can also view the certificate at the command prompt:

```
certutil -store root | findstr Contoso
certutil -store root <SHA-1 id of certificate>
```

Or, from PowerShell:

```
Get-ChildItem -path cert: \LocalMachine\My | findstr Contoso
```

The Certmgr.exe tool is part of the Windows SDK and is typically installed to

```
C:\Program Files (x86)\Windows Kits\10\bin\<build>\x86\certmgr.exe .
```

For more information about CertMgr and its command-line arguments, see [CertMgr](#).

For more information about how to install test certificates, see [Installing a Test Certificate on a Test Computer](#).

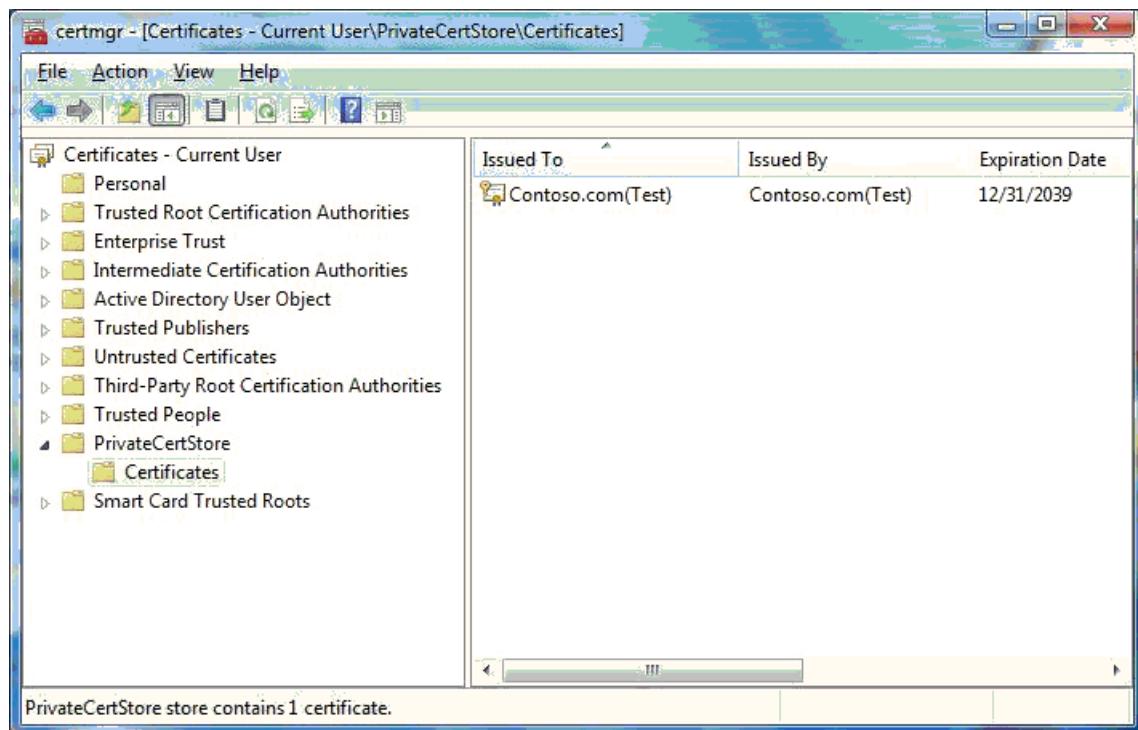
# Viewing Test Certificates

12/5/2018 • 2 minutes to read • [Edit Online](#)

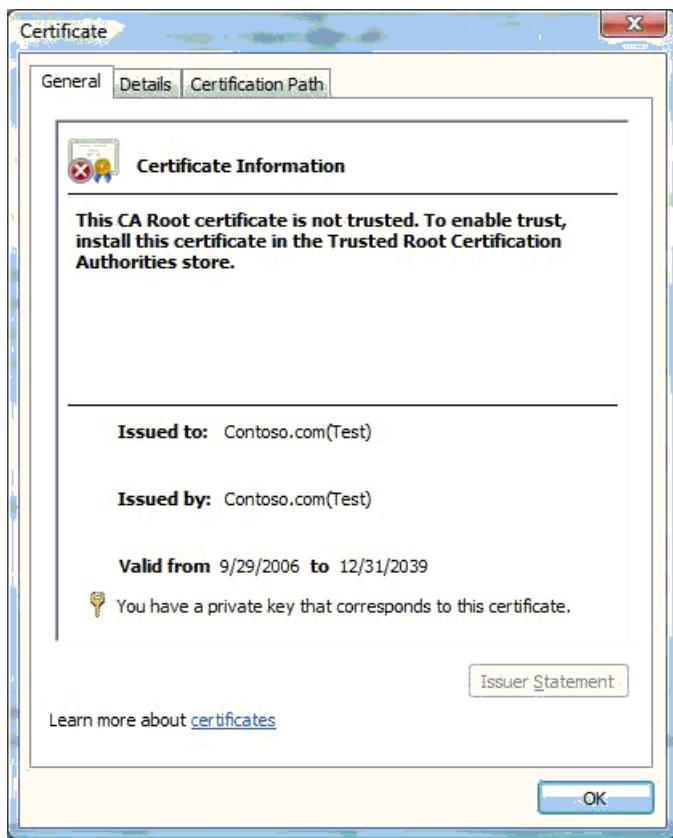
After the certificate is created and a copy is put in the certificate store, the Microsoft Management Console (MMC) Certificates snap-in can be used to view it. Do the following to view a certificate through the MMC **Certificates** snap-in:

1. Click **Start** and then click Start Search.
2. To start the Certificates snap-in, type Certmgr.msc and press the **Enter** key.
3. In the left pane of the Certificates snap-in, expand the PrivateCertStore certificate store folder and double-click Certificates.

The following screen shot shows the Certificates snap-in view of the **PrivateCertStore** certificate store folder.



To view the details about the Contoso.com(Test) certificate, double-click the certificate in the right pane. The following screen shot shows the details about the certificate.



Notice that the Certificate dialog box states: "This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store." This is the expected behavior. The certificate cannot be verified because Windows does not trust the issuing authority, "Contoso.com(Test)" by default.

# Creating a Catalog File for Test-Signing a Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

The catalog (.cat) file contains the digital signature for all the files which are part of the [driver package](#). For more information, see [Catalog Files](#).

There are two ways to create a [catalog file](#):

- If the driver package is installed through an INF file, use the [Inf2Cat](#) tool to create the catalog file. Inf2Cat automatically includes all the files in the driver package that are referenced within the package's INF file. For more information about how to use the Inf2Cat tool, see [Using Inf2Cat to Create a Catalog File](#).
- If the driver package is not installed through an INF file, use the [MakeCat](#) tool to create a catalog file by using a manually-created Catalog Definition File (.cdf).

For example, if the driver package is installed through an application, you may want to create a catalog file to digitally-sign all kernel-mode binary components of the package, such as the driver and any supporting .dll files. For more information about how to use the MakeCat tool, see [Using MakeCat to Create a Catalog File](#).

A catalog file is not needed to install the following types of drivers:

- A *boot-start driver*.
- A driver that is installed by using an application that does not use a [catalog file](#).

For these types of drivers, you have to embed a digital signature within the driver. For more information about this procedure, see [Test-Signing a Driver through an Embedded Signature](#).

For more information about how to create catalog files, see [Creating a Catalog File for a Test-Signed Driver Package](#).

# Using Inf2Cat to Create a Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Inf2Cat tool can be used to create catalog files for any [driver package](#) that has an INF file. For more information about Inf2Cat and its command-line arguments, see [Inf2Cat](#).

This topic discusses how to create a [catalog file](#) from a driver package's INF file. In this example, the INF file of the *ToastPkg* sample driver package is used. Within the WDK installation directory, this INF file is named *toastpkg.inf* and is located in the *src\general\toaster\toastpkg\inf* directory.

The name of catalog file that [Inf2Cat](#) produces is specified through the CatalogFile directive. One or more of these directives are declared within the [INF Version section](#) of the INF file. The INF Version section of the *toastpkg.inf* file is shown below:

```
[Version]
Signature="$WINDOWS NT$"
Class=TOASTER
ClassGuid={B85B7C50-6A01-11d2-B841-00C04FAD5171}
Provider=%ToastRUs%
DriverVer=09/21/2006,6.0.5736.1
CatalogFile.NTx86 = tostx86.cat
CatalogFile.NTIA64 = tostia64.cat
CatalogFile.NTAMD64 = tstampd64.cat
```

Two things should be noted about this [INF Version section](#):

1. The [INF Version section](#) declares three different catalog files, one for each Windows version which the driver package supports. When [Inf2Cat](#) is executed, it creates a catalog file for each Windows version that is specified through the /os option.

For example, Inf2Cat creates the catalog file *tstampd64.cat* if the command-line argument /os:Vista\_X64 is used. Similarly, the tool creates the catalog file *tostx86.cat* if the /os:Vista\_X86 option is used.

2. The [DriverVer directive](#) of the INF Version section declares an old time stamp and version.

Before you use [Inf2Cat](#), you must make sure that the INF file's DriverVer directive has a current time stamp and version value. This is needed for the [driver package](#) to install and replace a previously installed version of the package on the test computer.

You can use the [Stampinf](#) tool to update the time stamp and version value in the [DriverVer](#) directive. For example, to update the [DriverVer](#) directive in the *toastpkg.inf*, run the following command:

```
stampinf -f toastpkg.inf -d 09/01/2008 -v 9.0.9999.0
```

The following command line shows how to create a catalog file through the Inf2Cat tool by using the *Toastpkg.inf* file:

```
Inf2cat.exe /driver:src\general\toaster\toastpkg\toastcd\ /os:Vista_x64
```

Where:

- The /driver option specifies the directory which contains one or more INF files. Within this directory,

catalog files are created for those INF files that contain one or more CatalogFile directives. For more information about the CatalogFile directive, see [INF Version sections](#).

In this example, only the *toastpkg.inf* INF file is located within the specified *src\general\toaster\toastpkg\toastcd* directory.

- The */os:Vista\_x64* option specifies the catalog file is for the 64-bit version of Windows Vista. The Inf2Cat tool will match the name of the catalog file to the requested Windows version. Since the *toastpkg.inf* INF file contains a CatalogFile directive which has the NTAMD64 platform extension, Inf2Cat will create a catalog file that is named *tstampd64.cat*.

One or more Windows versions may be specified in the */os:* option. For example, if */os:Vista\_x64, Vistax32* is specified, Inf2Cat will create the *tstampd64.cat* and *tstx86.cat* files because of the INF CatalogFile directives in the *toastpkg.inf* INF file.

For more information about the tool's command-line arguments, see [Inf2Cat](#).

# Using MakeCat to Create a Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the [MakeCat](#) tool to create a [catalog file](#) for a [driver package](#).

You must use the MakeCat tool only to create catalog files for driver packages that are not installed by using an INF file. If the driver package is installed by using an INF file, use the [Inf2Cat](#) tool to create the catalog file. Inf2Cat automatically includes all the files in the driver package that are referenced within the package's INF file. For more information about how to use the Inf2Cat tool, see [Using Inf2Cat to Create a Catalog File](#).

**Note** Instead of creating and signing a catalog file, you can also embed a signature in the kernel-mode binaries of your [driver package](#), such as the driver and any .dll files your package may provide. For more information about this procedure, see [Test-Signing a Driver through an Embedded Signature](#).

To create a catalog file, you must first manually create a Catalog Definition File (.cdf) that describes the catalog header attributes and file entries. Once this file is created, you can then run the [MakeCat](#) tool to create a catalog file. The MakeCat tool does the following when it processes the .cdffile:

- Verifies the list of attributes for each file that is listed in the .cdffile.
- Adds the listed attributes to the [catalog file](#).
- Generates a cryptographic hash, or *thumbprint*, of each of the listed files.
- Stores the thumbprint of each file in the catalog file.

This topic describes how to create a .cdffile for the 64-bit kernel-mode binary files of the *ToastPkg* sample driver package. Within the WDK installation directory, these binary files are located in the *src\general\toaster\toastpkg\toastcd\amd64* directory.

To create a .cdffile for the *ToastPkg* sample [driver package](#), do the following:

1. Start Notepad and copy the text from the following sample. It contains the list of files to be cataloged, along with their attributes.

```
[CatalogHeader]
Name=tstamd64.cat
PublicVersion=0x00000001
EncodingType=0x00010001
CATATTR1=0x10010001:OSAttr:2:6.0
[CatalogFiles]
<hash>File1=amd64\toaster.pdb
<hash>File2=amd64\toaster.sys
<hash>File3=amd64\toastva.exe
<hash>File4=amd64\toastva.pdb
<hash>File5=amd64\tostrcls.dll
<hash>File6=amd64\tostrcls.pdb
<hash>File7=amd64\tostrco2.dll
<hash>File8=amd64\tostrco2.pdb
```

2. Save the file as *tstamd64.cdf* in the same folder as the driver package. **Note** When building a driver for multiple platforms, create a separate catalog file for each platform.

The following command line shows how to create a catalog file through the [MakeCat](#) tool by using the *tstamd64.cdffile*:

```
makecat -v tstampd64.cdf
```

After you run the tool, a file that is named *tstampd64.cat* is created.

For more information about the MakeCat tool and its command-line arguments, see the [Using MakeCat](#) website.

For more information about how to use the MakeCat tool, see [Creating a Catalog File for a Non-PnP Driver Package](#).

# Test-Signing a Driver Package's Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

After the [catalog file](#) for a [driver package](#) is created or updated, the catalog file can be signed through [SignTool](#).

Once signed, the digital signature stored within the catalog file is invalidated if any components of the driver package are modified.

When digitally signing a catalog file, SignTool saves the digital signature within the catalog file. The components of the driver package are not changed by SignTool. However, since the catalog file contains hashed values of the components of the driver package, the digital signature within the catalog file is maintained as long as the components hash to the same value.

SignTool can also add a time stamp to the digital signature. The time stamp allows you to determine when a signature was created and supports more flexible certificate revocation options, if necessary.

The following command line shows how to run SignTool to do the following:

- Test-sign the *tstamd64.cat* catalog file of the *ToastPkg* sample [driver package](#). For more information about how this [catalog file](#) was created, see [Creating a Catalog File for Test-Signing a Driver Package](#).
- Use the Contoso.com(Test) certificate from the PrivateCertStore for the test signature. For more information about how this certificate was created, see [Creating Test Certificates](#).
- Timestamps the digital signature through a time stamp authority (TSA).

To test-sign the *tstamd64.cat* catalog file, run the following command line:

```
SignTool sign /v /fd sha256 /s PrivateCertStore /n Contoso.com(Test) /t http://timestamp.digicert.com  
tstamd64.cat
```

Where:

- The **sign** command configures SignTool to sign the specified catalog file, *tstamd64.cat*.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/fd** option specifies the file digest algorithm to use for creating file signatures. The default is SHA1.
- The **/s** option specifies the name of the certificate store (*PrivateCertStore*) that contains the test certificate.
- The **/n** option specifies the name of the certificate (*Contoso.com(Test)*) that is installed in the specified certificate store.
- The **/t** option specifies URL of the TSA (<http://timestamp.digicert.com>) which will time stamp the digital signature.

## IMPORTANT

Including a time stamp provides the necessary information for key revocation in case the signer's code signing private key is compromised.

- *tstamd64.cat* specifies the name of the catalog file, which will be digitally-signed.

For more information about SignTool and its command-line arguments, see [SignTool](#).

For more information about test-signing a driver package's catalog file, see [Test-Signing a Catalog File](#).

# Test-Signing a Driver through an Embedded Signature

12/1/2020 • 2 minutes to read • [Edit Online](#)

A signed [catalog file](#) is all that you must have to correctly install and load most [driver packages](#). However, embedded-signing might also be an option. Embedded-signing refers to adding a digital signature to the driver's binary image file itself, instead of saving the digital signature in a catalog file. As a result, the driver's binary image is modified when the driver is embedded-signed.

Embedded-signing of kernel-mode binaries (for example, drivers and associated .dll files) are required whenever:

- The driver is a boot-start driver. In 64-bit versions of Windows Vista and later versions of Windows, the kernel-mode code signing requirements state that a *boot-start driver* must have an embedded signature. This is required regardless of whether the driver's driver package has a digitally-signed catalog file.
- The driver is installed through a driver package that does not include a catalog file.

As with [catalog files](#), [SignTool](#) is used to embed a digital signature within kernel-mode binary files by using a test certificate. The following command line shows how to run SignTool to do the following:

- Test-sign the 64-bit version of the Toastpkg sample's binary file, toaster.sys. Within the WDK installation directory, this file is located in the *src\general\toaster\toastpkg\toastcd\amd64* directory.
- Use the Contoso.com(Test) certificate from the PrivateCertStore for the test signature. For more information about how this certificate was created, see [Creating Test Certificates](#).
- Time stamp the digital signature through a time stamp authority (TSA).

To test-sign the *toaster.sys* file, run the following command line:

```
Signtool sign /v /fd sha256 /s PrivateCertStore /n Contoso.com(Test) /t http://timestamp.digicert.com  
amd64\toaster.sys
```

Where:

- The **sign** command configures SignTool to sign the specified catalog file, tstamd64.cat.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/fd** option specifies the file digest algorithm to use for creating file signatures. The default is SHA1.
- The **/s** option specifies the name of the certificate store (*PrivateCertStore*) that contains the test certificate.
- The **/n** option specifies the name of the certificate (*Contoso.com(Test)*) that is installed in the specified certificate store.
- The **/t** option specifies URL of the TSA (<http://timestamp.digicert.com>) which will time stamp the digital signature.

**IMPORTANT**

Including a time stamp provides the necessary information for key revocation in case the signer's code signing private key is compromised.

- *amd64\toaster.sys* specifies the name of the kernel-mode binary file which will be embedded-signed.

For more information about SignTool and its command-line arguments, see [SignTool](#).

For more information about how to test-sign a driver by using an embedded signature, see [Test-Signing a Driver File](#).

# Configuring the Test Computer to Support Test-Signing

12/5/2018 • 2 minutes to read • [Edit Online](#)

Before you install a test-signed driver package on a test computer, you must configure the computer to support test-signing. This section describes the procedures involved with enabling test-signing support on a computer, and includes the following topics:

[The TESTSIGNING Boot Configuration Option](#)

[Enabling Code Integrity Event Logging and System Auditing](#)

Once the test computer is reconfigured to enable test-signing, the computer must be restarted for the changes to take effect.

# Enable Loading of Test Signed Drivers

12/1/2020 • 2 minutes to read • [Edit Online](#)

By default, Windows does not load test-signed kernel-mode drivers. To change this behavior and enable test-signed drivers to load, use the boot configuration data editor, BCDEdit.exe, to enable or disable TESTSIGNING, a boot configuration option. You must have Administrator rights to enable this option.

## NOTE

For 64-bit versions of Windows Vista and later versions of Windows, the kernel-mode code signing policy requires that all kernel-mode code have a digital signature. However, in most cases, an unsigned driver can be installed and loaded on 32-bit versions of Windows Vista and later versions of Windows. For more information, see [Driver Signing Policy](#).

## Administrator rights required

To use BCDEdit, you must be a member of the Administrators group on the system and run the command from an elevated command prompt. To open an elevated Command Prompt window, type **cmd** into the search box in the Windows taskbar, select and hold (or right-click) **Command Prompt** in the search results, and then select **Run as administrator**.

## WARNING

Administrative rights are required to use BCDEdit to modify boot configuration data. Changing some boot entry options by using **BCDEdit /set** could render your computer inoperable. As an alternative, use System Configuration utility (MSConfig.exe) to change boot settings.

## Enable or disable use of test-signed code

Run BCDEdit command lines to enable or disable the loading of test-signed code. For a change to take effect, whether enabling or disabling the option, you must restart the computer after changing the configuration.

## NOTE

Before setting BCDEdit options you might need to disable or suspend BitLocker and Secure Boot on the computer.

To enable test-signed code, use the following BCDEdit command line:

```
Bcdedit.exe -set TESTSIGNING ON
```

To disable use of test-signed code, use the following BCDEdit command line:

```
Bcdedit.exe -set TESTSIGNING OFF
```

The following figure shows the result of using the BCDEdit command line to enable test-signing.

```
cmd Select D:\Windows\System32\cmd.exe
D:\Windows\system32>bcdedit -set TESTSIGNING ON
The operation completed successfully.

D:\Windows\system32>bcdedit

Windows Boot Manager
identifier          <bootmgr>
device              C:
description         Windows Boot Manager
locale              en-US
inherit              {globalsettings}
default             <current>
resumeobject        <f4b9eab3-d8c2-11da-8bb4-d1019e73cc8d>
displayorder        <7af09567-c765-11da-aefe-0014224ab5ca>
toolsdisplayorder  <mendiag>
timeout             30

Windows Boot Loader
identifier          <current>
device              D:
path                \Windows\system32\winload.exe
description         Microsoft Windows
locale              en-US
inherit              {bootloadersettings}
osdevice            D:
systemroot          \Windows
resumeobject        <f4b9eab3-d8c2-11da-8bb4-d1019e73cc8d>
nx                 OptIn
testsingning        Yes
```

## Behavior of Windows when loading test-signed code is enabled

When loading test-signed code is enabled, Windows does the following:

- Displays a watermark with the text "Test Mode" in all four corners of the desktop, to remind users the system has test-signing enabled. **Note** Starting with Windows 7, Windows displays this watermark only in the lower right-hand corner of the desktop.
- Displays a watermark with the text "Test Mode" in the lower-left corner of the desktop to remind users that the system has test-signing enabled.
- The operating system loader and the kernel load drivers that are signed by any certificate. The certificate validation is not required to chain up to a trusted root certification authority. However, each driver image file must have a digital signature.

# Code Integrity Event Logging and System Auditing

12/5/2018 • 2 minutes to read • [Edit Online](#)

Code Integrity is the kernel-mode component that implements driver signature verification. It generates system events that are related to image verification and logs the information in the Code Integrity log:

- The Code Integrity operational log view shows only image verification error events.
- The Code Integrity verbose log view shows the events for successful signature verifications.

For more information about code integrity event logging and system auditing, see [Code Integrity Diagnostic System Log Events](#).

For more information about how to enable the system audit log and verbose logging, see [Enabling the System Event Audit Log](#).

# Verifying the Test Signature

11/2/2020 • 2 minutes to read • [Edit Online](#)

After the test certificate is copied to the Trusted Root Certification Authorities certificate store on the test computer, **SignTool** can be used to do the following:

- Verify the signature of a specified file in a [driver package's catalog files](#).
- Verify the embedded signature of a kernel-mode binary file, such as a *boot-start driver*.

The following example verifies the signature for one of the files, *toastpkg.inf*, in the Toastpkg sample's signed catalog file, *tstamd64.cat*. For more information about how this catalog file was created, see [Using Inf2Cat to Create a Catalog File](#):

```
Signtool verify /pa /v /c tstamd64.cat toastpkg.inf
```

Where:

- The **verify** command configures SignTool to verify the specified file, *toastpkg.inf*.
- The **/pa** option specifies the use of the Authenticode verification policy when verifying the digital signature.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/c** option specifies the catalog file name.

**Note** When verifying the signature of an embedded-signed kernel-mode binary file, do not use the **/c** argument.

- *toastpkg.inf* specifies the name of the file to be verified.

The following example verifies the signature of the *Toastpkg* sample's signed catalog file, *Tstamd64.cat*.

```
Signtool verify /pa /v tstamd64.cat
```

For more information about how to use **SignTool** to verify a digital signature of a catalog file, see [Verifying the Signature of a Test-Signed Catalog File](#).

# Installing a Test-Signed Driver Package on the Test Computer

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can install the test-signed [driver package](#) on the test computer once:

- The test computer is prepared to install test-signed drivers or driver packages. For more information, see [Configuring the Test Computer to Support Test-Signing](#).
- The test certificate is copied to the Trusted Root Certification Authorities certificate store on the test computer. For more information, see [Installing Test Certificates](#).

You can install the test-signed [driver package](#) on the computer through:

- The [DevCon](#) tool, which is a WDK command line tool for installing drivers.

This topic will first review the process of test-signing a driver package, and then describe how you can install the driver package on the test computer. This topic uses the *ToastPkg* sample driver package. Within the WDK installation directory, the package's source files are located in the *src\general\toaster\toastpkg* directory.

Follow these steps to build and test-sign the *ToastPkg* sample driver package:

1. On the signing computer, build the *ToastPkg* sample driver package's kernel-mode binaries. For more information about how to build drivers, see [Building a Driver](#).
2. On the signing computer, create the *Contoso.com(Test)* certificate as described in [Creating Test Certificates](#).
3. On the signing computer, create a [catalog file](#) for the *ToastPkg* sample driver package as described in [Creating a Catalog File for Test-Signing a Driver Package](#).
4. On the signing computer, test-sign the *ToastPkg* sample driver package's catalog file as described in [Test-Signing a Driver Package's Catalog File](#). When signing the driver package, use the *Contoso.com (Test)* certificate from the *PrivateCertStore* for the test signature.
5. Prepare the test computer to support test-signing, as described in [Configuring the Test Computer to Support Test-Signing](#).
6. Copy the *Contoso.com(Test)* certificate to the Trusted Root Certification Authorities certificate store on the test computer, as described in [Installing Test Certificates](#).

The following topics describe how the *ToastPkg* sample driver package can be installed on the test computer:

- [Using the DevCon Tool to Install a Driver Package](#)

Once the [driver package](#) is installed, you can troubleshoot problems with the loading of test-signed drivers through the methods described in [Troubleshooting Install and Load Problems with Signed Driver Packages](#).

For more information about how to install a test-signed driver package, see [Installing Test-Signed Driver Packages](#).

# Using the DevCon Tool to Install a Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

The example in this topic uses the *ToastPkg* sample [driver package](#). Within the WDK installation directory, the package's source files are located in the *src\general\toaster\toastpkg\toastcd* directory. After you have built and digitally-signed this driver package, copy the driver package to the directory *c:\toaster* on the test computer.

To install the driver package through DevCon, do the following:

1. To use the DevCon tool, the user must be a member of the Administrators group on the test computer and run DevCon from an elevated command prompt. To open an elevated Command Prompt window, create a desktop shortcut to *Cmd.exe*, select and hold (or right-click) the *Cmd.exe* shortcut, and select **Run as administrator**.
2. From the elevated, Command Prompt window, enter the following:

```
devcon.exe install c:\toaster\toastpkg.inf {b85b7c50-6a01-11d2-b841-00c04fad5171}\mstoaster
```

This command-line specifies the location of the driver package's INF file (*c:\toaster\toastpkg.inf*) and the toaster device's hardware identifier (ID), which is specified within the INF file.

# WHQL Test Signature Program

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Windows Hardware Quality Labs (WHQL) Test Signature program supports test-signing of drivers that will subsequently be submitted for a [WHQL release signature](#). Independent Hardware Vendors (IHVs) that participate in this program can submit [driver packages](#) to be test-signed. The signature on a test-signed [catalog file](#) is generated by a test root certificate that is issued by WHQL under the Microsoft Test Root Authority. Together with the test-signed catalog file, WHQL also provides participants with the test root certificate.

Before users can install a driver from a WHQL test-signed driver package, the test computer must be configured by following these steps:

1. The User Account Control (UAC) must be disabled to install the WHQL test certificate correctly. For more information, see [Disabling UAC](#).
2. The WHQL test root certificate (*Testroot.cer*) must be installed in the test computer in the Local Computer folder of the Trusted Root Certification Authorities certificate store. For more information, see [Installing the WHQL test root certificate](#).
3. The [TESTSIGNING Boot Configuration Option](#) must be set on the test computer. For more information, see [Setting the TESTSIGNING Boot Configuration Option](#).

For information about how to obtain a WHQL test signature, email [winqual@microsoft.com](mailto:winqual@microsoft.com) and include "Test Signature" in the Subject line.

## Disabling UAC

**Note** To disable UAC on the test computer, you must be able to log on with or provide the credentials of a member of the local **Administrators** group.

You can disable UAC on Windows 7 by following these steps:

1. On the **Start** menu, enter "UAC" and then select **Change User Account settings**.
2. Move the slide bar to the bottom (**Never Notify**) and then select **OK**.

You can disable UAC on Windows Vista and Windows Server 2008 by following these steps:

1. Start Control Panel and double-click **User Accounts**.
2. In the User Accounts tasks window, select **Turn User Account Control on or off**.
3. Clear the **Use User Account Control (UAC) to help protect your computer** check box, and then select **OK**.

As soon as UAC is disabled, you must restart the test computer to apply the change.

## Installing the WHQL test root certificate

The WHQL test root certificate is installed on the test computer by following these steps:

1. Double-click the test root certificate (*Testroot.cer*) and then select **Install Certificate**.
2. Select **Place all certificates in the following store**, and then select **Browse**.
3. Select the **Show physical stores** check box, and then expand **Trusted Root Certification Authorities**.
4. Select the Local Computer folder, and then select **OK**.

**Note** If UAC was not previously disabled on the test computer, the Local Computer folder is not displayed.

5. Complete the Certificate Import Wizard and accept all the default settings to install the certificate.

### **Setting the TESTSIGNING Boot Configuration Option**

The Microsoft Test Root Authority is accepted when test-signing is enabled by setting the [TESTSIGNING Boot Configuration Option](#) on the computer in which the test-signed driver package is to be installed. This option is enabled by following these steps:

1. Open an elevated Command Prompt window. To open an elevated Command Prompt window, create a desktop shortcut to Cmd.exe, select and hold (or right-click) the Cmd.exe shortcut, and select **Run as administrator**.
2. In the elevated Command Prompt window, run the following command:

**BCDEdit /SET TESTSIGNING ON**

3. Restart the test computer to apply the change.

# MakeCert Test Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

A MakeCert test certificate is an [X.509 digital certificate](#) that is created by using the [MakeCert](#) tool. A MakeCert test certificate is a self-signed root certificate that can be used to test-sign a [driver package's catalog file](#) or to test-sign a driver file by embedding a signature in the driver file.

To learn more about creating a MakeCert test certificate, see [Creating Test Certificates](#).

## Installing a MakeCert test certificate

To test-sign a [catalog file](#) or embed a signature in a driver file, the MakeCert test certificate can be in the Personal certificate store ("my" store), or some other custom certificate store, of the local computer that signs the software. However, to verify a test signature, the corresponding test certificate must be installed in the [Trusted Root Certification Authorities certificate store](#) of the local computer that you use to verify the signature.

Use the [CertMgr](#) tool, as follows, to install a test certificate in the Trusted Root Certification Authorities certificate store of the local computer that you use to sign drivers:

```
CertMgr /add CertFileName.cer /s /r localMachine root
```

Before you can install a [driver package](#) that is signed by a MakeCert test certificate, the test certificate must be installed in the Trusted Root Certification Authorities certificate store and the [Trusted Publishers certificate store](#) of the test computer. For information about how to install a MakeCert test certificate on a test computer, see [Installing a Test Certificate on a Test Computer](#).

# Commercial Test Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

## Caution

Starting in 2021, the majority of cross-certificates will begin to expire. Once these cross-certificates expire, code signing certificates that chain to these cross-certificates will no longer be able to create new kernel mode digital signatures. This will affect all versions of Windows. For more information, see [Deprecation of software publisher certificates and commercial release certificates](#).

A *commercial test certificate* refers to a digital certificate that a publisher obtains from a trusted, third-party, commercial certification authority (CA) that is a member of the Microsoft Root Certificate Program. GTE and VeriSign, Inc. are two examples of such a CA.

A CA that is a member of the Microsoft Root Certificate Program validates the identity and entitlement of an applicant and then issues a certificate that the applicant uses to sign its drivers. The signing process stamps the driver with the publisher's identity and can be used to verify that the driver has not been modified since it was signed.

A commercial test certificate is the same type of certificate as a [commercial release certificate](#). However, for security reasons, you should not use a digital certificate that is used to test-sign drivers to also sign a release driver. You should obtain separate digital certificates to use for release-signing and for test-signing. For information about how to manage digital certificates, see [Managing the Digital Signature or Code Signing Keys](#).

Follow the instructions that are provided by the CA on how to obtain and install the certificate on a computer where you will sign a driver.

For more information about how to obtain a digital certificate from a CA, see the [Microsoft Root Certificate Program Members](#) website.

# Enterprise CA Test Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

An *Enterprise CA test certificate* is an Authenticode digital certificate that is deployed by an Enterprise certification authority (Enterprise CA) across an enterprise. As part of a public key infrastructure, a domain administrator can create an Enterprise CA to manage the enterprise-wide Authenticode certification of [driver packages](#) that are under development.

An Enterprise CA is integrated with Active Directory and publishes certificates and certificate revocation lists to Active Directory. The Enterprise CA uses information that is stored in Active Directory, including user accounts and security groups, to approve or deny certificate requests.

An Enterprise CA uses certificate templates. When a certificate is issued, the Enterprise CA uses information in the certificate template to generate a certificate with the appropriate attributes for that certificate type.

If you want to enable automated certificate approval and automatic user certificate enrollment, the Enterprise CA infrastructure must be integrated with Active Directory.

In summary, a domain administrator has to do the following to create an Enterprise CA to manage the enterprise-wide Authenticode certification of [driver packages](#) that are under development:

- Install an Enterprise CA.
- Create a test (code-signing) certificate template.
- Publish the test certificate template in Active Directory.
- Configure Group Policy to distribute the test certificates that are issued by the Enterprise CA.

Detailed information on how to configure an Enterprise CA is beyond the scope of this documentation. For complete information about how to design a public key infrastructure and installing Enterprise CA, see the [Code-Signing Best Practices](#) website, the Windows Server 2003 Deployment Kit, the Windows Server 2003 Help and Support Center, and the [Public Key Infrastructures](#) webpage of the [Microsoft TechNet](#) website. The TechNet website includes information about certificates, certificate services, and certificate templates.

Information about configuring an Enterprise CA to test-sign [driver packages](#) is also provided in the readme file *Selfsign\_readme.htm*, which located in the *src\general\build\driversigning* directory of the WDK.

# Test-Signing Driver Packages

12/1/2020 • 4 minutes to read • [Edit Online](#)

In this section, a computer that test-signs drivers for release on Windows Vista and later versions of Windows is referred to as the *signing computer*. The signing computer must be running Windows XP SP2 or later versions of Windows. For example, a driver intended for release on Windows 7 can be signed on a computer running Windows Vista.

In order to use the [driver signing tools](#), the signing computer must have the Windows Vista and later versions of the WDK installed.

**Note** You must use the version of the [SignTool](#) tool that is provided in the Windows Vista and later versions of the Windows Driver Kit (WDK). Earlier versions of the SignTool do not support the kernel-mode code signing policy for Windows Vista and later versions of Windows.

To comply with the [kernel-mode code signing policy](#) and the [Plug and Play \(PnP\) device installation signing requirements](#) of Windows Vista and later versions of Windows, you must sign a driver during the development and test of that driver. You can sign the driver on the signing computer as follows, based on the driver type.

**Note** The Windows code-signing policy requires that a signed [catalog file](#) for a [driver package](#) be installed in the system component and driver database. PnP device installation automatically installs the catalog file of a PnP driver in the driver database. However, if you use a signed catalog file to sign a non-PnP driver, the installation application that installs the driver must also install the catalog file in the driver database.

## PnP Kernel-Mode Boot-Start Driver

To comply with the kernel-mode code signing policy of 64-bit versions of Windows Vista and later versions of Windows, embed a signature in the *boot-start driver* file as follows:

1. [Test-sign the driver file](#).
2. [Verify the signature of the test-signed driver file](#).

Starting with Windows Vista, embedding a signature in a *boot-start driver* file is optional for 32-bit versions of Windows. Although Windows will check if a kernel-mode driver file has an embedded signature, an embedded signature is not required.

To comply with the [PnP device installation signing requirements](#) of Windows Vista and later versions of Windows, you must also test-sign a [catalog file](#) for the [driver package](#). If a driver file will also include an embedded signature, embed the signature in the driver file before signing the driver package's catalog file.

You can submit a request to have the [Windows Hardware Quality Labs \(WHQL\)](#) test-sign the [catalog file](#).

Alternatively, you can test-sign a catalog file yourself with a test certificate, as follows:

1. [Create a catalog file](#).
2. [Test-sign the catalog file](#).
3. [Verify the signature of the test-signed catalog file](#).

You can verify the signature of the catalog file itself or the signature of individual files that have corresponding entries in the catalog file.

## Non-PnP Kernel-Mode Boot-Start Driver

To comply with the kernel-mode code signing policy of 64-bit versions of Windows Vista and later versions of Windows, embed a signature in a *boot-start driver* file as follows:

1. [Test-sign the driver file.](#)
2. [Verify the signature of the test-signed driver file.](#)

Starting with Windows Vista, embedding a signature in a *boot-start driver* file is optional for 32-bit versions of Windows. Although Windows will check if a kernel-mode driver file has an embedded signature, an embedded signature is not required.

The [PnP device installation signing requirements](#) do not apply to non-PnP drivers.

#### **PnP Kernel-Mode Driver that is not a Boot-Start Driver**

The kernel-mode code signing policy on 64-bit versions of Windows Vista and later versions of Windows does not require a non-boot PnP driver to have an embedded signature. However, if the driver file will include an embedded signature, embed the signature in the driver file before signing the [driver package's catalog file](#).

For a PnP kernel-mode driver that is not a *boot-start driver*, signing the catalog file for the driver package complies with the kernel-mode code signing policy on 64-bit versions of Windows Vista and later versions of Windows, as well as the PnP device installation signing requirements for all versions of Windows Vista and later.

You can submit a request to have the Windows Hardware Quality Labs (WHQL) test-sign the catalog file.

Alternatively, you can test-sign a catalog file yourself with a test certificate in the same manner as described in this section for test-signing the catalog file of a PnP kernel-mode *boot-start driver*.

#### **Non-PnP Kernel-Mode Driver that is not a Boot-Start Driver**

To comply with the kernel-mode code signing policy of 64-bit versions of Windows Vista and later versions of Windows, embed a signature in the driver file or sign the [driver package's catalog file](#).

Starting with Windows Vista, embedding a signature in a driver file is optional for 32-bit versions of Windows. Although Windows will check if a kernel-mode driver file has an embedded signature, an embedded signature is not required.

The PnP device installation signing requirements do not apply to non-PnP drivers.

**Note** Using embedded signatures is generally simpler and more efficient than using a signed catalog file. For more information about the advantages and disadvantages of using embedded signatures versus signed catalog files, see [Test Signing a Driver](#).

#### **To embed a test signature in a file for a non-PnP kernel-mode driver that is not a boot-start driver**

1. [Test-sign the driver file.](#)
2. [Verify the signature of the test-signed driver file.](#)

#### **To test-sign a catalog file for a non-PnP kernel-mode driver that is not a boot-start driver**

1. [Create a catalog file for the non-PnP driver.](#)
2. [Test-sign the catalog file.](#)
3. [Verify the signature of the test-signed catalog file.](#)

# Test-Signing a Driver File

11/2/2020 • 2 minutes to read • [Edit Online](#)

Use [SignTool](#) to embed a signature in a driver file.

## Using a MakeCert Test Certificate or a Commercial Test Certificate to Embed a Test Signature in a Driver File

Use the following SignTool command to embed a signature in a driver file by using a [MakeCert test certificate](#) or a [commercial test certificate](#).

```
SignTool sign /v /s TestCertStoreName /n TestCertName /t http://timestamp.digicert.com DriverFileName.sys
```

Where:

- The `sign` command configures SignTool to embed a signature in the driver file `DriverFileName.sys`.
- The `/v` verbose option configures SignTool to print execution and warning messages.
- The `/s TestCertStoreName` option supplies the name of the test certificate store that contains the test certificate named `TestCertName`.
- The `/n TestCertName` option supplies the name of the test certificate that is installed in the certificate store named `TestCertStoreName`. The test certificate can be either a MakeCert test certificate or a commercial test certificate.
- The `/t http://timestamp.digicert.com` option supplies the URL to the publicly-available time-stamp server that DigiCert provides.
- `DriverFileName.sys` is the name of the driver file.

The following command shows how to use SignTool to test-sign a driver file. This example embeds a signature in `Toaster.sys`, which is in the `amd64` subdirectory under the directory in which the command is run. The test certificate is named "contoso.com(test)" and it is installed in the certificate store named "PrivateCertStore."

```
SignTool sign /v /s PrivateCertStore /n contoso.com(test) /t http://timestamp.digicert.com amd64\toaster.sys
```

## Using an Enterprise CA Test Certificate to Embed a Test Signature in a Driver File

The following SignTool command assumes that an Enterprise CA issues the test certificate that you use to test-sign a [driver package](#). If the [Enterprise CA test certificate](#) is the only test certificate that is present in your certificate stores, you can use the following command where you specify only the `/a` option and the name of the driver file. In this situation, SignTool will locate and use your Enterprise CA test certificate by default.

If you have created or obtained other test certificates in addition to an Enterprise CA test certificate, you must use the SignTool options `/s` and `/n` to specify the name of the test certificate store and the name of the test certificate that is installed in the test certificate store.

```
SignTool sign /v /a /t http://timestamp.digicert.com DriverFileName.sys
```

# Verifying the Signature of a Test-Signed Driver File

11/2/2020 • 2 minutes to read • [Edit Online](#)

To verify a test signature that is embedded in a driver file, use the following **SignTool** command:

```
SignTool verify /v /pa DriverFileName.sys
```

Where:

- The **verify** command configures SignTool to verify the signature that is embedded in the driver file *DriverFileName.sys*.
- The **/v** option configures SignTool to print the execution and warning messages.
- The **/pa** option configures SignTool to verify that the signature that is embedded in *DriverFileName.sys* complies with the PnP device installation signing requirements.
- *DriverFileName.sys* is the name of the driver file.

Be aware that the SignTool **verify** command does not explicitly specify the test certificate that was used to sign the driver file. For the verify operation to succeed, you must first install the test certificate in the [Trusted Root Certification Authorities certificate store](#) of the local computer that you use to verify the signature. For more information about how to install the test certificate in the Trusted Root Certification Authorities certificate store of a local computer, see [Installing a Test Certificate on a Test Computer](#). The installation procedure is the same on both the signing computer and a test computer.

For example, the following command verifies the embedded signature in *Toaster.sys*, which is in the *amd64* subdirectory under the directory in which the command is run.

```
SignTool verify /v /pa amd64\toaster.sys
```

# Creating a Catalog File for a Test-Signed Driver Package

12/5/2018 • 2 minutes to read • [Edit Online](#)

To create a [catalog file](#) for a test-signed PnP [driver package](#), follow the same procedure that is used to [create a catalog file for a PnP driver package](#).

To create a catalog file for a test-signed non-PnP driver package, follow the same procedure that is used to [create a catalog file for a non-PnP driver package](#).

# Test-Signing a Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

After you create and verify a [driver package's catalog file](#), use [SignTool](#) to test-sign the catalog file

## Using a MakeCert Test Certificate or Commercial Test Certificate to Test-Sign a Driver Package's Catalog File

Use the following SignTool command to sign a [catalog file](#) by using a [MakeCert test certificate](#) or a [commercial test certificate](#):

```
SignTool sign /v /s TestCertStoreName /n TestCertName /t http://timestamp.digicert.com CatalogFileName.cat
```

Where:

- The `sign` command configures SignTool to sign the [catalog file](#) that is named *CatalogFileName.cat*.
- The `/v` verbose option configures SignTool to print execution and warning messages.
- The `/s TestCertStoreName` option supplies the name of the test certificate store that contains the test certificate named *TestCertName*.
- The `/n TestCertName` option supplies the name of the test certificate that is installed in the certificate store named *TestCertStoreName*. The test certificate can be either a MakeCert test certificate or a commercial test certificate.
- The `/t http://timestamp.digicert.com` option supplies the URL to the publicly-available time-stamp server that DigiCert provides.
- *CatalogFileName.cat* is the name of the [catalog file](#).

The following command shows how to use SignTool to test-sign a [driver package's catalog file](#). This example signs the catalog file *Tstamd64.cat*, which is in the same directory in which the command is run. The test certificate is named "contoso.com(test)," which is installed in the certificate store named "PrivateCertStore."

```
SignTool sign /v /s PrivateCertStore /n contoso.com(test) /t http://timestamp.digicert.com tstamd64.cat
```

## Using an Enterprise CA Test Certificate to Test-Sign a Driver Package's Catalog File

The following SignTool command assumes that an Enterprise CA issues the test certificate that you use to test-sign a [driver package](#). If the [Enterprise CA test certificate](#) is the only test certificate that is present in your certificate stores, you can use the following command where you specify only the `/a` option and the name of the [catalog file](#). In this situation, SignTool will locate and use your Enterprise CA test certificate by default.

If you have created or obtained other test certificates in addition to an Enterprise CA test certificate, you must use the SignTool options `/s` and `/n` to specify the name of the test certificate store and the name of the test certificate that is installed in the test certificate store.

```
SignTool sign /v /a /t http://timestamp.digicert.com CatalogFileName.cat
```

# Verifying the Signature of a Test-Signed Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

To verify that a [driver package's catalog file](#) was signed by a valid [test certificate](#), use the following **SignTool** command:

```
SignTool verify /v /pa CatalogFileName.cat
```

To verify that a file, listed in a [driver package's catalog file](#), is signed by a test certificate, use the following **SignTool** command:

```
SignTool verify /v /pa /c CatalogFileName.cat DriverFileName
```

Where:

- The **verify** command configures SignTool to verify the signature of the driver package's catalog file *CatalogFileName.cat* or the driver file *DriverFileName*.
- The **/v** option configures SignTool to print execution and warning messages.
- The **/pa** option configures SignTool to verify that the signature of the catalog file or driver file complies with the PnP device installation signing requirements.
- *CatalogFileName.cat* is the name of the catalog file for a driver package.
- The **/c CatalogFileName.cat** option specifies a catalog file that includes an entry for the file *DriverFileName*.
- *DriverFileName* is the name of a file that has an entry in the catalog file *CatalogFileName.cat*.

Be aware that the SignTool **verify** command does not explicitly specify the test certificate that was used to sign the [catalog file](#). For the verify operation to succeed, you must first install the test certificate in the [Trusted Root Certification Authorities certificate store](#) of the local computer that you use to verify the signature. For more information about how to install the test certificate in the Trusted Root Certification Authorities certificate store of a local computer, see [Installing a Test Certificate on a Test Computer](#). The installation procedure is the same on both the signing computer and a test computer.

For example, the following command verifies that *Tstam64.cat* has a test signature that complies with the PnP device installation signing requirements of Windows Vista and later versions of Windows. In this example, *Tstam64.cat* is in the same directory in which the command is run.

```
SignTool verify /v /pa tstam64.cat
```

# Installing Test-Signed Driver Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, a test-signed [driver package](#) should install and load without user interaction if the following conditions are true:

- The driver package complies with the generic requirements of Windows Vista and later versions of Windows.
- The driver package is signed and the signature is verified, as described in [Test-Signing Driver Packages](#).
- The driver package is not altered after it is signed.
- The test certificates that were used to sign the driver package are installed correctly on the test computer, as described in [Installing a Test Certificate on a Test Computer](#).
- If a non-PnP driver has a signed [catalog file](#) instead of an embedded signature, the installation application that installs the driver has installed the catalog file in the system catalog root directory, as described in [Installing a Test-Signed Catalog File for a Non-PnP Driver](#).
- Test-signing is enabled on the test computer. For more information, see [the TESTSIGNING Boot Configuration Option](#).

For an overview of how to install a test-signed driver package, see [Installing a Test-Signed Driver Package on the Test Computer](#).

For more information about how to troubleshoot installation problems, see [Troubleshooting Install and Load Problems with Signed Driver Packages](#).

# Installing a Test Certificate on a Test Computer

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [test certificates](#) that are used to embed signatures in driver files and to sign a [driver package's catalog file](#) must be added to the [Trusted Root Certification Authorities certificate store](#) and the [Trusted Publishers certificate store](#). You can manually add a test certificate to these certificate stores by using the [CertMgr](#) tool, as described in [Using CertMgr to Install Test Certificates on a Test Computer](#).

You can also configure the default domain policy to automatically deploy a test certificate on computers in a domain, as described in [Deploying a Test Certificate by Using the Default Domain Policy](#).

# Using CertMgr to Install Test Certificates on a Test Computer

11/2/2020 • 2 minutes to read • [Edit Online](#)

To install test certificates on a test computer by using [CertMgr](#), follow these steps:

1. Copy the certificate (.cer) file, which was used to [test-sign](#) drivers, to the test computer. You can copy the certificate file to any directory on the test computer.
2. Use [CertMgr](#) commands to add the certificate to the [Trusted Root Certification Authorities certificate store](#) and the [Trusted Publishers certificate store](#).

The following CertMgr command adds the certificate in the certificate file *CertificateFileName.cer* to the Trusted Root Certification Authorities certificate store on the test computer:

```
CertMgr.exe /add CertificateFileName.cer /s /r localMachine root /all
```

The following CertMgr command adds the certificate in the certificate file *CertificateFileName.cer* to the Trusted Publishers certificate store on the test computer:

```
CertMgr.exe /add CertificateFileName.cer /s /r localMachine trustedpublisher
```

# Deploying a Test Certificate by Using the Default Domain Policy

12/5/2018 • 2 minutes to read • [Edit Online](#)

A domain administrator who is logged on to a domain on a computer that is running Windows Vista and later versions of Windows can configure the default domain policy to deploy a test certificate to the certificate stores of computers in the domain.

After the default domain policy is configured as described in this topic, the certificate stores of computers in the domain are updated approximately every 90 minutes and every time that a computer is restarted. In addition, each computer can force an update of the default domain policy by using the **gpupdate /force** command.

To configure the default domain policy to deploy a test certificate to the Trusted Root Certification Authorities certificate store, follow these steps:

1. Click **Start**, point to **Settings**, click **Control Panel**, and then double-click **Administrative Tools**.
2. Open Domain Security Policy.
3. In the left pane of the **Console** box, expand **Default Domain Policy**, expand **Computer Configuration**, expand **Windows Settings**, expand **Security Settings**, expand **Public Key Policies**, and select **Trusted Root Certification Authorities**.
4. On the main menu bar, click **Action**, and then click **Import** to open the Certificate Import Wizard.
5. On the first page of the Certificate Import Wizard, click **Next**, and then in the **File Name** box of the **File to Import** page, enter the name of the file that contains the certificate to be imported, and then click **Next**. (Or, click **Browse**, browse to the file, and select it).
6. To finish the Certificate Import Wizard, click **Next** twice and then click **Finish**.

Use the same type of procedure to configure the default domain policy to deploy a test certificate to the Trusted Publishers certificate store as described above for the Trusted Root Certification Authorities certificate store. In step (3) of the procedure in this section, select Trusted Publishers certificate store instead of the Trusted Root Certification Authorities certificate store.

# Installing a Test-Signed Catalog File for a Non-PnP Driver

11/2/2020 • 2 minutes to read • [Edit Online](#)

To comply with the [kernel-mode code signing policy](#) of 64-bit versions of Windows Vista and later versions of Windows, a non-boot, non-PnP driver must have either an embedded signature or a signed [catalog file](#) that is installed in the system component and driver database. PnP device installation automatically installs the catalog file of a PnP driver in the driver database. However, for non-PnP drivers, the installation application that installs a non-PnP driver must install the catalog file in the driver database.

A driver installation application can programmatically install a catalog file in the system component and driver database by using the [CryptCATAdminAddCatalog](#) cryptography function. If the application is to be redistributable, you should use this approach to install the catalog file. For more information about this approach, see [Installing a Catalog File by using CryptCATAdminAddCatalog](#).

Alternatively, you can use the [SignTool](#) tool to install a [catalog file](#) in the system component and driver database. However, SignTool is not a redistributable tool. Therefore, an installation application can use SignTool on a computer only if the tool is already installed on the computer in a manner that complies with the Microsoft Software License Terms for the tool. For more information about this approach, see [Installing a Catalog File by using SignTool](#).

**Tip** Using embedded signatures is generally easier and more efficient than by using a signed catalog file. For more information about the advantages and disadvantages of using embedded signatures versus signed catalog files, see [Test Signing a Driver](#).

# Installing a Catalog File by using CryptCATAdminAddCatalog

11/2/2020 • 2 minutes to read • [Edit Online](#)

An installation program can use the [CryptCATAdminAddCatalog](#) and other [CryptCATAdminXxx](#) cryptography functions to programmatically install a [catalog file](#) in the system component and driver database.

The installation program must use the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0 in the following way:

- The source files of the installation program must include the following header (.h) files:
  - *Mscat.h*, which defines the prototypes and structures for the various [CryptCATAdminXxx](#) functions.
  - *Softpub.h*, which defines the various data structures and GUIDs that are used by the [CryptCATAdminXxx](#) functions.
- The installation program must link to *Wintrust.lib*.

To use these [CryptCATAdminXxx](#) cryptography functions, an installation program does the following:

1. Calls [CryptCATAdminAcquireContext](#) to obtain a handle to a catalog administrator context. The application identifies the subsystem by setting the *pgSubsystem* input parameter to a pointer to the GUID `DRIVER_ACTION_VERIFY`. This GUID is defined in *Softpub.h*.
2. Calls [CryptCATAdminAddCatalog](#) to add the [catalog file](#) to the system component and driver database. The installation program supplies the handle to the catalog administrator context that was obtained in step 1, a pointer to the fully qualified path of the catalog file, and a pointer to the name of the catalog file that the function uses to install a copy of the catalog file in the database. The function returns a handle to the catalog information context for the catalog file that was added to the database.
3. Calls [CryptCATAdminReleaseCatalogContext](#) to release the handle to the catalog information context for the catalog file. The installation program supplies the handle to the catalog administrator context that was obtained in step 1 and the handle to the catalog information context that was returned in step 2.
4. Calls [CryptCATAdminReleaseContext](#) to release the handle to the catalog administrator context. The application supplies the handle to the catalog administrator context that was obtained in step 1.

# Installing a Catalog File by using SignTool

11/2/2020 • 2 minutes to read • [Edit Online](#)

**SignTool** is not a redistributable tool and therefore cannot be included with a redistributed installation application. However, SignTool can be used on a computer that has SignTool already installed in a manner that complies with the Microsoft Software License Terms for the tool. A [catalog file](#) can be manually installed from a command line or installed by command script by using the following SignTool command:

```
SignTool catdb /v /u CatalogFileName.cat
```

Where:

- The **catdb** command configures SignTool to add or remove a catalog file from a catalog database. By default, SignTool adds the catalog file to the system component and driver database.
- The **/v** option configures SignTool to operate in verbose mode.
- The **/u** option configures SignTool to generate a unique name for the catalog file being added, if necessary, to prevent replacing an already existing catalog file that has the same name as *CatalogFileName.cat*.
- *CatalogFileName.cat* is the name of the catalog file.

# Installing an Unsigned Driver during Development and Test

11/2/2020 • 2 minutes to read • [Edit Online](#)

By default, 64-bit versions of Windows Vista and later versions of Windows will load a kernel-mode driver only if the kernel can verify the driver signature. However, this default behavior can be disabled during early driver development and for non-automated testing. Developers can use one of the following mechanisms to temporarily disable load-time enforcement of a valid driver signature. However, to fully automate testing of a driver that is installed by Plug and Play (PnP), the [catalog file](#) of the driver must be signed. Signing the driver is required because Windows Vista and later versions of Windows display a driver signing dialog box for unsigned drivers that require a system administrator to authorize the installation of the driver, potentially preventing any user without the necessary privileges from installing the driver and using the device. This PnP driver installation behavior cannot be disabled on Windows Vista and later versions of Windows.

## Use the F8 Advanced Boot Option

Windows Vista and later versions of Windows support the F8 Advanced Boot Option -- "Disable Driver Signature Enforcement" -- that disables load-time signature enforcement for a kernel-mode driver only for the current system session. This setting does not persist across system restarts.

## Attach a Kernel Debugger to Disable Signature Verification

Attaching an active kernel debugger to a development or test computer disables load-time signature enforcement for kernel-mode drivers. To use this debugging configuration, attach a debugging computer to a development or test computer, and enable kernel debugging on the development or test computer by running the following command:

```
bcdeedit -debug on
```

To use BCDEdit, the user must be a member of the Administrators group on the system and run the command from an elevated command prompt. To open an elevated Command Prompt window, create a desktop shortcut to *Cmd.exe*, select and hold (or right-click) the shortcut, and select **Run as administrator**.

## Enforcing Kernel-Mode Signature Verification in Kernel Debugging Mode

However, there are situations in which a developer might need to have a kernel debugger attached, yet also need to maintain load-time signature enforcement. For example, when a driver stack has an unsigned driver (such as a filter driver) that fails to load it may invalidate the entire stack. Because attaching a debugger allows the unsigned driver to load, the problem appears to vanish as soon as the debugger is attached. Debugging this type of issue may be difficult.

In order to facilitate debugging such issues, the [kernel-mode code signing policy](#) supports the following registry value:

```
HKLM\SYSTEM\CurrentControlSet\Control\CI\DebugFlags
```

This registry value is of type [REG\\_DWORD](#), and can be assigned a value based on a bitwise OR of one or more of the following flags:

**0x00000001**

This flag value configures the kernel to break into the debugger if a driver is unsigned. The developer or tester can

then choose to load the unsigned driver by entering **g** at the debugger prompt.

#### **0x00000010**

This flag value configures the kernel to ignore the presence of the debugger and to always block an unsigned driver from loading.

If this registry value does not exist in the registry or has a value that is not based on the flags described previously, the kernel always loads a driver in kernel debugging mode regardless of whether the driver is signed.

**Note** This registry value does not exist in the registry by default. You must create the value in order to debug the kernel-mode signature verification.

# Signing Drivers for Public Release

11/2/2020 • 2 minutes to read • [Edit Online](#)

Release-signing identifies the publisher of a kernel-mode binary (for example, driver or *.dll*) that loads into Windows Vista and later versions of Windows. Kernel-mode binaries are release-signed through either:

- A [WHQL Release Signature](#) obtained through the [Windows Logo Program](#).
- A release signature created through a [Software Publisher Certificate \(SPC\)](#).

To understand the steps that are involved in release-signing [driver packages](#), review the following topics:

## [Introduction to Release-Signing](#)

This topic describes the reasons why release-signing a driver package is important, and provides a high-level summary of the release-signing process.

## [How to Release-Sign a Driver Package](#)

This topic provides a high-level overview of the release-signing process, and reviews many examples of release-signing by using the *ToastPkg* sample driver package within the Windows Driver Kit (WDK).

For more information about the release-signing process, see the following topics:

### [WHQL Release Signature](#)

### [Release Certificates](#)

### [Release-Signing Driver Packages](#)

# Introduction to Release-Signing

12/1/2020 • 3 minutes to read • [Edit Online](#)

[Driver packages](#) should be release-signed for the following reasons:

- To ensure the authenticity, integrity, and reliability of driver packages.

Windows uses digital signatures to verify the identity of the publisher and to verify that the driver has not been altered since it was published.

- To provide the best user experience by facilitating automatic driver installation.

If a driver is not signed, the [Plug and Play \(PnP\) driver installation policy](#) requires that a system administrator manually authorize the installation of an unsigned driver, adding an extra step to the installation process. This extra step can be potentially confusing and bothersome to the average user.

- To run kernel-mode drivers on 64-bit versions of Windows Vista and later versions of Windows.

The [kernel-mode code signing policy](#) for 64-bit versions of Windows Vista and later requires that kernel-mode drivers be signed in order for the operating system to load the driver.

- To play back certain types of next-generation premium content, all kernel-mode components in Windows Vista and later versions of Windows must be signed. In addition, all the user-mode and kernel-mode components in the Protected Media Path (PMP) must comply with PMP signing policy. For information about PMP signing policy, see the white paper [Code-signing for Protected Media Components in Windows Vista](#).

The [Hardware Certification Kit \(HCK\)](#) has [test categories](#) for a variety of device types. If a test category for the device type is included in this list, the driver publisher should obtain a [WHQL release signature](#) for the driver package.

**Note** On Windows Server 2003, Windows XP, and Windows 2000, the INF file from the WHQL-signed [driver package](#) must use a [device setup class](#) that is defined in `%SystemRoot%/inf/Certclas.inf`. Otherwise, Windows treats the driver package as unsigned.

If a driver package is digitally-signed by WHQL, it can be distributed through the Windows Update program or other Microsoft-supported distribution mechanisms. WHQL signs the driver package [catalog file](#), but does not embed signatures in driver files. If a driver is a *boot-start driver* for 64-bit processors, the driver publisher must also [embed a signature](#) in the kernel-mode driver files before submitting the driver package to WHQL.

If the [Hardware Certification Kit \(HCK\)](#) does not have a [test category](#) for your device type, you must use the following types of digital signatures to [release-sign](#) driver packages on Windows Vista and later versions of Windows:

- To comply with the [kernel-mode code signing policy](#) of 64-bit versions of Windows Vista and later versions of Windows, you must use a [Software Publisher Certificate \(SPC\)](#) to sign a kernel-mode driver package. For *non-boot-start drivers*, you only have to sign the driver package's [catalog file](#). For a boot-start driver, you must embed an SPC signature in a kernel-mode driver file and, optionally, also sign the driver package's catalog file.
- To comply with the [PnP device installation signing requirements](#) of 32-bit versions of Windows Vista and later versions of Windows, you can use either an SPC or a [commercial release certificate](#) to sign a kernel-mode driver package's catalog file. These latter two signature types verify the authenticity and integrity of a driver, but unlike a WHQL release signature, do not verify the reliability of the driver.

An SPC and a commercial release certificate are collectively referred to as [release certificates](#) and a signature generated with a release certificate is referred to as a *release signature*.

For more information about the release-signing requirements and procedures, see [Release-Signing Driver Packages](#).

**Note** To understand the steps that are involved in release-signing driver packages, see [How to Release-Sign a Driver Package](#). This topic provides a summary of the release-signing process, and steps through many examples of release-signing by using the *ToastPkg* sample driver package within the Windows Driver Kit (WDK).

# How to Release-Sign a Driver Package

12/1/2020 • 2 minutes to read • [Edit Online](#)

This section provides the basic steps that you have to follow when you release-sign a [driver package](#). This includes the following:

- Obtaining a [Software Publisher Certificate \(SPC\)](#) from a commercial certificate authority (CA).
- Preparing a [driver package](#) for release-signing. This includes creating a [catalog file](#), which contains the digital signature for the driver package.
- Release-signing the driver package's catalog file.
- Release-signing a driver through an embedded signature. You have to embed a digital signature within the driver if the driver is a *boot-start driver*.

Each topic in this section describes a separate procedure in the release-signing process, and provides the general information that you have to understand about the procedure. In addition, each topic points you to other topics that provide detailed information about the procedure.

**Note** This section discusses the steps involved when a driver publisher has to manually release-sign a driver package. The [Hardware Certification Kit \(HCK\)](#) has [test categories](#) for a variety of device types. If a test category for the device type is included in this list, the driver publisher should obtain a [WHQL release signature](#) for the driver package instead of manually release-signing the driver package.

Throughout this section, separate computers are used for the various processes involved in release-signing a driver. These computers are referred to as follows:

## **Signing computer**

This is the computer that is used to release-sign a driver package for Windows Vista and later versions of Windows. This computer must be running Windows XP SP2 or later versions of Windows. To use the [driver signing tools](#), this computer must have the Windows Vista and later versions of the Windows Driver Kit (WDK) installed.

## **Test computer**

This is the computer that is used to install and test the release-signed driver package. This computer must be running Windows Vista or later versions of Windows.

When discussing the release-signing process, the topics of this section use the *ToastPkg* sample driver package. Within the WDK installation directory, the *ToastPkg* driver package is located in the *src\general\toaster\toastpkg* directory.

This section contains the following topics:

[Obtaining a Software Publisher Certificate \(SPC\)](#)

[Creating a Catalog File for Release-Signing a Driver Package](#)

[Release-Signing a Driver Package's Catalog File](#)

[Release-Signing a Driver Through an Embedded Signature](#)

[Verifying the Release-Signature](#)

[Configuring a Computer to Support Release-Signing](#)

[Installing a Release-Signed Driver Package](#)

# Obtaining a Software Publisher Certificate (SPC)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Release-signing requires:

- A Software Publisher Certificate (SPC), along with the certificate's public and private cryptographic keys.
- An associated cross-certificate.

The SPC is obtained through a commercial certificate authority (CA). The CA will also provide private and public keys for the certificate. In order to be used for creating digital signatures, the SPC and keys must be converted to a Personal Information Exchange (.pfx) file. For more information about this process, see [Personal Information Exchange \(.pfx\) Files](#).

Once the SPC and keys are stored in a .pfx file, they must be imported into the Personal certificate store on the signing computer. For more information, see [Importing an SPC into a Certificate Store](#).

Microsoft has issued one cross-certificate for each public key root certificate for CAs, which supports the use of Software Publisher Certificates for kernel-mode code signing. You must use correct cross-certificate when release-signing a driver package. To determine which cross-certificate is needed for release-signing, see [Determining an SPC's Cross-Certificate](#).

For a list of certification authorities that provide SPCs and for more information about cross-certificates, see [Cross-Certificates for Kernel Mode Code Signing](#). Follow the instructions on the CA's website for obtaining and installing the SPC and corresponding cross-certificate on the signing computer.

For more information about SPCs and their management, see [Software Publisher Certificate \(SPC\)](#).

# Personal Information Exchange (.pfx) Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

To be used for release signing, a Software Publisher Certificate (SPC), and its private and public keys, must be stored in a Personal Information Exchange (.pfx) file. However, some certificate authorities (CAs) use different file formats to store this data. For example, some CAs store the certificate's private key in a Private Key (.pvk) file and store the certificate and public key in a .spc or .cer file.

If the CA issued an .spc and its keys in non-.pfx files, you must convert and store the files in a .pfx file before they can be used for release-signing. The **Pvk2Pfx** tool is used to perform this conversion.

The following command-line example converts a .pvk file that is named *abc.pvk* and a .spc that is named *abc.spc* into a .pfx file that is named *abc.pfx*.

```
Pvk2Pfx -pvk abc.pvk -pi pvkpassword -spc abc.spc -pfx abc.pfx -po pfxpassword -f
```

Where:

- The **-pvk** option specifies a .pvk file (*abc.pvk*).
- The **-pi** option specifies the password for the .pvk file (*pvkpassword*).
- The **-spc** option specifies the name and extension of the SPC file that contains the certificate. The file can be either an .spc file or a .cer file. In this example, the certificate and public key are in the *abc.spc* file.
- The **-pfx** option specifies the name of the .pfx file (*abc.pfx*). If this option is not specified, Pvk2Pfx opens an Export Wizard and ignores the -po and -f arguments.
- The **-po** option specifies a password for the .pfx file (*pfxpassword*). If this option is not specified, the specified .pfx file is assigned the same password that is associated with the specified .pvk file.
- The **-f** option configures Pvk2Pfx to replace an existing .pfx file if one exists.

For more information about SPCs and their management, see [Software Publisher Certificate \(SPC\)](#).

# Importing an SPC into a Certificate Store

11/2/2020 • 2 minutes to read • [Edit Online](#)

Once a Personal Information Exchange (.pfx) file is created to store a **Software Publisher Certificate (SPC)** and its private and public keys, the .pfx file must be imported into the Personal certificate store on the signing computer. For more information about .pfx files, see [Personal Information Exchange \(.pfx\) Files](#).

To import a .pfx file into the local Personal certificate store, do the following:

1. Start Windows Explorer and select and hold (or right-click) the .pfx file, then select Open to open the Certificate Import Wizard.
2. Follow the procedure in the Certificate Import Wizard to import the code-signing certificate into the Personal certificate store.

The certificate and private key are now available for SignTool to use.

Starting with Windows Vista, an alternative way to import the .pfx file into the local Personal certificate store is with the [CertUtil](#) command-line utility. The following command-line example uses CertUtil to import the abc.pfx file into the Personal certificate store:

```
certutil -user -p pfxpassword -importPFX abc.pfx
```

Where:

- The **-user** option specifies "Current User" Personal store.
- The **-p** option specifies the password for the .pfx file (*pfxpassword*).
- The **-importPFX** option specifies name of the .pfx file (*abc.pfx*).

Once the .pfx file is imported into the Personal certificate store on the signing computer, you can use [SignTool](#) to release-sign [driver packages](#).

# Determining an SPC's Cross-Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

In addition to obtaining a Software Publisher Certificate (SPC) from a commercial certificate authority (CA), you must obtain a cross-certificate that Microsoft issues. The cross certificate is used to verify that the CA that issued an SPC is a trusted root authority. Both a cross-certificate and an SPC are required for release-signing.

A cross-certificate is an X.509 certificate issued by a CA that signs the public key for the root certificate of another CA. Cross-certificates allow the kernel to have a single trusted Microsoft root authority, but also provide the flexibility to extend the chain of trust to commercial CAs that issue SPCs.

Before you can determine which cross-certificate is needed for release-signing, you must first import the Personal Information Exchange (.pfx) file, which stores a Software Publisher Certificate (SPC) and its private and public keys, into the Personal certificate store. For more information about this process, see [Importing an SPC into a Certificate Store](#).

Once the .pfx file is imported into the Personal store on the signing computer, do the following to determine which cross-certificate you can use with your SPC for release-signing.

1. Click **Start** and then click **Run**.
2. To start the MMC Certificates snap-in, type Certmgr.msc and press the **Enter** key.
3. Locate the signing certificate in the certificate store. The certificate should be listed in one of the following locations, depending on how it was installed:
  - Current User->Personal->Certificates store
  - Local Machine->Certificates store
4. To open the **Certificate** dialog box, double click the certificate.
5. In the **Certificate** dialog box, select the **Certification Path** tab, and then select the top-most certificate in the certification path.

This is the CA that is the issuing root authoring for your certificate.

6. To view the root authority certificate, select **View Certificate**, and then click the **Details** property tab.
7. Find the **Issuer Name** and **Thumbprint** for the issuing CA of this certificate. Locate the corresponding cross-certificate in the "Root Authority Cross Certificate List" section of the [Microsoft Cross-Certificates for Windows Vista Kernel Mode Code Signing](#) white paper.
8. Download the related cross-certificate from the "Root Authority Cross Certificate List" section and use this cross-certificate when digitally signing [driver packages](#).

For more information about SPCs and their management, see [Software Publisher Certificate \(SPC\)](#).

# Creating a Catalog File for Release-Signing a Driver Package

12/5/2018 • 2 minutes to read • [Edit Online](#)

The process of creating a [catalog file](#) is the same for test-signing and release-signing a [driver package](#). For more information about this process, see [Creating a Catalog File for Test-Signing a Driver Package](#).

# Release-Signing a Driver Package's Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

Once the [catalog file](#) for a [driver package](#) is created or updated, the catalog file can be signed through [SignTool](#).

Once signed, the digital signature stored within the catalog file is invalidated if any components of the driver package are modified.

When digitally signing a catalog file, SignTool saves the digital signature within the catalog file. The components of the driver package are not changed by SignTool. However, since the catalog file contains hashed values of the components of the driver package, the digital signature within the catalog file is maintained as long as the components hash to the same value.

SignTool can also add a time stamp to the digital signature. The time stamp lets you determine when a signature was created and supports more flexible certificate revocation options, if necessary.

The following command line shows how to run SignTool to do the following:

- Release-sign the *tstamd64.cat* catalog file of the *ToastPkg* sample [driver package](#). For more information about how this [catalog file](#) was created, see [Creating a Catalog File for Release-Signing a Driver Package](#).
- Use a [Software Publisher Certificate \(SPC\)](#) issued by a commercial certificate authority (CA).
- Use a compatible cross-certificate for SPC.
- Assign a time stamp to the digital signature through a time stamp authority (TSA).

To release-sign the *tstamd64.cat* catalog file, run the following command line:

```
SignTool sign /v /fd sha256 /ac MSCV-VSClass3.cer /s MyPersonalStore /n contoso.com /t http://timestamp.digicert.com tstamd64.cat
```

Where:

- The **sign** command configures SignTool to sign the specified catalog file, *tstamd64.cat*.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/fd** option specifies the file digest algorithm to use for creating file signatures. The default is SHA1.
- The **/ac** option specifies the name of the file which contains the cross-certificate (*MSCV-VSClass3.cer*) obtained from the CA. Use the full path name if the cross-certificate is not in the current directory.
- The **/s** option specifies the name of the Personal certificate store (*MyPersonalStore*) that contains the SPC.
- The **/n** option specifies the name of the SPC (*Contoso.com*) that is installed in the specified certificate store.
- The **/t** option specifies URL of the TSA (<http://timestamp.digicert.com>) which will timestamp the digital signature.

## IMPORTANT

Including a time stamp provides the necessary information for key revocation in case the signer's code signing private key is compromised.

- *tstampd64.cat* specifies the name of the catalog file, which will be digitally-signed.

For more information about SignTool and its command-line arguments, see [SignTool](#).

For more information about release-signing driver packages, see [Release-Signing Driver Packages](#).

# Release-Signing a Driver through an Embedded Signature

11/2/2020 • 2 minutes to read • [Edit Online](#)

A signed [catalog file](#) is all that you must have to correctly install and load most [driver packages](#). However, embedded-signing might also be an option. Embedded-signing refers to adding a digital signature to the driver's binary image file itself, instead of saving the digital signature in a catalog file. As a result, the driver's binary image is modified when the driver is embedded-signed.

Embedded-signing of kernel-mode binaries (for example, drivers and associated .dll files) are required whenever:

- The driver is a *boot-start driver*. In 64-bit versions of Windows Vista and later versions of Windows, the [kernel-mode code signing requirements](#) state that a *boot-start driver* must have an embedded signature. This is required regardless of whether the driver's driver package has a digitally-signed catalog file.
- The driver is installed through a driver package that does not include a catalog file.

As with [catalog files](#), the [SignTool](#) tool is used to embed a digital signature within kernel-mode binary files by using a test certificate. The following command line shows how to run SignTool to do the following:

- Test-sign the 64-bit version of the Toastpkg sample's binary file, toaster.sys. Within the WDK installation directory, this file is located in the *src\general\toaster\toastpkg\toastcd\amd64* directory.
- Use a [Software Publisher Certificate \(SPC\)](#) issued by a commercial certificate authority (CA).
- Use a compatible cross-certificate for SPC.
- Assign a time stamp to the digital signature through a time stamp authority (TSA).

To test-sign the *toaster.sys* file, run the following command line:

```
SignTool sign /v /fd sha256 /ac MSCV-VSClass3.cer /s MyPersonalStore /n contoso.com /t http://timestamp.digicert.com amd64\toaster.sys
```

Where:

- The **sign** command configures SignTool to sign the specified kernel-mode binary file, *amd64\toaster.sys*.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/fd** option specifies the file digest algorithm to use for creating file signatures. The default is SHA1.
- The **/ac** option specifies the name of the file which contains the cross-certificate (*MSCV-VSClass3.cer*) obtained from the CA. Use the full path name if the cross-certificate is not in the current directory.
- The **/s** option specifies the name of the Personal certificate store (*MyPersonalStore*) that contains the SPC.
- The **/n** option specifies the name of the certificate (*Contoso.com*) that is installed in the specified certificate store.
- The **/t** option specifies URL of the TSA (<http://timestamp.digicert.com>) which will timestamp the digital signature.

**IMPORTANT**

Including a time stamp provides the necessary information for key revocation in case the signer's code signing private key is compromised.

- *amd64\toaster.sys* specifies the name of the kernel-mode binary file which will be embedded-signed.

For more information about SignTool and its command-line arguments, see [SignTool](#).

For more information about release-signing a driver through an embedded signature, see [Release-Signing Driver Packages](#) and [Release-Signing a Driver File](#).

# Verifying the Release-Signature

11/2/2020 • 2 minutes to read • [Edit Online](#)

After a [driver package](#) is release-signed, the [SignTool](#) tool can be used to verify the signatures of:

- Individual files within the driver package.
- Kernel-mode binaries, such as drivers, which have been embedded-signed.

The examples in this topic use the 64-bit version of the Toastpkg sample's binary file, *toaster.sys*. Within the WDK installation directory, this file is located in the *src\general\toaster\toastpkg\toastcd\amd64* directory.

The following example verifies the signature of *toaster.sys* in the *tstamd64.cat* release-signed [catalog file](#):

```
Signtool verify /kp /v /c tstamd64.cat amd64\toaster.sys
```

Where:

- The **verify** command configures SignTool to verify the signature in the specified [catalog file](#) (*tstamd64.cat*).
- The **/kp** option configures SignTool to verify that the kernel policy has been met.
- The **/v** option configures SignTool to print execution and warning messages.
- The **/c** option specifies the driver package's [catalog file](#) which was released-signed (*tstamd64.cat*). If you are verifying the digital signature of an embedded-signed driver, do not use this option.
- *amd64\toaster.sys* is the name of the file to be verified.

Under the output from this command labeled "Signing Certificate Chain", you should verify that the following is true:

- The root of the certificate chain for kernel policy is issued to and by the Microsoft Code Verification Root.
- The cross-certificate, which is issued to the Class 3 Public Primary Certification Authority, is also issued by the Microsoft Code Verification Root.

For a signed catalog file, the Default Authenticode verification policy signature can also be verified on any kernel-mode binary file within the driver package. This ensures that the file appears as signed in the user-mode Plug and Play installation dialog boxes and the MMC Device Manager snap-in.

**Note** This example is used only for verification of release-signed [catalog files](#) and not embedded-signed kernel-mode binary files.

The following example verifies the Default Authenticode verification policy of *toaster.sys* in the *tstamd64.cat* signed catalog file:

```
Signtool verify /pa /v /c tstamd64.cat amd64\toaster.sys
```

Where:

- The **verify** command configures SignTool to verify the signature in the specified file.
- The **/pa** option configures SignTool to verify that the Authenticode verification policy has been met.

- The **/v** option configures SignTool to print execution and warning messages.
- The **/c** option specifies the driver package's **catalog file** that was released-signed (*tstampd64.cat*).
- *amd64\toaster.sys* is the name of the file to be verified.

Under the output from this command labeled "Signing Certificate Chain," you should verify that the default Authenticode certificate chain is issued to and by a Class 3 Public Primary Certification Authority.

You can also verify the digital signature of the catalog file itself through Windows Explorer by following these steps:

- Right-click the **catalog file** and select **Properties**.
- For digitally signed files, the file's **Properties** dialog box has an additional **Digital Signature** tab, on which the signature, time stamp, and details of the certificate that was used to sign the file appear.

For more information about how to release-sign driver packages, see [Release-Signing Driver Packages](#) and [Verifying the SPC Signature of a Catalog File](#).

# Configuring a Computer to Support Release-Signing

12/5/2018 • 2 minutes to read • [Edit Online](#)

Before a release-signed driver package can be installed on a computer, follow these steps to configure the computer to support release signing:

- Disable the TESTSIGNING Boot Configuration Option. For more information about this option, see [the TESTSIGNING Boot Configuration Option](#).
- Enable Code Integrity event logging and system auditing to help troubleshoot problems during release-signed driver installation and loading. For more information, see [Enabling Code Integrity Event Logging and System Auditing](#).

After the computer is reconfigured to enable release-signing, restart the computer for the changes to take effect.

# Installing a Release-Signed Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

The release-signed [driver package](#) can be installed on the computer through:

- The [DevCon](#) tool, which is a WDK command line tool for installing drivers. For more information, see [Using the DevCon Tool to Install a Driver Package](#).

You can troubleshoot problems with released-signed driver installation and loading through the methods described in [Troubleshooting Install and Load Problems with Signed Driver Packages](#).

# WHQL Release Signature

11/2/2020 • 2 minutes to read • [Edit Online](#)

Driver packages that pass [Windows Hardware Certification Kit \(HCK\)](#) testing can be digitally-signed by WHQL. If your driver package is digitally-signed by WHQL, it can be distributed through the [Windows Update](#) program or other Microsoft-supported distribution mechanisms.

Obtaining a WHQL release signature is part of the [Windows Hardware Certification Kit \(HCK\)](#). A WHQL release signature consists of a digitally-signed [catalog file](#). The digital signature does not change the driver binary files or the INF file that you submit for testing.

Obtaining a WHQL release signature consists of the following:

- Testing the [driver package](#) with the Windows HCK to verify that the driver package is compatible with Microsoft Windows. Once the HCK is installed, the Driver Test Manager (DTM) is run to test and verify the driver package. For more information, see the [Windows Hardware Certification Kit \(HCK\)](#).
- Submitting DTM test logs to the Windows Quality Online Services to obtain a WHQL release signature for the driver package. For more information, see the [Windows Hardware Certification Kit \(HCK\)](#).

For more information about WHQL, see the [Windows Hardware Quality Labs](#) website.

**Note** WHQL does not embed signatures in driver files. You can embed a signature in a driver file using a third-party commercial [release certificate](#). Embed the signature in the driver file before submitting the driver package to WHQL.

# Release Certificates

11/2/2020 • 2 minutes to read • [Edit Online](#)

To comply with the kernel-mode code signing policy of 64-bit versions of Windows Vista and later versions of Windows, you must obtain a [WHQL release signature](#) or use a [Software Publisher Certificate \(SPC\)](#) to sign the [catalog file](#) of kernel-mode driver packages.

If the driver is a *boot-start driver* for 64-bit systems, you must also [embed](#) an SPC signature in the driver file. This applies to any type of Plug and Play (PnP) or non-PnP kernel-mode driver.

The [Hardware Certification Kit \(HCK\)](#) has [test categories](#) for a variety of device types. To comply with the [PnP device installation requirements](#) of 32-bit versions of Windows Vista and later versions of Windows, you should obtain a WHQL release signature if the HCK has a test category for the device type. If you cannot obtain a WHQL release signature, you must use either an SPC or a [commercial release certificate](#) to sign a PnP kernel-mode driver.

# Software Publisher Certificate

11/2/2020 • 4 minutes to read • [Edit Online](#)

To comply with the [kernel-mode code signing policy](#) of 64-bit versions of Windows, you can sign a kernel-mode driver by using a Software Publisher Certificate (SPC). The SPC is obtained from a third-party certificate authority (CA) that is authorized by Microsoft to issue such certificates. Signatures generated with this type of SPC also comply with the [PnP driver signing requirements](#) for 64-bit and 32-bit versions of Windows.

**Note** Windows 10 for desktop editions (Home, Pro, Enterprise, and Education) and Windows Server 2016 kernel-mode drivers must be signed by the Windows Hardware Dev Center Dashboard and the Windows Hardware Dev Center Dashboard requires an EV certificate. For more info about these changes, see [Driver Signing Changes in Windows 10](#).

**Caution**

Starting in 2021, the majority of cross-certificates will begin to expire. Once these cross-certificates expire, code signing certificates that chain to these cross-certificates will no longer be able to create new kernel mode digital signatures. This will affect all versions of Windows. For more information, see [Deprecation of software publisher certificates and commercial release certificates](#).

## Cross-Certificates

In addition to obtaining an SPC, you must obtain a cross-certificate that is issued by Microsoft. The cross certificate is used to verify that the CA that issued an SPC is a trusted root authority. A cross-certificate is an X.509 certificate issued by a CA that signs the public key for the root certificate of another CA. Cross-certificates allow the system to have a single trusted Microsoft root authority, but also provide the flexibility to extend the chain of trust to commercial CAs that issue SPCs.

Publishers do not have to distribute a cross-certificate with a [driver package](#). The cross-certificate is included with the digital signature for a driver package's [catalog file](#) or the signature that is embedded in a driver file. Users who install the driver package do not have to perform any additional configuration steps caused by the use of cross-certificates.

For a list of certification authorities that provide SPCs and for more information about cross-certificates, see [Cross-Certificates for Kernel Mode Code Signing](#). Follow the instructions on the certification authority's website on how to obtain and install the SPC and corresponding cross-certificate on the computer with which you will sign a driver. In addition, you must add the SPC information to the Personal certificate store of the local computer that signs drivers. For information about this requirement, see the [Installing SPC Information in the Personal Certificate Store](#).

## Installing SPC Information in the Personal Certificate Store

In order to use an SPC to sign a driver in a manner that complies with the [kernel-mode code signing policy](#), the certificate information must first be contained in a Personal Information Exchange (.pfx) file. The information that is contained in the .pfx file must then be added to the Personal certificate store of the local computer that signs a driver.

A CA might issue a .pfx file that contains the necessary certificate information. If so, you can add the .pfx file to the Personal certificate store by following the instructions described in [Installing a .pfx File in the Personal Certificate Store](#).

However, a CA might issue the following pairs of files:

- A *.pvk* file that contains the private key information.
- An *.spc* or *.cer* file that contains the public key information.

In this case, the pair of files (a *.pvk* and an *.spc* or a *.pvk* and a *.cer*) must be converted to a *.pfx* file in order to add the certificate information to the Personal certificate store.

To create a *.pfx* file from the pair of files issued by the CA, follow these instructions:

- To convert a *.pvk* file and an *.spc* file to a *.pfx* file, use the following [Pvk2Pfx](#) command at a command prompt:

```
Pvk2Pfx -pvk mypvkfile.pvk -pi mypvkpassword -spc myspcfile.spc -pfx mypfxfile.pfx -po pfxpassword  
-f
```

- To convert a *.pvk* file and a *.cer* file, to a *.pfx* file, use the following Pvk2Pfx command at a command prompt:

```
Pvk2Pfx -pvk mypvkfile.pvk -pi mypvkpassword -spc mycerfile.cer -pfx mypfxfile.pfx -po pfxpassword  
-f
```

The following describes the parameters that are used in the [Pvk2Pfx](#) command:

- The **-pvk** *mypvkfile.pvk* parameter specifies a *.pvk* file.
- The **-pi** *mypvkpassword* option specifies the password for the *.pvk* file.
- The **-spc** *myspcfile.spc* parameter specifies an *.spc* file or the **-spc** *mycerfile.cer* parameter specifies a *.cer* file.
- The **-pfx** *mypfxfile.pfx* option specifies the name of a *.pfx* file.
- The **-po** *pfxpassword* option specifies a password for the *.pfx* file.
- The **-f** option configures Pvk2Pfx to replace a existing *.pfx* file if one exists.

## Installing a *.pfx* File in the Personal Certificate Store

After obtaining a *.pfx* file from a CA, or creating a *.pfx* file from a *.pvk* and either an *.spc* or a *.cer* file, add the information in the *.pfx* file to the Personal certificate store of the local computer that signs the driver. You can use the Certificate Import Wizard to import the information in the *.pfx* file to the Personal certificate store, as follows:

1. Locate the *.pfx* file in Windows Explorer and select and hold (or right-click) the file and then select Open to open the Certificate Import Wizard.
2. Follow the procedure in the Certificate Import Wizard to import the code-signing certificate into the Personal certificate store.

# Commercial Release Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

## Caution

Starting in 2021, the majority of cross-certificates will begin to expire. Once these cross-certificates expire, code signing certificates that chain to these cross-certificates will no longer be able to create new kernel mode digital signatures. This will affect all versions of Windows. For more information, see [Deprecation of software publisher certificates and commercial release certificates](#).

A *commercial release certificate* refers to a digital certificate that a publisher obtains from a trusted, third-party, commercial certification authority (CA) that is a member of the Microsoft Root Certificate Program. GTE and VeriSign, Inc. are two examples of such CAs. A CA that is a member of this program validates the identity and entitlement of an applicant and then issues a certificate that the applicant uses to sign its drivers. The signing process stamps the driver with the publisher's identity and can be used to verify that the driver has not been modified since it was signed.

For security reasons, you should not use a digital certificate that is used to release-sign drivers to test-sign drivers. You should obtain separate digital certificates to use for release-signing and for test-signing. For information about how to manage digital certificates, see [Managing the Digital Signature or Code Signing Keys](#).

Follow the instructions that are provided by the CA about how to obtain and install the release certificate on a computer that you will be using to sign a driver.

For more information about how to obtain a digital certificate from a CA, see the [Microsoft Root Certificate Program Members](#) website.

# Deprecation of Software Publisher Certificates, Commercial Release Certificates, and Commercial Test Certificates

12/1/2020 • 4 minutes to read • [Edit Online](#)

The [Microsoft Trusted Root Program](#) no longer supports root certificates that have kernel mode signing capabilities.

For policy requirements, see [Windows 10 Kernel Mode Code Signing Requirements](#).

Existing [cross-signed root certificates](#) with kernel mode code signing capabilities will continue working until expiration. As a result, all [software publisher certificates](#), [commercial release certificates](#), and [commercial test certificates](#) that chain back to these root certificates also become invalid on the same schedule. To get your driver signed, first [Register for the Windows Hardware Dev Center program](#).

## Frequently asked questions

- [What is the expiration schedule of the trusted cross-certificates?](#)
- [What alternatives to cross signed certificates are available for testing drivers?](#)
- [What will happen to my existing signed driver packages?](#)
- [Is there a way to run production driver packages without exposing it to Microsoft?](#)
- [Does every new version of my driver package need to be resubmitted to Hardware Dev Center?](#)
- [Will we continue to be able to sign non-driver code with our existing 3rd party issued certificates after 2021?](#)
- [Will I be able to continue using my EV certificate for signing submissions to Hardware Dev Center?](#)
- [How do I know if my signing certificate will be impacted by these expirations?](#)
- [How can we automate Microsoft Test Signing to work with our build processes?](#)
- [Starting in 2021, will Microsoft be the sole provider of production kernel mode code signatures?](#)
- [Hardware Dev Center doesn't provide driver signing for Windows XP, how can I have my drivers run in XP?](#)
- [How do production signing options differ by Windows version?](#)

### What is the expiration schedule of the trusted cross-certificates?

The majority of cross-signed root certificates will expire in 2021, according to the following schedule:

COMMON NAME	EXPIRATION DATE
VeriSign Class 3 Public Primary Certification Authority - G5	2/22/2021
thawte Primary Root CA	2/22/2021
GeoTrust Primary Certification Authority	2/22/2021
GeoTrust Primary Certification Authority - G3	2/22/2021
thawte Primary Root CA - G3	2/22/2021
VeriSign Universal Root Certification Authority	2/22/2021

COMMON NAME	EXPIRATION DATE
TC TrustCenter Class 2 CA II	4/11/2021
COMODO RSA Certification Authority	4/11/2021
UTN-USERFirst-Object	4/11/2021
DigiCert Assured ID Root CA	4/15/2021
DigiCert High Assurance EV Root CA	4/15/2021
DigiCert Global Root CA	4/15/2021
Entrust.net Certification Authority (2048)	4/15/2021
GlobalSign Root CA	4/15/2021
Go Daddy Root Certificate Authority - G2	4/15/2021
Starfield Root Certificate Authority - G2	4/15/2021
NetLock Arany (Class Gold) Fotanúsítvány	4/15/2021
NetLock Arany (Class Gold) Fotanúsítvány	4/15/2021
NetLock Platina (Class Platinum) Fotanúsítvány	4/15/2021
Security Communication RootCA1	4/15/2021
StartCom Certification Authority	4/15/2021
Certum Trusted Network CA	4/15/2021
COMODO ECC Certification Authority	4/11/2021

### What alternatives to cross-signed certificates are available for testing drivers?

For all options below, the [TESTSIGNING boot option](#) must be enabled.

- [MakeCert Process](#)
- [WHQL Test Signature Program](#)
- [Enterprise CA Process](#)

For testing drivers at boot, see [How to Install a Test-signed Driver Required for Windows Setup and Boot](#).

For more info, see [Signing drivers during development and test](#).

### What will happen to my existing signed driver packages?

As long as driver packages are timestamped before the expiration date of the intermediate certificate, they will continue working.

### Is there a way to run production driver packages without exposing it to Microsoft?

No, all production driver packages must be submitted to, and signed by Microsoft.

## **Does every new Production version of a driver package need to be signed by Microsoft?**

Yes, every time a Production level driver package is rebuilt, it must be signed by Microsoft.

## **Will we continue to be able to sign non-driver code with our existing 3rd party issued certificates after 2021?**

Yes, these certificates will continue to work until they expire. Code which is signed using these certificates will only be able to run in user mode, and will not be allowed to run in the kernel, unless it has a valid Microsoft signature.

## **Will I be able to continue using my EV certificate for signing submissions to Hardware Dev Center?**

Yes, EV certificates will continue to work until they expire. If you sign a kernel-mode driver with an EV certificate after the expiration of the cross-certificate that issued that EV certificate, the resulting driver will not load, run, or install.

## **How do I know if my signing certificate will be impacted by these expirations?**

If your Cross Certificate Chain ends in `Microsoft Code Verification Root`, your signing certificate is affected.

To view the cross certificate chain, run `signtool verify /v /kp <mydriver.sys>`. For example:

```
Signature Index: 0 (Primary Signature)
Hash of File (sha1): 3C2ECE86170689EC14DA144E745836E8CC745C15

Signing Certificate Chain:
Issued to: AddTrust External CA Root
Issued by: AddTrust External CA Root
Expires: Sat May 30 03:48:38 2020
SHA1 hash: 02FAF3E291435468607857694DF5E45B68851868

Issued to: Intel External Basic Policy CA
Issued by: AddTrust External CA Root
Expires: Sat May 30 03:48:38 2020
SHA1 hash: 7EA2490DF8EA1F80134498CAF93DDE770C4E195A

Issued to: Intel External Basic Issuing CA 3B
Issued by: Intel External Basic Policy CA
Expires: Thu Feb 08 15:31:23 2018
SHA1 hash: 4788CB7431BF91FAB995F8A6DDF5A9EFA817A892

Issued to: Intel Corporation - Client Components Group
Issued by: Intel External Basic Issuing CA 3B
Expires: Thu May 25 09:52:07 2017
SHA1 hash: BEE66E16EE0ED55909DB01263B5134979397B0CC

The signature is timestamped: Mon Feb 02 02:00:37 2015
Timestamp Verified by:
Issued to: Thawte Timestamping CA
Issued by: Thawte Timestamping CA
Expires: Thu Dec 31 16:59:59 2020
SHA1 hash: BE364A4562FB2EE05DBB3D32323ADF445084ED656

Issued to: Symantec Time Stamping Services CA - G2
Issued by: Thawte Timestamping CA
Expires: Wed Dec 30 16:59:59 2020
SHA1 hash: 6C07453FFDDA08883707C09B82FB3D15F35336B1

Issued to: Symantec Time Stamping Services Signer - G4
Issued by: Symantec Time Stamping Services CA - G2
Expires: Tue Dec 29 16:59:59 2020
SHA1 hash: 65439929B67973EB192D6FF243E6767ADF0834E4

Cross Certificate Chain:
Issued to: Microsoft Code Verification Root
Issued by: Microsoft Code Verification Root
Expires: Sat Nov 01 06:54:03 2025
SHA1 hash: 8FBE4D070EF8AB18CAF2A9D5CCAEE7282A2C66B3

Issued to: AddTrust External CA Root
Issued by: Microsoft Code Verification Root
Expires: Tue Aug 15 13:36:30 2023
SHA1 hash: A75AC657AA7A4CDFESF9DE393E69EFCAB659D250

Issued to: Intel External Basic Policy CA
Issued by: AddTrust External CA Root
Expires: Sat May 30 03:48:38 2020
SHA1 hash: 7EA2490DF8EA1F80134498CAF93DDE770C4E195A

Issued to: Intel External Basic Issuing CA 3B
Issued by: Intel External Basic Policy CA
Expires: Thu Feb 08 15:31:23 2018
SHA1 hash: 4788CB7431BF91FAB995F8A6DDF5A9EFA817A892

Issued to: Intel Corporation - Client Components Group
Issued by: Intel External Basic Issuing CA 3B
Expires: Thu May 25 09:52:07 2017
SHA1 hash: BEE66E16EE0ED55909DB01263B5134979397B0CC
```

## **How can we automate Microsoft Test Signing to work with our build processes?**

Your build processes can call the [Hardware Dev Center API](#).

For samples that show usage, see the [Surface Dev Center Manager](#) repository.

## **Starting in 2021, will Microsoft be the sole provider of production kernel mode code signatures?**

Yes.

## **Hardware Dev Center doesn't provide driver signing for Windows XP, how can I have my drivers run in XP?**

Drivers can still be signed with a 3rd party issued code signing certificate. However, the certificate that signed the driver must be imported into the `Local Computer Trusted Publishers` certificate store on the target computer. See [Trusted Publishers Certificate Store](#) for more information.

## How do production signing options differ by Windows version?

DRIVER RUNS ON	DRIVERS SIGNED BEFORE JULY 1 2021 BY	DRIVER SIGNED ON OR AFTER JULY 1 2021 BY
Windows Server 2008 and later, Windows 7, Windows 8	WHQL or cross-signed drivers	WHQL or drivers cross-signed before July 1 2021
Windows 10	WHQL or attested	WHQL or attested

If you have challenges signing your driver with WHQL, please report the specifics using one of the following:

- Use the Microsoft Collaborate portal, available through the [Microsoft Partner Center Dashboard](#), to create a feedback bug.
- Go to [Windows hardware engineering support](#), select the **Contact us** tab, and in the **Developer support topic** dropdown, select **HLK/HCK**. Then select **Submit an incident**.

## Related information

- [Register for the Hardware Program](#)
- [Software Publisher Certificate](#)
- [Commercial Release Certificate](#)
- [Commercial Test Certificate](#)

# Release-Signing Driver Packages

12/1/2020 • 5 minutes to read • [Edit Online](#)

In this section, a computer that signs drivers for release on Windows Vista and later versions of Windows is referred to as the *signing computer*. The signing computer must be running Windows XP SP2 or later versions of the Windows operating system. For example, a driver intended for release on Windows 7 can be signed on a computer that is running Windows Vista.

In addition, the signing computer must have the [driver signing tools](#) installed.

**Note** You must use the version of the [SignTool](#) tool that is provided in the Windows Vista and later versions of the Windows Driver Kit (WDK). Earlier versions of this tool do not support the kernel-mode code signing policy for Windows Vista and later versions of Windows.

To comply with the [kernel-mode code signing policy](#) and the [Plug and Play \(PnP\) device installation signing requirements](#) of Windows Vista and later versions of Windows, sign a driver for release as follows, based on the type of driver.

**Note** The Windows code-signing policy requires that a signed [catalog file](#) for a driver be installed in the system component and driver database. PnP device installation automatically installs the catalog file of a PnP driver in the driver database. However, if you use a signed catalog file to sign a non-PnP driver, the installation application that installs the driver must also install the catalog file in the driver database.

## PnP Kernel-Mode Boot-Start Driver

To comply with the [kernel-mode code signing policy](#) of 64-bit versions of Windows Vista and later versions of Windows, embed a signature in the *boot-start driver* file as follows:

1. [Release-sign the driver file](#) with a Software Publisher Certificate (SPC).
2. [Verify the SPC signature of the driver file](#).

Starting with Windows Vista, embedding a signature in a *boot-start driver* file is optional for 32-bit versions of Windows. Although Windows will check whether a kernel-mode driver file has an embedded signature, an embedded signature is not required.

To comply with the [PnP device installation signing requirements](#) of Windows Vista and later versions of Windows, you must obtain a signed [catalog file](#) or sign the catalog file of the [driver package](#). If a driver file will also include an embedded signature, embed the signature in the driver file before signing the driver package's catalog file.

If the [Hardware Certification Kit \(HCK\)](#) has a test program for the driver, obtain a [WHQL Release Signature](#) for the driver package. If the HCK does not have a test program for the driver, [create a catalog file](#) and sign the [catalog file](#) as follows:

### **Signing a catalog file for 64-bit versions**

You can sign a catalog file for 64-bit operating systems as follows:

1. [Sign the catalog file with the SPC](#) that was used to embed a signature in the driver file.
2. [Verify the SPC signature of the catalog file](#). You can verify the signature of a catalog file or you can verify the signatures of the individual file entries in the catalog file.

### **Signing a catalog file for 32-bit versions**

You can either sign the [catalog file](#) with an SPC, as described in the section for 64-bit versions, or with a

[commercial release certificate](#) as follows:

1. [Sign the catalog file with a commercial release certificate.](#)
2. [Verify the signature of the catalog file.](#) You can verify the signature of a catalog file or you can verify the signatures of the individual file entries in the catalog file.

### **Non-PnP Kernel-Mode Boot-Start Driver**

To comply with the kernel-mode code signing policy of 64-bit versions of Windows Vista and later versions of Windows, embed a signature in a *boot-start driver* file as follows:

1. [Release-sign the driver file](#) with an SPC.
2. [Verify the SPC signature of the driver file.](#)

Starting with Windows Vista, embedding a signature in a *boot-start driver* file is optional for 32-bit versions of Windows. Although Windows will check whether a kernel-mode driver file has an embedded signature, an embedded signature is not required.

The PnP device installation signing requirements do not apply to non-PnP drivers.

### **PnP Kernel-Mode Driver that is not a Boot-Start Driver**

The kernel-mode code signing policy on 64-bit versions of Windows Vista and later versions of Windows does not require a non-boot PnP driver have an embedded signature. However, if the driver file will include an embedded signature, embed the signature in the driver file before signing the [driver package's catalog file](#).

To comply with the PnP device installation signing requirements, you must obtain a signed catalog file or sign the catalog file of the driver package.

If the [Hardware Certification Kit \(HCK\)](#) has a test program for the driver, obtain a [WHQL release signature](#) for the driver package. If the HCK does not have a test program for the driver, [create a catalog file](#) and sign the [catalog file](#) in the same manner as described in this section for signing the catalog file of a PnP kernel-mode *boot-start driver*.

### **Non-PnP Kernel-Mode Driver that is not a Boot-Start Driver**

To comply with the kernel-mode code signing policy of 64-bit versions Windows Vista and later versions of Windows , embed a signature in the driver file or sign a catalog file for the [driver package](#).

Starting with Windows Vista, embedding a signature in a driver file is optional for 32-bit versions of Windows. Although Windows will check whether a kernel-mode driver file has an embedded signature, an embedded signature is not required.

The PnP device installation signing requirements do not apply to non-PnP drivers.

**Note** Using embedded signatures is generally simpler and more efficient than by using a signed catalog file. For more information about the advantages and disadvantages of using embedded signatures versus signed catalog files, see [Test Signing a Driver](#).

To embed a release signature in a file for a non-PnP kernel-mode driver that is not a *boot-start driver*, follow these steps:

1. [Sign the driver file](#) with an SPC.
2. [Verify the signature of the driver file.](#)

To release-sign a catalog file for a non-PnP kernel-mode driver that is not a *boot-start driver*, follow these steps:

1. [Create a catalog file for the non-PnP driver.](#)
2. [Sign the catalog file with an SPC.](#)

3. Verify the SPC signature of the catalog file.

If this type of driver has a signed [catalog file](#) instead of an embedded signature, the installation application that installs the driver must install the catalog file in the system component and driver database. For more information, see [Installing a Release-Signed Catalog File for a Non-PnP Driver](#).

# Release-Signing a Driver File

12/1/2020 • 2 minutes to read • [Edit Online](#)

Use the following **SignTool** command to embed a **Software Publisher Certificate (SPC)** signature in a driver file. To comply with the **kernel-mode code signing policy**, *boot-start driver* for 64-bit versions of Windows Vista and later versions of Windows must have an embedded SPC signature.

```
SignTool sign /v /ac CrossCertificateFile /s SPCCertificateStore /n SPCCertificateName /t  
http://timestamp.digicert.com DriverFileName.sys
```

Where:

- The **sign** command configures SignTool to embed a signature in the file *DriverFileName.sys*.
- The **/v** verbose option configures SignTool to print execution and warning messages.
- The **/ac** *CrossCertificateFile* option specifies the cross-certificate *.cer* file that is associated with the SPC that is specified by *SPCCertificateName*.
- The **/s** *SPCCertificateStore* option specifies the name of the Personal certificate store that holds the SPC that is specified by *SPCCertificateName*. As described in **Software Publisher Certificate (SPC)**, the certificate information must be contained in a *.pfx* file, and the information in the *.pfx* file must be added to the Personal certificate store of the local computer. The Personal certificate store is specified by the option **/s my**.
- The **/n** *SPCCertificateName* option specifies the name of the certificate in the *SPCCertificateStore* certificate store.
- The **/t** \*<https://timestamp.digicert.com> option supplies the URL to the publicly-available time-stamp server that DigiCert provides.
- *DriverFileName.sys* is the name of the driver file.

The following command embeds a signature in *Toaster.sys* that is generated from a certificate named "contoso.com" in the Personal "my" certificate store and the corresponding cross-certificate *Rsacertsrvcross.cer*. In addition, the signature is time-stamped by the time stamp service <https://timestamp.digicert.com>. In this example, *Toaster.sys* is in the *amd64* subdirectory under the directory in which the command is run.

```
SignTool sign /v /ac c:\lab\rsacertsrvcross.cer /s my /n contoso.com /t http://timestamp.digicert.com  
amd64\toaster.sys
```

# Verifying the Signature of a Release-Signed Driver File

11/2/2020 • 2 minutes to read • [Edit Online](#)

To verify an embedded signature in a driver file that is created by a [Software Publisher Certificate \(SPC\)](#), use the following [SignTool](#) command:

```
SignTool verify /v /kp DriverFileName.sys
```

Where:

- The **verify** command configures SignTool to verify the signature that is embedded in the driver file *DriverFileName.sys*.
- The **/v** option configures SignTool to print execution and warning messages.
- The **/kp** option configures SignTool to verify that the signature that is embedded in *DriverFileName.sys* complies with the [kernel-mode code signing policy](#) and the [PnP device installation signing requirements](#) for Windows Vista and later versions of Windows.
- *DriverFileName.sys* is the name of the driver file.

For example, the following command verifies that *Toaster.sys* has a valid embedded signature. In this example, *Toaster.sys* is in the *amd64* subdirectory under the directory in which the command is run.

```
SignTool verify /kp amd64\toaster.sys
```

# Creating a Catalog File for a PnP Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

To create an unsigned catalog file for a driver package, follow these steps:

1. Add the required INF **CatalogFile=FileName.Cat** entry or INF **CatalogFile.PlatformExtension=unique-filename.Cat** entries to the **INF Version section** of a **driver package's** INF file. For information about how to use platform extensions, see [Cross-Platform INF Files](#).
2. Use the **Inf2Cat** tool to verify that the driver package can be signed for the target platforms and to generate the unsigned **catalog files (.cat files)** that apply to the target platforms.

Use the following Inf2Cat command to create unsigned catalog files:

```
Inf2Cat /driver:DriverPath /os:WindowsVersionList
```

Where:

- The **/driver:DriverPath** parameter supplies the name of the directory where the **driver package** is located.
- The **/os:WindowsVersionList** parameter configures Inf2Cat to verify that the driver package complies with the signing requirements for the Windows versions that are specified by the list of Windows version identifiers.

## Examples

The following examples apply to the toaster **driver package** that is located in `c:\WindDDK\5739\src\general\toaster\toastpkg\toastcd`. The INF file for the toaster package is `Toastpkg.inf` and this INF file contains the following **CatalogFile** directives with platform extensions:

```
[Version]
...
CatalogFile.NTx86 = tostx86.cat
CatalogFile.NTIA64 = toastia64.cat
CatalogFile.NTAMD64 = tstampd64.cat
...
```

To generate `Tostx86.cat` for specific x86 versions of Windows, specify the Windows versions in `WindowsVersionList`. For example, the following Inf2Cat command verifies that the **driver package** can be signed for Windows 2000 and the x86 versions of Windows Vista, Windows Server 2003, and Windows XP.

```
Inf2Cat /driver:c:\WindDDK\5739\src\general\toaster\toastpkg\toastcd /os:2000,XP_X86,Server2003_X86,Vista_X86
```

To generate `Tstampd64.cat` for x64 versions of Windows, specify the Windows versions in `WindowsVersionList`. For example, the following Inf2Cat command verifies that the driver package can be signed for the x64 versions of Windows Vista, Windows Server 2003, and Windows XP.

```
Inf2Cat /driver:c:\WindDDK\5739\src\general\toaster\toastpkg\toastcd /os:XP_X64,Server2003_X64,Vista_X64
```

To generate `Tstampd64.cat` only for Windows Vista x64 Edition, specify only "Vista\_X64" in `WindowsVersionList`. For example, the following Inf2Cat command only verifies that the **driver package** can be signed for Windows

Vista x64 Edition.

```
Inf2Cat /driver:c:\WindDDK\5739\src\general\toaster\toastpkg\toastcd /os:Vista_X64
```

# Signing a Catalog File with an SPC

11/2/2020 • 2 minutes to read • [Edit Online](#)

Use the following **SignTool** command to sign the [catalog file](#) of a kernel-mode [driver package](#) with a [Software Publisher Certificate \(SPC\)](#). For 64-bit versions of Windows Vista and later versions of Windows, kernel-mode driver packages that do not have a [WHQL release signature](#) must be signed with an SPC signature to comply with both the [kernel-mode code signing policy](#) and the [PnP device installation signing requirements](#).

```
SignTool sign /v /ac CrossCertificateFile /s SPCCertificateStore /n SPCCertificateName /t  
http://timestamp.digicert.com CatalogFileName.cat
```

Where:

- The **sign** command configures SignTool to sign the catalog file *CatalogFileName.cat*.
- The **/v** verbose option configures SignTool to print execution and warning messages.
- The **/ac** *CrossCertificateFile* option specifies the cross-certificate .cer file that is associated with the Software Publisher Certificate (SPC) that is specified by *SPCCertificateName*.
- The **/s** *SPCCertificateStore* option specifies the name of the certificate store that holds the Software Publisher Certificate that is specified by *SPCCertificateName*. As described in [Software Publisher Certificate \(SPC\)](#), the certificate information must be contained in .pfx file, and the information in the .pfx file must be added to the Personal certificate store of the local computer. The Personal certificate store is specified by the option **/s my**.
- The **/n** *SPCCertificateName* option specifies the name of the certificate in the *SPCCertificateStore* certificate store.
- The **/t** `http://timestamp.digicert.com` option supplies the URL to the publicly-available time-stamp server that VeriSign provides.
- *CatalogFileName.cat* is the name of the catalog file.

For example, the following command signs the *Tstamd64.cat* catalog file with the SPC named "contoso.com" in the Personal "my" certificate store and the corresponding cross-certificate *Rsacertsrvcross.cer*. The signature is time-stamped by the service <https://timestamp.digicert.com>. In this example, the catalog file is in the same directory in which the command is run.

```
SignTool sign /v /ac c:\lab\rsacertsrvcross.cer /s my /n contoso.com /t http://timestamp.digicert.com  
tstamd64.cat
```

# Verifying the SPC Signature of a Catalog File

11/2/2020 • 2 minutes to read • [Edit Online](#)

To verify that a [catalog file](#) is signed by a valid [Software Publisher Certificate \(SPC\)](#) and corresponding cross-certificate, use the following [SignTool](#) command:

```
SignTool verify /v /kp CatalogFileName.cat
```

To verify that a file that is listed in a [driver package's catalog file](#) is signed by a [Software Publisher Certificate \(SPC\)](#) and corresponding cross-certificate, use the following SignTool command:

```
SignTool verify /v /kp /c CatalogFileName.cat DriverFileName
```

Where:

- The **verify** command configures SignTool to verify the signature of the [driver package's](#) catalog file *CatalogFileName.cat* or the driver file *DriverFileName*.
- The **/v** option configures SignTool to print execution and warning messages.
- The **/kp** option configures SignTool to verify that the signature of the file complies with the [kernel-mode code signing policy](#) and the [PnP device installation signing requirements](#) of Windows Vista and later versions of Windows.
- *CatalogFileName.cat* is the name of the catalog file for a driver package.
- *The /c CatalogFileName.cat* option specifies a catalog file that includes an entry for the file *DriverFileName*.
- *DriverFileName* is the name of a file that has an entry in the catalog file *CatalogFileName.cat*.

For example, the following command verifies that *Tstamd64.cat* has a digital signature that complies with the kernel-mode code signing policy and the PnP device installation signing requirements for Windows Vista and later versions of Windows. In this example, *Tstam64.cat* is in the same directory in which the command is run.

```
SignTool verify /kp tstamd64.cat
```

In the next example, the following command verifies that *Toastpkg.inf*, which has an entry in the catalog file *Tstamd64.cat*, has a digital signature that complies with the kernel-mode code signing policy and the PnP device installation signing requirements of Windows Vista and later versions of Windows. In this example, *Tstam64.cat* and *Toastpkg.inf* are in the same directory in which the command is run.

```
SignTool verify /kp /c tstamd64.cat toastpkg.inf
```

# Signing a Catalog File with a Commercial Release Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

Use the following **SignTool** command to sign the [catalog file](#) of a kernel-mode [driver package](#) with a [commercial release certificate](#).

## NOTE

For 32-bit versions of Windows Vista and later versions of Windows, only a commercial release certificate can be used to sign kernel-mode drivers to be released on these Windows versions.

```
SignTool sign /v /s CertificateStore /n CertificateName /t http://timestamp.digicert.com CatalogFileName.cat
```

Where:

- The **sign** command configures SignTool to sign the catalog file *CatalogFileName.cat*.
- The **/v** verbose option configures SignTool to print execution and warning messages.
- The **/s** *CertificateStore* option specifies the name of the certificate store that contains the *CertificateName* certificate. Follow the instructions of the certification authority (CA) that issued the certificate on how to install the certificate in the system certificate stores.
- The **/n** *CertificateName* option specifies the name of the certificate in the *CertificateStore* certificate store.
- The **/t** `http://timestamp.digicert.com` option supplies the URL to the publicly-available time-stamp server that DigiCert provides.
- *CatalogFileName.cat* is the name of the catalog file.

# Verifying the Signature of a Catalog File Signed by a Commercial Release Certificate

11/2/2020 • 2 minutes to read • [Edit Online](#)

To verify that a [catalog file](#) is signed by a valid [commercial release certificate](#), use the following [SignTool](#) command:

```
SignTool verify /v /pa CatalogFileName.cat
```

To verify that a file that is listed in a [driver package's](#) catalog file is signed by a valid commercial release certificate, use the following SignTool command:

```
SignTool verify /v /pa /c CatalogFileName.cat DriverFileName
```

Where:

- The **verify** command configures SignTool to verify the signature of the driver package's catalog file *CatalogFileName.cat* or the driver file *DriverFileName*.
- The **/v** option configures SignTool to print execution and warning messages.
- The **/pa** option configures SignTool to verify that the signature of the catalog file or driver file complies with the PnP driver installation requirements.
- *CatalogFileName.cat* is the name of the catalog file for a driver package.
- The **/c CatalogFileName.cat** option specifies a catalog file that includes an entry for the file *DriverFileName*.
- *DriverFileName* specifies the name of a file that has an entry in the catalog file *CatalogFileName.cat*.

# Creating a Catalog File for a Non-PnP Driver Package

11/2/2020 • 3 minutes to read • [Edit Online](#)

You can use the [MakeCat](#) tool to create a [catalog file](#) for a non-PnP driver package.

**Note** You must use the MakeCat tool only to create catalog files for driver packages that are not installed by using an INF file. If the driver package is installed by using an INF file, use the [Inf2Cat](#) tool to create the catalog file. Inf2Cat automatically includes all the files in the driver package that are referenced within the package's INF file. For more information about how to use the Inf2Cat tool, see [Using Inf2Cat to Create a Catalog File](#).

To create a catalog file, you must first manually create a Catalog Definition File ([..cdf](#)) that describes the catalog header attributes and file entries. After this file is created, you can then run the [MakeCat](#) tool to create a catalog file

## Creating a catalog file

To create a catalog file for a non-PnP driver package, follow these steps:

1. Use a text editor to create a [..cdffile](#) that lists the name of the [catalog file](#) to be created, its attributes, and the names of the files that are to be listed in the catalog file.
2. Use the [MakeCat](#) command-line tool to create the catalog file. For more information about the MakeCat tool, see the [Using MakeCat](#) website.
3. Install the catalog file on a computer on which the driver will be installed.

## Overview of the MakeCat tool

The MakeCat tool does the following when it processes the [..cdffile](#):

- Verifies the attributes of the [catalog file](#) that is defined by the [..cdffile](#), and adds the attributes to the catalog file.
- Verifies the attributes for each file that is listed within the [..cdffile](#), and adds the attributes to the catalog file.
- Generates a cryptographic hash, or *thumbprint*, of each of the listed files.
- Stores each file's thumbprint in the catalog file.

Use the following MakeCat command to create a catalog file.

```
MakeCat -v CatalogDefinitionFileName..cdf
```

Where:

- The `-v` option configures MakeCat to print execution and warning messages.
- *CatalogDefinitionFileName..cdf* is the name of the catalog definition file.

## Examples

The following example shows the contents of a typical catalog definition file that is named `Good..cdf`. The package to be cataloged contains two files, `File1` and `File2`. The resulting catalog file is named `Good.cat`.

```
[CatalogHeader]
Name=Good.cat
PublicVersion=0x0000001
EncodingType=0x00010001
CATATTR1=0x10010001:OSAttr:2:6.0
[CatalogFiles]
<hash>File1=File1
<hash>File2=File2
```

The options that are used in this example are described below. For more information about these options, see the [MakeCat](#) website.

Name=Good.cat

Specifies the name of the catalog file (*Good.cat*).

PublicVersion=0x0000001

Specifies the version of the catalog file.

EncodingType=0x00010001

Specifies the message encoding type that is used to generate the thumbprint. The value 0x00010001 specifies a message encoding type of PKCS\_7 ASN\_ENCODING | X509 ASN\_ENCODING.

CATATTR1=0x10010001:OSAttr:2:6.0

Specifies an attribute of the catalog file. To specify additional attributes, you must use separate CATATTR options, with each option assigned a unique numeric digit as a suffix. For example, use CATATT1 to specify one catalog file attribute and CATATT2 to specify another.

In this example, the attribute specified by using the CATATTR1 option has the following value:

0x10010001

Specifies the attribute to be the following:

- 0x10000000 - Authenticated attribute (signed, included in the thumbprint).
- 0x00010000 - Attribute is represented in plain text.
- 0x00000001 - Attribute is a name-value pair.

OSAttr:2:6.0

The OSAttr attribute specifies the target Windows version whose signing requirements are compatible with the [driver package](#). The attribute's value specifies the following:

- The value 2 specifies the catalog file is compatible with NT-based versions of the Windows operating system.
- The value 6.0 specifies the catalog file is compatible with Windows Vista. **Note** If the [driver package](#) is compatible with multiple Windows versions, you must use separate CATATTR options to specify the OSAttr attribute for each Windows version.

<hash>File1=File1

Specifies a reference tag for the file File1 which is referenced through the catalog file. The value <hash>File1 results in the tag being the file's cryptographic hash, or *thumbprint*.

<hash>File1=File2

Specifies a reference tag for the file, File2, which is referenced through the catalog file. The value <hash>File2 results in the tag being the file's thumbprint.

The following example shows how to generate the [catalog file](#), *Good.cat*, from a corresponding catalog definition file *Good.cdf*. Makecat saves *Good.cat* in the same folder where *File1* and *File2* are located.

```
MakeCat -v Good.cdf
```

# Installing a Release-Signed Catalog File for a Non-PnP Driver

11/2/2020 • 2 minutes to read • [Edit Online](#)

To comply with the [kernel-mode code signing policy](#) of 64-bit versions of Windows Vista and later versions of Windows, a non-boot, non-PnP kernel-mode driver must have either an embedded release signature or a release-signed [catalog file](#) that is installed in the system component and driver database. In addition, if you use a release-signed catalog file to authenticate a user-mode driver or a 32-bit non-PnP kernel-mode driver, the Windows code-signing policy requires that the catalog file be installed in the system component and driver database. PnP device installation automatically installs the catalog file of a PnP driver in the driver database. However, for non-PnP drivers, the installation application that installs a non-PnP driver must install the catalog file in the driver database.

To install a catalog file for a non-PnP driver that is released to the public, a redistributable installation application should use the [CryptCATAdminAddCatalog](#) cryptography function, as described in [Installing a Catalog File by using CryptCATAdminAddCatalog](#).

**Note** In general, a redistributable installation application cannot use the [SignTool](#) tool to install a catalog file because SignTool is not a redistributable tool.

**Tip** Using embedded signatures is generally easier and more efficient than by using a signed catalog file. For more information about the advantages and disadvantages of using embedded signatures versus signed catalog files, see [Test Signing a Driver](#).

# Detecting Driver Load Errors

2/21/2020 • 2 minutes to read • [Edit Online](#)

To detect whether a driver loaded, check the status of the device in Device Manager. If the [kernel-mode code signing policy](#) blocks a driver from loading because the driver is not correctly signed, the device status message will indicate that Windows could not load the driver and that the driver might be corrupted or missing. If this occurs, you can use [Code Integrity diagnostic system log events](#) to further diagnose the problem. For more info on code 52, see [CM\\_PROB\\_UNSIGNED\\_DRIVER](#).

For a full list of errors reported by Device Manager, see [Device Manager Error Messages](#).

For additional information that may help with the problem code, see [DEVPKEY\\_Device\\_ProblemStatus](#).

# Code Integrity Diagnostic System Log Events

12/5/2018 • 2 minutes to read • [Edit Online](#)

The Code Integrity component of Windows Vista and later versions of Windows enforces the requirement that kernel-mode drivers be signed in order to load. Windows Vista and later versions of Windows always generate Code Integrity operational events and optionally will generate additional system audit events and verbose diagnostic events that provide information about the status of driver signing, as follows:

- The Code Integrity operational log includes warning events that indicate that a kernel-mode driver failed to load because the driver signature could not be verified. Signature verification can fail for the following reasons:
  - An administrator preinstalled an unsigned driver, but Code Integrity subsequently blocked loading the unsigned driver.
  - The driver is signed, but the signature is invalid because the driver file has been altered.
  - The system disk device might have device errors when reading the file for the driver from bad disk sectors.
- If system audit policy is enabled, Code Integrity generates System Audit log events that correspond to the Operational warning events that indicate that signature verification of driver file failed. System audit policy is not enabled by default.
- If verbose logging for Code Integrity is enabled, Code Integrity logs analytic and debug events that provide information about successful verification checks that occur prior to loading kernel-mode driver files. Verbose logging for Code Integrity is not enabled by default.

You can use Event Viewer to view Code Integrity events, as described in [Viewing Code Integrity Events](#). For more information about these event log messages, see [Code Integrity Event Log Messages](#).

For more information about how to enable the system audit log and verbose logging, see [Enabling the System Event Audit Log](#).

# Viewing Code Integrity Events

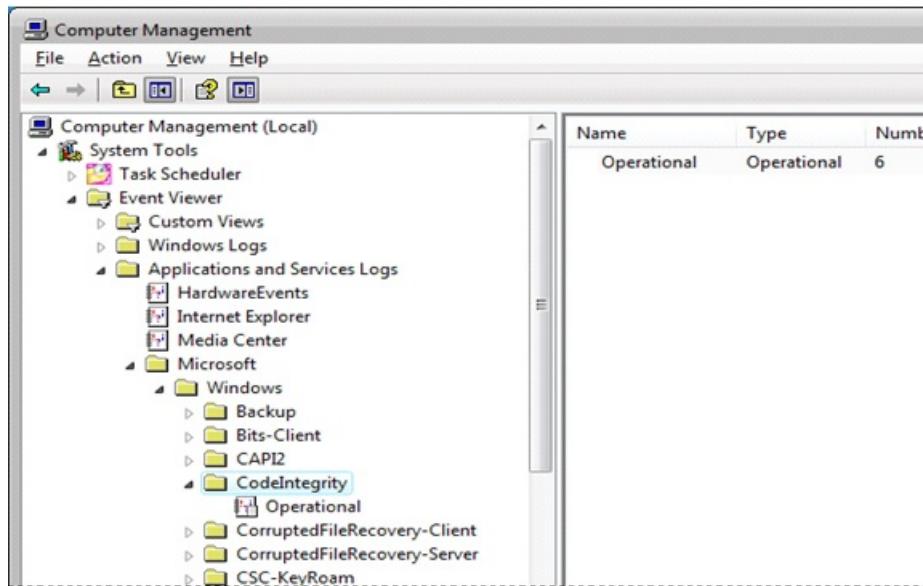
12/5/2018 • 2 minutes to read • [Edit Online](#)

You can use the Event Viewer to view Code Integrity events. You can access the Event Viewer in the Computer Management Microsoft Management Console (MMC) or by running the `Eventvwr.exe` command from a command line.

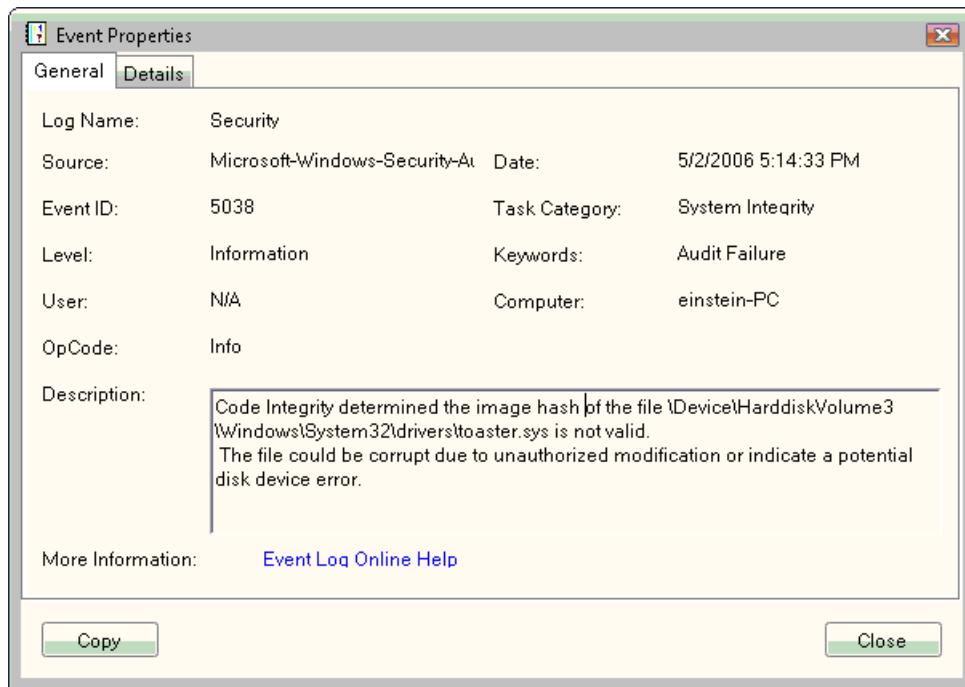
To view Code Integrity events in the Event Viewer, expand the following sequence of subfolders under the **Event Viewer** folder in the left pane of the Computer Management MMC or the Event Viewer window:

1. Applications and Services Logs
2. Microsoft
3. Windows
4. CodeIntegrity

The following screen shot shows the result of expanding the **CodeIntegrity** subfolder under the **Event Viewer** folder.



For more information about a particular Code Integrity log entry, right-click the entry and then select **Event Properties** on the pop-up menu. The following screen shot shows the details about a Code Integrity event.



This event indicates that the Toaster driver (toaster.sys) could not be loaded because it was unsigned (or the toaster.sys image that is trying to load is not the same one that was digitally-signed by the publisher).

For a list of all Code Integrity event log messages, see [Code Integrity Event Log Messages](#).

The System log events are viewable in the Event Viewer under the Windows Logs, System log view.

# Enabling the System Event Audit Log

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic includes the following information:

[How to Enable Security Audit Policy](#)

[How to Enable Verbose Logging of Code Integrity Diagnostic Events](#)

## How to Enable Security Audit Policy

To enable security audit policy to capture load failures in the audit logs, follow these steps:

1. Open an elevated Command Prompt window. To open an elevated Command Prompt window, create a desktop shortcut to *Cmd.exe*, select and hold (or right-click) the *Cmd.exe* shortcut, and select **Run as administrator**.
2. In the elevated Command Prompt window, run the following command:

```
Auditpol /set /Category:System /failure:enable
```

3. Restart the computer for the changes to take effect.

The following screen shot shows an how to use Auditpol to enable security auditing.

## How to Enable Verbose Logging of Code Integrity Diagnostic Events

To enable verbose logging, follow these steps:

1. Open an elevated Command Prompt window.
2. Run *Eventvwr.exe* on the command line.
3. Under the **Event Viewer** folder in the left pane of the Event Viewer, expand the following sequence of subfolders:
  - a. **Applications and Services Logs**
  - b. **Microsoft**
  - c. **Windows**
4. Expand the **Code Integrity** subfolder under the **Windows** folder to display its context menu.
5. Select **View**.
6. Select **Show Analytic and Debug Logs**. Event Viewer will then display a subtree that contains an **Operational** folder and a **Verbose** folder.

7. Select and hold (or right-click) **Verbose** and then select **Properties** from the pop-up context menu.
8. Select the **General** tab on the **Properties** dialog box, and then select the **Enable Logging** option near the middle of the property page. This will enable verbose logging.
9. Restart the computer for the changes to take effect.

# Code Integrity Event Log Messages

12/5/2018 • 2 minutes to read • [Edit Online](#)

The following are warning events that are logged to the Code Integrity operational log:

- Code Integrity is unable to verify the image integrity of the file <*file name*> because file hash could not be found on the system.
- Code Integrity detected an unsigned driver.

This event is related to Software Quality Monitoring (SQM).

The following are informational events that are logged to the Code Integrity verbose log:

- Code Integrity found a set of per-page image hashes for the file <*file name*> in a catalog <*catalog name*>.
- Code Integrity found a set of per-page image hashes for the file <*file name*> in the image embedded certificate.
- Code Integrity found a file hash for the file <*file name*> in a catalog <*catalog name*>.
- Code Integrity found a file hash for the file <*file name*> in the image embedded certificate.
- Code Integrity determined an unsigned kernel module <*file name*> is loaded into the system. Check with the publisher to see whether a signed version of the kernel module is available.
- Code Integrity is unable to verify the image integrity of the file <*file name*> because the set of per-page image hashes could not be found on the system.
- Code Integrity is unable to verify the image integrity of the file <*file name*> because the set of per-page image hashes could not be found on the system. The image is allowed to load because kernel mode debugger is attached.
- Code Integrity is unable to verify the image integrity of the file <*file name*> because a file hash could not be found on the system. The image is allowed to load because kernel mode debugger is attached.
- Code Integrity was unable to load the <*file name*> catalog.
- Code Integrity successfully loaded the <*file name*> catalog.

# SetupAPI Device Installation Log Entries

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, [SetupAPI logs](#) information about device installation in a plain-text log file that you can use to verify the installation of a device and to troubleshoot device installation problems. If a signing problem exists, SetupAPI will log information about the signing problem in the log file. The name of this log file is *SetupAPI.dev.log*, and it is located, by default, in the Windows INF file directory (%SystemRoot%\inf).

# Windows Driver Signing Tutorial

11/2/2020 • 2 minutes to read • [Edit Online](#)

This tutorial provides an overview and details the steps to sign driver binaries for Windows in one consolidated location. The following subtopics describe the process:

[Test Signing](#) [Release Signing](#) [Troubleshooting](#) [Driver Signing Installation](#)

## Overview

Starting with Windows Vista, x64-based versions of Windows required all software running in kernel mode, including drivers, to be digitally signed in order to be loaded.

Microsoft provides the following two ways to digitally sign drivers:

1. [Certify your driver with Microsoft](#) and Microsoft will provide a signature for it. When your driver package passes the certification tests, it can be signed by Windows Hardware Quality Labs (WHQL). If your driver package is signed by WHQL, it can be distributed through the Windows Update program or other Microsoft-supported distribution mechanisms.
2. Vendors or driver developers can obtain a software publishing certificate (SPC) from a Microsoft authorized Certificate Authority (CA) and use it to sign kernel mode and user mode binaries by themselves.

In the case of boot-start drivers during system start, drivers that are loaded by the system loader (Windows Vista and later versions of Windows), vendors must use a Software Publishers Certificate (SPC) to embed-sign their driver binary image file.

**Note** The mandatory kernel-mode code-signing policy applies to all kernel-mode software for x64-based systems that are running on Windows Vista and later versions of Windows. However, Microsoft encourages publishers to digitally sign all kernel-mode software, including device drivers (user-mode drivers included) for 32-bit systems as well. Windows Vista and later versions of Windows, verify kernel-mode signatures on 32-bit systems. Software to support protected media content must be digitally signed even if it is 32-bit.

User-mode drivers, like the Printer driver will install and work in an x64-based computer. A dialog will appear to the user during installation asking for approval to install the driver. Beginning in Windows 8 and later versions of Windows, installation will not proceed unless these driver packages are also signed.

The following resources describe Driver Signing in greater detail:

- The main [Driver Signing](#) topic
- The subtopic "How to Release Sign a Kernel Module" in the [Kernel-Mode Code Signing Walkthrough](#) describes what you should know about signing kernel-mode code. The information in the document also applies to signing user-mode drivers.
- The selfsign\_readme.htm file in the Windows 7 WDK is located in the WDK install directory, \WinDDK\7600.16385.1\src\general\build\driversigning. This directory also has a command file, selfsign\_example.cmd, which can be edited to run the driver signing commands. The selfsign\_readme.htm file in the Windows 8.1 WDK is located at C:\Program Files (x86)\Windows Kits\8.1\bin\selfsign, and provides links to additional driver signing information.

# Test Signing

12/1/2020 • 18 minutes to read • [Edit Online](#)

Starting with Windows Vista, x64-based versions of Windows required all software running in kernel mode, including drivers, to be digitally signed in order to be loaded. Initially you could use the F8 switch (on each boot, before Windows loads) to temporarily disable the load-time enforcement of requiring a valid signature in your driver. But this will become tedious after the first few uses. You can attach a kernel debugger to your test computer which will disable the same load-time enforcement checks after you use the correct BCDEdit commands. However, eventually it will become necessary to test-sign your driver during its development, and ultimately release-sign your driver before publishing it to users.

## Installing an Unsigned Driver during Development and Test

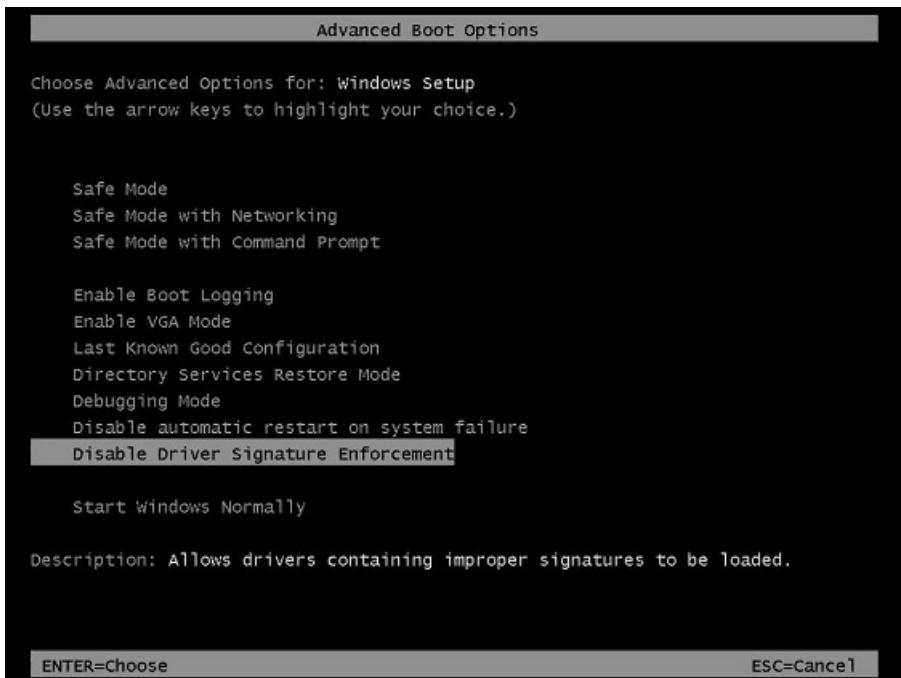
*Excerpt from [Installing an Unsigned Driver during Development and Test](#):*

By default, 64-bit versions of Windows Vista and later versions of Windows will load a kernel-mode driver only if the kernel can verify the driver signature. However, this default behavior can be disabled to during early driver development and for non-automated testing. Developers can use one of the following mechanisms to temporarily disable load-time enforcement of a valid driver signature. However, to fully automate testing of a driver that is installed by Plug and Play (PnP), the [catalog file](#) of the driver must be signed. Signing the driver is required because Windows Vista and later versions of Windows display a driver signing dialog box for unsigned drivers that require a system administrator to authorize the installation of the driver, potentially preventing any user without the necessary privileges from installing the driver and using the device. This PnP driver installation behavior cannot be disabled on Windows Vista and later versions of Windows.

## Use the F8 Advanced Boot Option

Windows Vista and later versions of Windows support the F8 Advanced Boot Option -- "Disable Driver Signature Enforcement" -- that disables load-time signature enforcement for a kernel-mode driver only for the current system session. This setting does not persist across system restarts.

The following boot option screen will appear during reboot providing the option to disable the driver signature enforcement. This provision will allow installation of an unsigned driver for test purpose.



### Attach a Kernel Debugger to Disable Signature Verification

Attaching an active kernel debugger to a development or test computer disables load-time signature enforcement for kernel-mode drivers. To use this debugging configuration, attach a debugging computer to a development or test computer, and enable kernel debugging on the development or test computer by running the following command:

```
bcdedit -debug on
```

To use BCDEdit, the user must be a member of the Administrators group on the system and run the command from an elevated command prompt. To open an elevated Command Prompt window, create a desktop shortcut to *Cmd.exe*, select and hold (or right-click) the shortcut, and select **Run as administrator**.

However, be aware that there are also situations in which a developer might have to attach a kernel debugger, yet also need to maintain load-time signature enforcement. See [Appendix 1: Enforcing Kernel-Mode Signature Verification in Kernel Debugging Mode](#) for how to accomplish this.

## Test Sign a Driver Package

Instead of using the above two methods to bypass driver signing enforcement requirements, the best approach is to test sign a driver package. Test signing and driver installation can be done on the development computer, but you may want to have two computers, one for development and signing, and the other for testing.

*Excerpt from [How to Test-Sign a Driver Package](#):*

### **Signing computer**

This is the computer that is used to test-sign a driver package for Windows Vista and later versions of Windows. This computer must be running Windows XP SP2 or later versions of Windows. In order to use the [driver signing tools](#), this computer must have the Windows Vista and later versions of the Windows Driver Kit (WDK) installed. This can also be the development computer.

### **Test computer**

This is the computer that is used to install and test the test-signed driver package. This computer must be running Windows Vista or later versions of Windows.

## Test Signing Procedure

Driver packages will contain the driver binary, the INF file, the CAT file and any other necessary files. A driver package may contain sub directories like x86, AMD64, IA64, if the driver is built for more than one target processor type. Perform these steps using your development/signing computer.

The following procedure describes the steps to test sign a driver package:

1. Build the driver for the target. If you are building a driver for Windows 8.0 or Windows 8.1, then use Visual Studio 2012 or Visual Studio 2013 with the corresponding WDK, installed e.g., Windows 8.0 or 8.1 WDK respectively.

All the command tools described below should be used from the corresponding tool/build command window Visual Studio 2012 or Visual Studio 2013.

**NOTE**

The command tools for Visual Studio are located in the install directory, C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\Tools\Shortcuts

Any of the five shortcuts for command prompt will have, makecert.exe, inf2cat.exe, signtool.exe, certmgr.exe, etc., commands.

You may choose the most general, "Developer Command Prompt for VS2013". The shortcuts can be pinned down to the Task Bar for easy access.

**NOTE**

Please note that with Visual Studio, instead of the command tool approach of driver signing, you can also use the Visual Studio 2013 development environment (also known as the IDE) to sign a driver package. Please refer to [Appendix 2: Signing Drivers with Visual Studio](#) for more information.

2. Create a driver package folder and copy the driver files, maintaining any sub directories needed, for example C:\DriverTestPackage.
3. Create an inf file for the driver package. Test the inf file using the [InfVerif](#) tool from WDK on the inf file so that no error is reported.
4. *Excerpt from [Creating Test Certificates](#):*

The following command-line example uses MakeCert to complete the following tasks:

- Create a self-signed test certificate named *Contoso.com(Test)*. This certificate uses the same name for the subject name and the certificate authority (CA).
- Put a copy of the certificate in an output file that is named *ContosoTest.cer*.
- Put a copy of the certificate in a certificate store that is named *PrivateCertStore*. Putting the test certificate in *PrivateCertStore* keeps it separate from other certificates that may be on the system.

Use the following MakeCert command to create the *Contoso.com(Test)* certificate:

```
makecert -r -pe -ss PrivateCertStore -n CN=Contoso.com(Test) ContosoTest.cer
```

Where:

- The **-r** option creates a self-signed certificate with the same issuer and subject name.
- The **-pe** option specifies that the private key that is associated with the certificate can be exported.

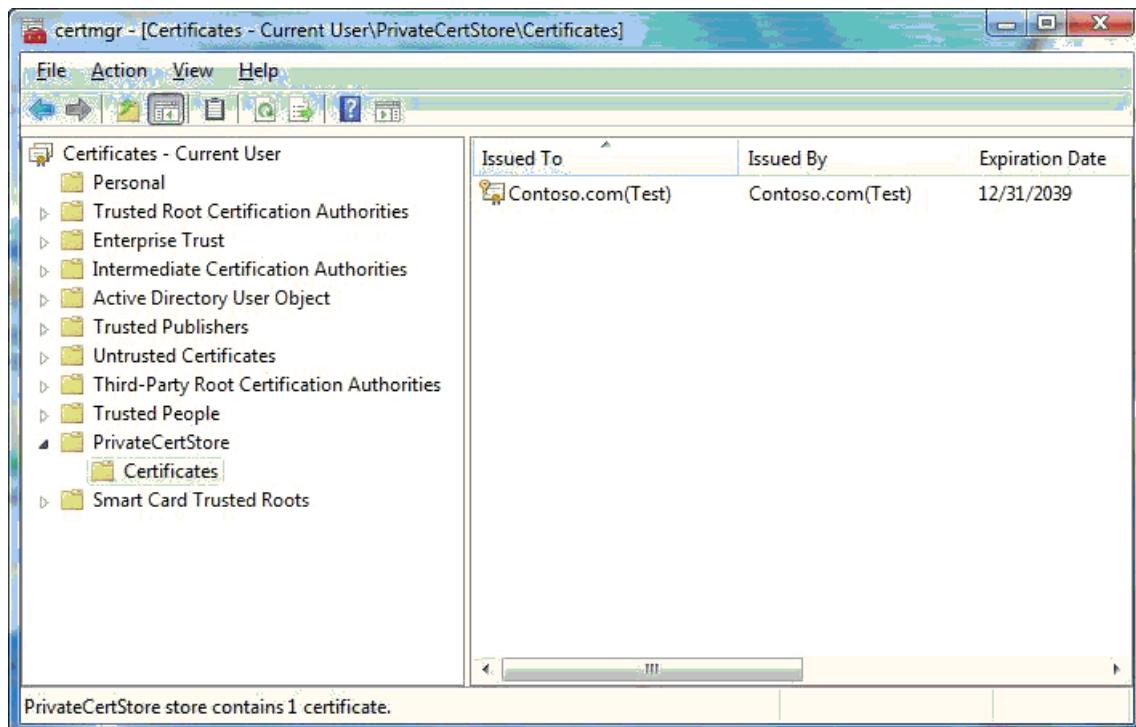
- The **-ss** option specifies the name of the certificate store that contains the test certificate (*PrivateCertStore*).
- The **-n CN=** option specifies the name of the certificate, Contoso.com(Test). This name is used with the **SignTool** tool to identify the certificate.
- *ContosoTest.cer* is the file name that contains a copy of the test certificate, Contoso.com(Test). The certificate file is used to add the certificate to the Trusted Root Certification Authorities certificate store and the Trusted Publishers certificate store.

*Excerpt from Viewing Test Certificates:*

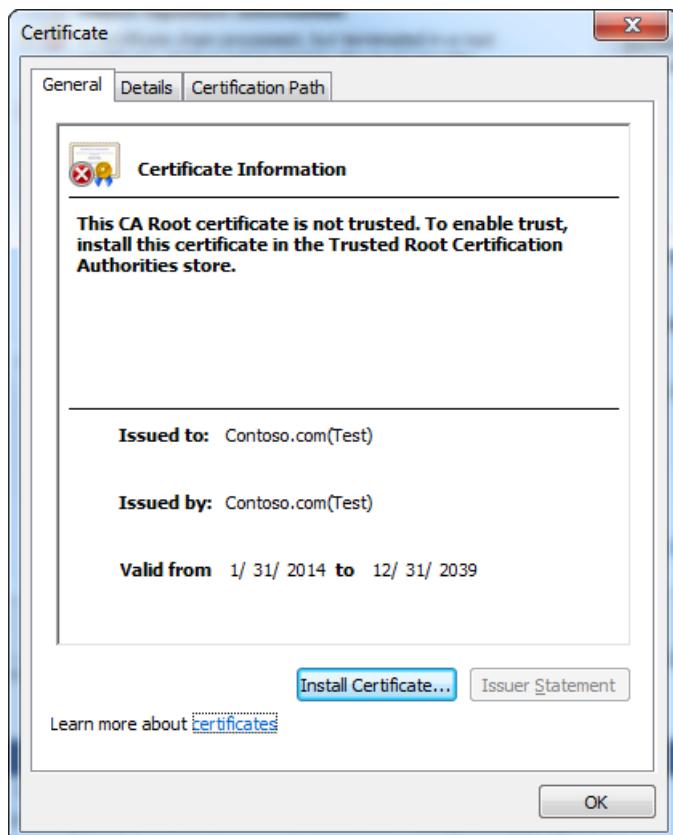
After the certificate is created and a copy is put in the certificate store, the Microsoft Management Console (MMC) Certificates snap-in can be used to view it. Do the following to view a certificate through the MMC Certificates snap-in:

- a. To start the Certificates snap-in, run Certmgr.msc.
- b. In the left pane of the Certificates snap-in, expand the PrivateCertStore certificate store folder and double-click Certificates.

The following screen shot shows the Certificates snap-in view of the **PrivateCertStore** certificate store folder.



To view the details about the Contoso.com(Test) certificate, double-click the certificate in the right pane. The following screen shot shows the details about the certificate.



Notice that the Certificate dialog box states: "This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store." This is the expected behavior. The certificate cannot be verified because Windows does not trust the issuing authority, "Contoso.com(Test)" by default.

5. Create a catalog file (.cat extension). Use the inf2cat tool as shown below to create the catalog file. Please note that no space is allowed for the switches, /driver:<no space><full path>, /os: <no space><os1 name>;<no space><os2 name>.

```
inf2cat /v /driver:C:\DriverTestPackage /os:7_64,7_x86 ,XP_X86
```

This creates a catalog file with the name given in the driver's .inf file. Additional comma separated OSes can be added selectively or all as shown below with no spaces.

```
/os:2000,XP_X86,XP_X64,Server2003_X64,Vista_X64,Vista_X86,7_x86,7_64,Server2008_x86,Server2008_x64,Seve  
r2008_IA64,Server2008R2_x86,Server2008R2_x64,Server2008R2_IA64,8_x86,8_x64, 8_ARM, Server8_x64
```

The updated inf2cat from the new 8.1 WDK has /os option values of 6\_3\_X86, 6\_3\_X64, 6\_3\_ARM and SERVER\_6\_3\_X64.

Example of INF file for the Version section.

```
[Version]
Signature="$WINDOWS NT$"
Class=TOASTER
ClassGuid={B85B7C50-6A01-11d2-B841-00C04FAD5171}
Provider=%ToastRUs%
DriverVer=09/21/2006,6.0.5736.1
CatalogFile.NTx86 = tostx86.cat
CatalogFile.NTIA64 = totstia64.cat
CatalogFile.NTAMD64 = tstammd64.cat
```

The /driver (or /drv) option specifies the directory which contains one or more INF files. Within this directory, catalog files are created for those INF files that contain one or more CatalogFile directives. The catalog file name is not restricted to 8.3 name.

Inf2Cat creates the catalog file *tstamd64.cat* if the command-line argument /os:7\_X64 is used. Similarly, the tool creates the catalog file *toastx86.cat* if the /os:XP\_X86, option is used, similarly for Server2008R2\_IA64. In case, only one catalog file is desired, then only one entry in the INF file as shown below will suffice.

```
CatalogFile.NT = toaster.cat
```

Or,

```
CatalogFile = toaster.cat
```

If the date in the INF file is not greater than the OS release date, then the following error will be reported by the inf2cat tool if the /os parameter was for Windows 7 and date set in the INF file was an earlier date.

```
Signability test failed.  
Errors:  
22.9.7: DriverVer set to incorrect date (must be postdated to 4/21/2009 for newest OS) in \toaster.inf
```

The inf2cat tool is very strict on checking each folder and sub-folder about the presence of every file which has an entry in the INF file. There will be meaningful error messages on such missing entries.

The cat file can be opened from explorer by double-clicking or right-clicking the file and selecting Open. The Security tab will show some entries with GUID values. Selecting a GUID value will display details including the driver files of the driver package and the OSes added as shown below:

```
OSAttr 2:5.1,6.1
```

The number 5.1 is the version number for XP OS and 6.1 for Windows 7.0 OS.

It is advisable that the cat file is checked to verify the inclusion of the driver files and the selected OSes. At any time if any driver file is added or removed, the INF file has been modified, the cat file must be recreated and signed again. Any omission here will cause installation errors which are reported on the setup log file (setupapi.dev.log for Vista and above or setupapi.log file for XP).

## 6. Excerpt from Test-Signing a Driver Package's Catalog File:

The following command line shows how to run SignTool to do the following:

- Test-sign the *tstamd64.cat* catalog file of the *ToastPkg* sample driver package. For more information about how this catalog file was created, see [Creating a Catalog File for Test-Signing a Driver Package](#).
- Use the Contoso.com(Test) certificate from the PrivateCertStore for the test signature. For more information about how this certificate was created, see [Creating Test Certificates](#).
- Timestamps the digital signature through a time stamp authority (TSA).

To test-sign the *tstamd64.cat* catalog file, run the following command line:

```
signtool sign /v /s PrivateCertStore /n Contoso.com(Test) /t http://timestamp.digicert.com tstamd64.cat
```

Where:

- The **sign** command configures SignTool to sign the specified catalog file, *tstamd64.cat*.
- The **/v** option enables verbose operations, in which SignTool displays successful execution and warning messages.
- The **/s** option specifies the name of the certificate store (*PrivateCertStore*) that contains the test certificate.
- The **/n** option specifies the name of the certificate (*Contoso.com(Test)*) that is installed in the specified certificate store.
- The **/t** option specifies URL of the TSA (`http://timestamp.digicert.com`) which will time stamp the digital signature.

**IMPORTANT**

Including a time stamp provides the necessary information for key revocation in case the signer's code signing private key is compromised.

- *tstamd64.cat* specifies the name of the catalog file, which will be digitally-signed.

*tstamd64.cat* specifies the name of the catalog file, which will be digitally-signed. You can open the cat file as described before

7. *Modified excerpt from Test-Signing a Driver through an Embedded Signature:*

- In 64-bit versions of Windows Vista and later versions of Windows, the kernel-mode code signing requirements state that a *boot-start driver* must have an embedded signature. This is required regardless of whether the driver's driver package has a digitally-signed catalog file.

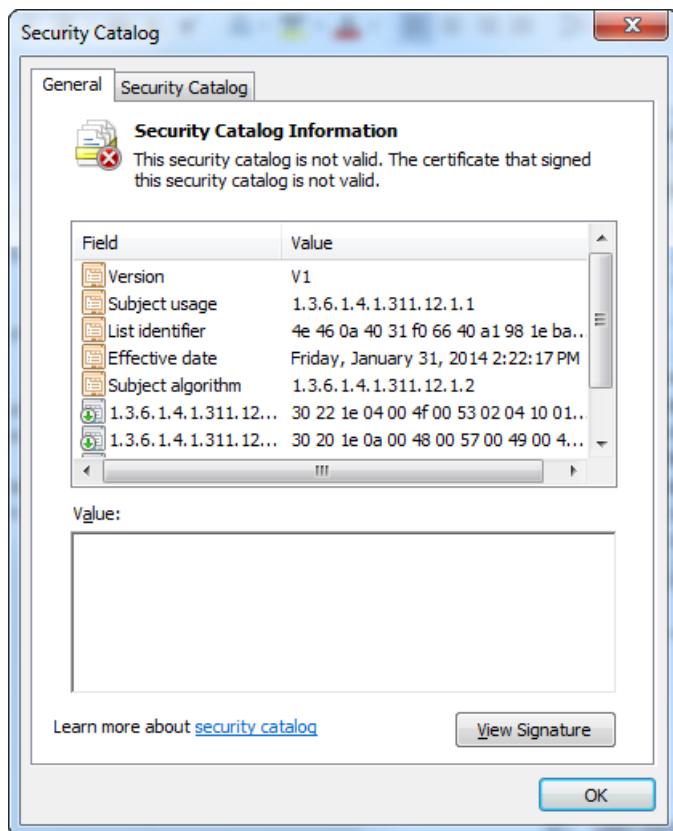
Below is the command to embed sign a kernel mode driver binary file.

```
signtool sign /v /s PrivateCertStore /n Contoso.com(Test) /t http://timestamp.digicert.com
amd64\toaster.sys
```

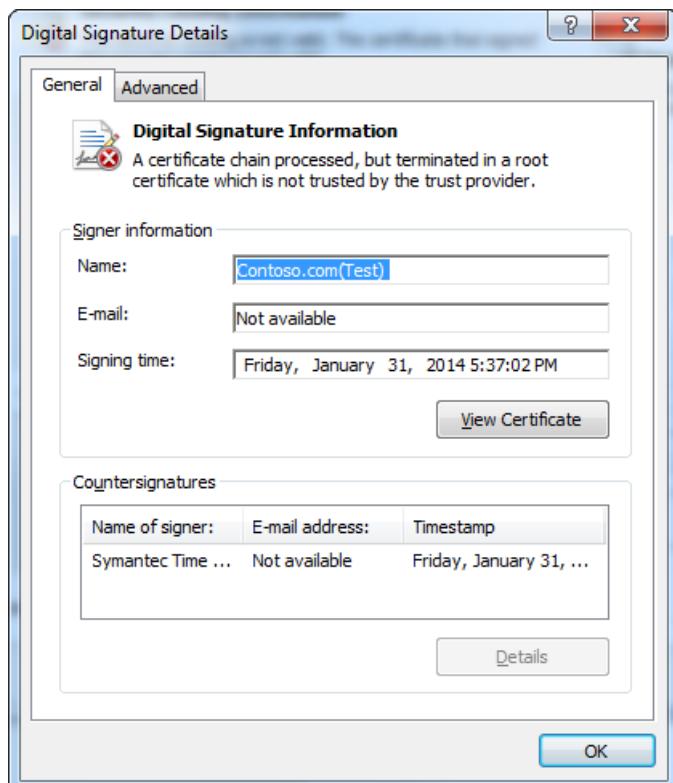
*amd64\toaster.sys* specifies the name of the kernel-mode binary file which will be embed-signed.

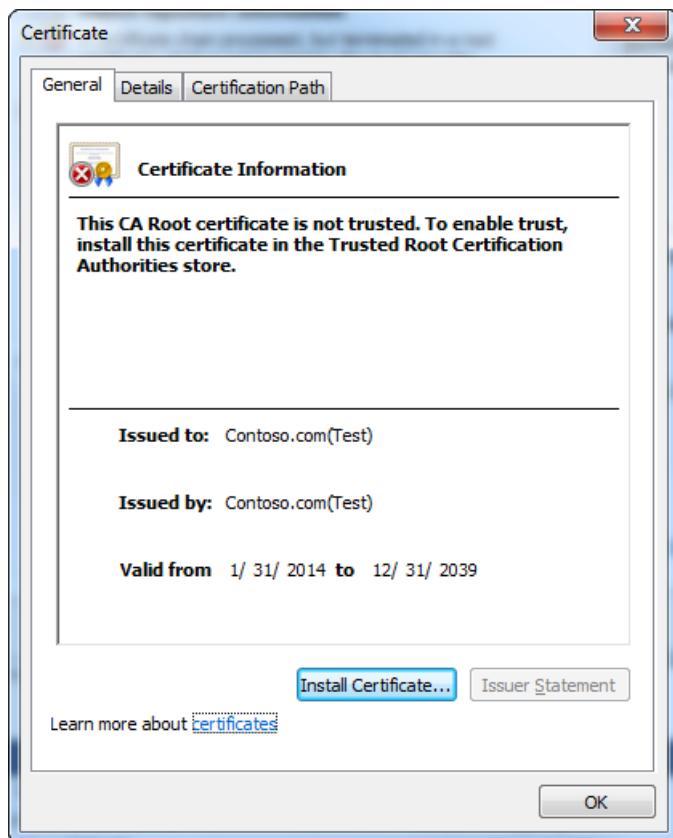
Within the WDK 7.1 installation directory, the toaster sample is located in the *src\general\toaster\toastpkg\toastcd\* directory. The Windows 8 or 8.1 WDK samples are to be downloaded from the Microsoft download site. The samples do not come with the Windows 8 or 8.1 Windows Driver Kit.

The catalog file when opened by double clicking the file in Windows Explorer, you will see the following screen shot. Note that "View Signature" is now highlighted.



If you select "View Signature", you will see the screen shot below providing the next viewing option from "View Certificate", which then will give the option of "Install Certificate" from the dialog itself. Below, we are providing the preferred command line option of installing the certificate using the certmgr.exe tool.





The driver can now be tested either on the signing computer or the test computer. If you are using the test computer, copy the driver package to the machine keeping the file structure intact. The tool certmgr.exe also has to be copied to the test computer. When using a test computer, copy the test-signed Toastpkg driver package to the c:\toaster temporary folder.

The following procedure describes the steps to use on either machine to test the driver:

1. In an elevated command window run the following command:

```
bcdedit /set testsigning on
```

Reboot the computer.

2. *Selected excerpts from Using CertMgr to Install Test Certificates on a Test Computer:*

Copy the certificate (.cer) file, which was used to [test-sign](#) drivers, to the test computer. You can copy the certificate file to any directory on the test computer.

The following CertMgr command adds the certificate in the certificate file *CertificateFileName.cer* to the Trusted Root Certification Authorities certificate store on the test computer:

```
CertMgr.exe /add CertificateFileName.cer /s /r localMachine root
```

The following CertMgr command adds the certificate in the certificate file *CertificateFileName.cer* to the Trusted Publishers certificate store on the test computer:

```
CertMgr.exe /add CertificateFileName.cer /s /r localMachine trustedpublisher
```

Where (*excerpts from CertMgr*):

/add CertificateName

Adds the certificate in the specified certificate file to the certificate store.

/s

Specifies that the certificate store is a system store.

/r RegistryLocation

Specifies that the registry location of the system store is under HKEY\_LOCAL\_MACHINE.

CertificateStore

Specifies the certificate store, trustedpublisher, similarly for "localMachine root".

Reboot the computer. You can now run Certmgr.msc and verify that the ContosoTest.cer is visible in the above two locations. If it is not visible, then another way to install the certificate is to open the certificate and install it on the above two nodes and verify again.

3. Verify signing of the cat file and the sys file. Open an elevated command window, and assuming the signtool.exe is available in the computer, go to the driver package directory where the cat, inf and the sys file is located. Execute the following commands at the appropriate directory

From [Verifying the SPC Signature of a Catalog File](#):

```
signtool verify /v /kp /c tstamd64.cat toaster.inf
```

To check for embed sign, execute the following command.

From [Verifying the Signature of a Release-Signed Driver File](#):

```
signtool verify /v /kp toaster.sys
```

The two commands above will generate one error as it is test signed and the certificate was not a trusted certificate.

```
SignTool Error: A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.
```

The above two verification commands will be very useful in release signing which will be discussed later.

The driver is now ready to be installed and tested in the test computer. It is always advisable that the following registry key is set correctly to gather verbose logs in setupapi.dev.log file (for Windows Vista and later operating systems) during the installation process.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Setup\Loglevel=0x4800FFFF
```

In %SystemRoot%\inf file, rename the setupapi.dev.log file before installing the driver. After install, a new log setupapi.dev.log file will be created which will contain valuable information encountered during installation.

Once the driver is successfully installed, it can be tested on the development computer or on the test computer.

## Installing, Uninstalling and Loading the Test-Signed Driver Package

After the system has rebooted in Step 2, the test-signed driver package can be installed and loaded. There are four ways to install a driver package:

1. By using the Dpinst (dpinst.exe) tool, which is a WDK command line tool for installing drivers and is redistributable.
2. By using the Devcon (devcon.exe) tool, which is a WDK command line tool for installing drivers, but not redistributable. The sample code of Devcon tool is provided in the WDK. To redistribute, you can implement your own Devcon tool from the sample code and can redistribute your version of the tool.
3. By using the OS provided Pnutil (pnutil.exe) tool.
4. By using the Windows Add Hardware Wizard.

Dpinst and Pnutil pre-installs the driver package, whereas with Devcon and Windows Add Hardware Wizard, the driver as well as the device can be installed. Pre-installing a driver helps the OS find the driver when a device is connected to the computer.

#### **To install (and uninstall) the driver package by using DPInst**

1. Open an elevated command window and set the default directory to c:\toaster.
2. Dpinst.exe is provided in the WDK redist directory the x86 version, the amd64 version and the ia64 version. Copy the relevant version to the c:\toaster directory and run the following command.

```
dpinst.exe /PATH c:\toaster
```

The above command will install all the drivers corresponding to all the inf files. You can also use ":" without the quotes from the current directory. "dpinst.exe /?" shows all the switches for this tool.

The /U switch on the driver inf file will remove the driver package from the DriverStore's FileRepository (%SystemRoot%\System32\ DriverStore\FileRepository) directory provided the device associated with the driver has been removed. With Dpinst tool a driver can be removed just by referring to the inf file of the driver.

```
dpinst.exe /U toaster.inf
```

#### **To install the driver package by using DevCon**

1. Open an elevated command window and set the default directory to c:\toaster.
2. Devcon.exe is provided in the WDK tool directory the x86 version, the amd64 version and the ia64 version. Copy the relevant version to the c:\toaster directory and run the following command. This command will install the driver as well as the device.

```
devcon.exe install <inf> <hwid>
```

It is advisable to use quotes around <hwid>. For the toaster sample, it will be:

```
devcon.exe install c:\toaster\toaster.inf "{b85b7c50-6a01-11d2-b841-00c04fad5171}\MsToaster"
```

A device can be removed using the Devcon tool using the "remove" switch. "devcon.exe /?" shows all the switches for this tool. To get specific information, on using a switch "help" should be added as shown below for the "remove" switch.

```
devcon.exe help remove
```

The above commands provides the following information.

Removes devices with the specified hardware or instance ID. Valid only on the local computer. (To reboot

when necessary, include -r.)

```
devcon [-r] remove <id> [<id>...]
devcon [-r] remove =<class> [<id>...]
<class>      Specifies a device setup class.
Examples of <id>:
*                  - All devices
ISAPNP\PNP0501 - Hardware ID
*PNP*              - Hardware ID with wildcards  (* matches anything)
@ISAPNP\*\*        - Instance ID with wildcards (@ prefixes instance ID)
'*PNP0501         - Hardware ID with apostrophe (' prefixes literal match - matches exactly as typed,
including the asterisk.)
```

After a device has been removed, to remove the driver, two commands are necessary. Use the first command with "dp\_enum" switch to find the driver inf file name corresponding to the driver package installed in the computer.

```
devcon dp_enum
```

This command will show the list of all oemNnn.inf files corresponding to a driver package, where Nnn is a decimal number with the Class information and the Provide information as shown below.

```
oem39.inf
Provider: Intel
Class: Network adapters
oem4.inf
Provider: Dell
Class: ControlVault Device
```

To remove the corresponding driver package from the DriverStore, use the next command shown below for the Intel "Network Adapters" driver:

```
devcon.exe dp_delete oem39.inf
```

### To install the driver package by using PnpUtil

1. Open an elevated command window and set the default directory to c:\toaster.
2. Run the following command which will show all the available switches. Use of the switches is self-explanatory, no need to show any examples.

```
C:\Windows\System32\pnputil.exe /?

Microsoft PnP Utility
Usage:
-----
pnputil.exe [-f | -i] [ -? | -a | -d | -e ] <INF name>
Examples:
pnputil.exe -a a:\usbcam\USBCAM.INF      -> Add package specified by USBCAM.INF
pnputil.exe -a c:\drivers\*.inf            -> Add all packages in c:\drivers\
pnputil.exe -i -a a:\usbcam\USBCAM.INF    -> Add and install driver package
pnputil.exe -e                                -> Enumerate all 3rd party packages
pnputil.exe -d oem0.inf                      -> Delete package oem0.inf
pnputil.exe -f -d oem0.inf                   -> Force delete package oem0.inf
pnputil.exe -?                                -> This usage screen
```

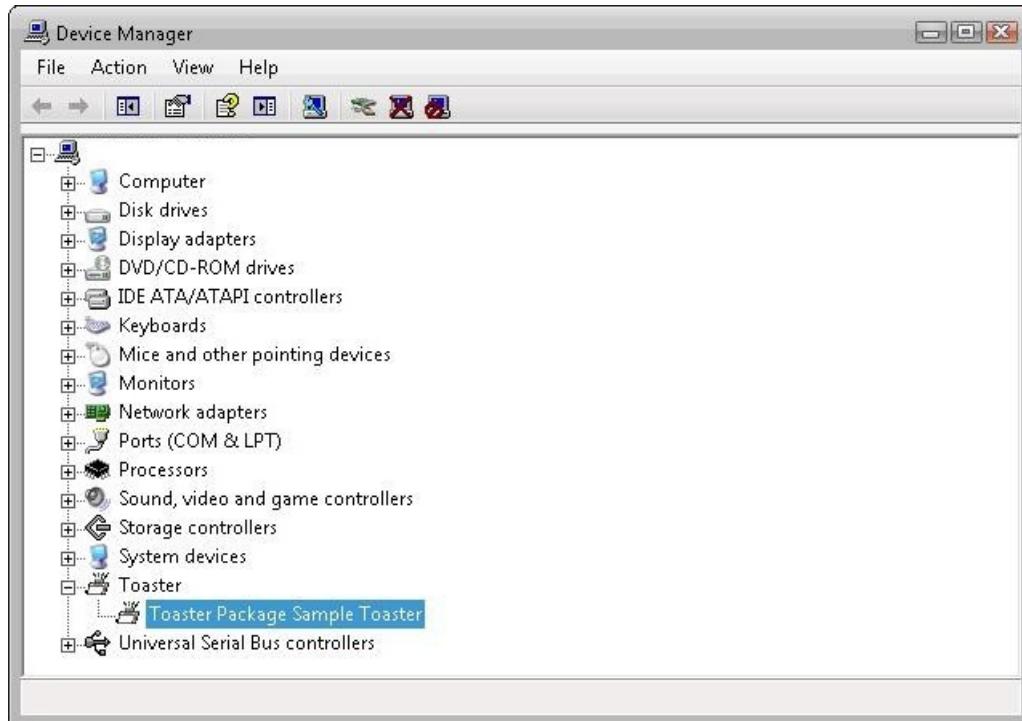
### To install the driver package by using the Add Hardware Wizard

1. Open an elevated command window
2. Run hdwwiz.cpl to start the Add Hardware Wizard, and select Next to go to the second page
3. Select Advanced Option and select Next
4. Select Show all devices from the list box and select Next
5. Select the Have Disk option
6. Enter the path to the folder that contains the C:\toaster driver package
7. Select the inf file and select Open
8. Select OK
9. Select Next in the next two pages, and then select Finish to complete the installation

### **Verify that the Test-Signed Driver Is Operating Correctly**

To verify that Toastpkg is operating correctly:

1. Start Device Manager
2. Select Toaster from the list of devices. For an example, see screen shot below.



3. To open the driver's Properties dialog box, double-click Toaster Package Sample Toaster and select Properties
4. To confirm that Toaster is working properly, on the General tab, check the Device status box

Device Manager can be used to uninstall the device and the driver from the Properties dialog box.

### **How to Troubleshoot Test-Signed Drivers**

See [Troubleshooting Driver Signing Installation](#) if you encounter any difficulties with these procedures.

# Release Signing

11/2/2020 • 12 minutes to read • [Edit Online](#)

After completing test signing and verifying that the driver is ready for release, the driver package has to be release signed. There are two ways of release signing a driver package.

Release-signing identifies the publisher of a kernel-mode or user-mode binaries (for example, .sys or .dll files) that loads into Windows Vista and later versions of Windows.

Kernel-mode binaries are release-signed through either:

1. Windows Hardware Quality Lab (WHQL also known as Winqual) to release sign a driver package. A WHQL Release Signature is obtained through the Windows Certification Program. The link below describes the five steps from start to finish on Windows Certification Program. See [Windows hardware certification: start here](#) for more details about this option. Any questions on the steps in the link above should be directed to [sysdev@microsoft.com](mailto:sysdev@microsoft.com) alias.
2. Instead of using the WHQL program, a driver package can be release signed by driver developers and vendors. This program has started from the Vista OS release. A release signature is created through a Software Publisher Certificate (SPC). The SPC is obtained from a third-party certificate authority (CA) that is authorized by Microsoft to issue such certificates. Signatures generated with this type of SPC also comply with the PnP driver signing requirements for 64-bit and 32-bit versions of Windows Vista and later versions of Windows

The steps needed to release sign a driver package for method 2 are described next.

## Obtain a Software Publisher Certificate (SPC)

Release signing requires a code-signing certificate, also referred to as a Software Publisher Certificate (SPC) from a commercial CA.

The [Cross-Certificates for Kernel Mode Code Signing](#) topic provides the list of commercial third-party certificate authorities (CA) authorized by Microsoft. The CA vendors listed must be used to provide a Software Publisher Certificate (SPC) to release sign the driver package.

Follow the CA's instructions for how to acquire the SPC and install the private key on the signing computer. Please note that the SPC is a proprietary instrument of the driver vendor who requested it for signing their driver package. The SPC, the private key, and the password must not be shared with anyone outside the requesting vendor's organization.

## Cross-Certificates

*Excerpt from [Software Publisher Certificate](#):*

In addition to obtaining an SPC, you must obtain a cross-certificate that is issued by Microsoft. The cross certificate is used to verify that the CA that issued an SPC is a trusted root authority. A cross-certificate is an X.509 certificate issued by a CA that signs the public key for the root certificate of another CA. Cross-certificates allow the system to have a single trusted Microsoft root authority, but also provide the flexibility to extend the chain of trust to commercial CAs that issue SPCs.

Publishers do not have to distribute a cross-certificate with a [driver package](#). The cross-certificate is included with the digital signature for a driver package's [catalog file](#) or the signature that is embedded in a driver file. Users who install the driver package do not have to perform any additional configuration steps caused by the use of cross-certificates.

*Selected excerpts from [Cross-Certificates for Kernel Mode Code Signing](#):*

A cross-certificate is a digital certificate issued by one Certificate Authority (CA) that is used to sign the public key for the root certificate of another Certificate Authority. Cross-certificates provide a means to create a chain of trust from a single, trusted, root CA to multiple other CAs.

In Windows, cross-certificates:

- Allow the operating system kernel to have a single trusted Microsoft root authority.
- Extend the chain of trust to multiple commercial CAs that issue Software Publisher Certificates (SPCs), which are used for code-signing software for distribution, installation, and loading on Windows

The cross-certificates that are provided here are used with the Windows Driver Kit (WDK) code-signing tools for properly signing kernel-mode software. Digitally signing kernel-mode software is similar to code-signing any software that is published for Windows. Cross-certificates are added to the digital signature by the developer or software publisher when signing the kernel-mode software. The cross-certificate itself is added by the code-signing tools to the digital signature of the binary file or catalog.

### Selecting the Correct Cross-certificate

Microsoft provides a specific cross-certificate for each CA that issues SPCs for code-signing kernel-mode code. For more information about cross-certificates, see [Cross-Certificates for Kernel Mode Code Signing](#). This topic lists the names of the Microsoft authorized CA vendors and the correct cross-certificate for the root authority that issued your SPC. Locate the correct cross-certificate for the SPC issued by your CA vendor and download the cross-certificate to the signing computer you will be using for release signing, and store it in your driver directory. It is advisable to provide the absolute path to this Certificate when using it in any signing command.

## Installing SPC Information in the Personal Certificate Store

*Further excerpts from [Software Publisher Certificate](#):*

In order to use an SPC to sign a driver in a manner that complies with the [kernel-mode code signing policy](#), the certificate information must first be contained in a Personal Information Exchange (.pfx) file. The information that is contained in the .pfx file must then be added to the Personal certificate store of the local computer that signs a driver.

A CA might issue a .pfx file that contains the necessary certificate information. If so, you can add the .pfx file to the Personal certificate store by following the instructions described in [Installing a .pfx File in the Personal Certificate Store](#).

However, a CA might issue the following pairs of files:

- A .pvk file that contains the private key information.
- An .spc or .cer file that contains the public key information.

In this case, the pair of files (a .pvk and an .spc, or a .pvk and a .cer) must be converted to a .pfx file in order to add the certificate information to the Personal certificate store.

To create a .pfx file from the pair of files issued by the CA, follow these instructions:

- To convert a .pvk file and an .spc file to a .pfx file, use the following [Pvk2Pfx](#) command at a command prompt:

```
Pvk2Pfx -pvk mypvkfile.pvk -pi mypvkpassword -spc myspcfile.spc -pfx mypfxfile.pfx -po pfxpassword -f
```

- To convert a .pvk file and a .cer file, to a .pfx file, use the following Pvk2Pfx command at a command prompt:

```
Pvk2Pfx -pvk mypvkfile.pvk -pi mypvkpassword -spc mycerfile.cer -pfx mypfxfile.pfx -po pfxpassword -f
```

The following describes the parameters that are used in the [Pvk2Pfx](#) command:

- The **-pvk** *mypvkfile.pvk* parameter specifies a *.pvk* file.
- The **-pi** *mypvkpassword* option specifies the password for the *.pvk* file.
- The **-spc** *myspcfile.spc* parameter specifies an *.spc* file or the **-spc** *mycerfile.cer* parameter specifies a *.cer* file.
- The **-pfx** *mypfxfile.pfx* option specifies the name of a *.pfx* file.
- The **-po** *pfxpassword* option specifies a password for the *.pfx* file.
- The **-f** option configures Pvk2Pfx to replace a existing *.pfx* file if one exists.

[!IMPORTANT] You should protect your pvk and pfx files with strong passwords.

## Installing a .pfx File in the Personal Certificate Store

For signing kernel-mode drivers, the certificates and key stored in the *.pfx* file must be imported into the local Personal certificate store. Signtool does not support using *.pfx* files for signing kernel-mode drivers. The restriction is due to a conflict in adding cross-certificates in the signature while using a certificate from a *.pfx* file

*Final excerpts from Software Publisher Certificate:*

After obtaining a *.pfx* file from a CA, or creating a *.pfx* file from a *.pvk* and either an *.spc* or a *.cer* file, add the information in the *.pfx* file to the Personal certificate store of the local computer that signs the driver. You can use the Certificate Import Wizard to import the information in the *.pfx* file to the Personal certificate store, as follows:

1. Locate the *.pfx* file in Windows Explorer and double-click the file to open the Certificate Import Wizard.
2. Follow the procedure in the Certificate Import Wizard to import the code-signing certificate into the Personal certificate store.

*Excerpt from Importing an SPC into a Certificate Store:*

Starting with Windows Vista, an alternative way to import the *.pfx* file into the local Personal certificate store is with the [CertUtil](#) command-line utility. The following command-line example uses CertUtil to import the *abc.pfx* file into the Personal certificate store:

```
certutil -user -p pfxpassword -importPFX abc.pfx
```

Where:

- The **-user** option specifies "Current User" Personal store.
- The **-p** option specifies the password for the *.pfx* file (*pfxpassword*).
- The **-importPFX** option specifies name of the *.pfx* file (*abc.pfx*).

## View SPC Properties

Use the MMC Certificates snap-in (certmgr.msc) to view the certificates in the Personal certificate store.

1. Launch the Certificates snap-in, certmgr.msc.
2. In the snap-in's left pane, select the Personal certificate store folder.

3. Click the Certificates folder and double-click the certificate that is to be used for release signing.
4. On the Details tab of the Certificate dialog box, select Subject from the list of fields to highlight the certificate's subject name. This is the name that is used with Signtool's /n argument in the examples in this section.

## Signing

Based on [Release-Signing a Driver Package's Catalog File](#):

Run the following commands to sign the cat file which signs the driver package. The /n command should use the quoted name of the certificate which you will see under Subject in step 4 above, as CN= MyCompany Inc.

```
signtool sign /v /ac MSCV-VSClass3.cer /s My /n "MyCompany Inc." /t http://timestamp.digicert.com toaster.cat
```

/ac FileName

Specifies a file that contains an additional certificate to add to the signature block. This is the cross signing certificate, MSCV-VSClass3.cer, obtained from Microsoft cross certificate download link. Use a full path name if the cross-certificate is not in the current directory. Though not required, it is advisable to add cross certificate while signing the cat file.

/s StoreName

Specifies the store to open when searching for the certificate. If this option is not specified, the My store is opened, which is the Personal certificate Store.

/n SubjectName

Specifies the name of the subject of the signing certificate. This value can be a substring of the entire subject name.

/t URL

Specifies the URL of the time stamp server. If this option is not present, then the signed file will not be time stamped. **With time stamping, the signed driver package remains valid indefinitely until the SPC signing certificate gets revoked for other reasons.**

You must follow every signing steps correctly as described above, otherwise you will not be able to sign the driver. You may get errors shown below.

```
SignTool Error: No certificates were found that met all the given criteria
```

## Embed Signing

Based on [Release-Signing a Driver through an Embedded Signature](#):

Starting with Windows Vista, the kernel-mode code signing policy controls whether a kernel-mode driver will be loaded. 64-bit versions of Windows starting with Windows Vista has stricter requirement compared to 32-bit versions of Windows.

A kernel-mode boot-start driver must have an embedded Software Publisher Certificate (SPC) signature. This applies to any type of PnP or non-PnP kernel-mode boot-start driver. Also applies to the 32-bit versions of Windows. A PnP kernel-mode driver that is not a boot-start driver must have either an embedded SPC signature, a catalog file with a WHQL release signature, or a catalog file with an SPC signature.

Please refer to [Kernel-Mode Code Signing Requirements](#) for additional information.

Command for embed signing the toaster.sys file.

```
signtool sign /v /ac MSCV-VSClass3.cer /s my /n "MyCompany Inc." /t http://timestamp.digicert.com toaster.sys
```

After signing is completed, run the command below to verify the signed driver.

```
signtool verify /kp /v /c tstamd64.cat toastpkg.inf
```

Command output:

```
Verifying: toaster.inf
File is signed in catalog: toaster.cat
Hash of file (sha1): 580C2A24C3A9E12817E18ADF1C4FE9CF31B01EA3
```

#### **Signing Certificate Chain:**

```
Issued to: VeriSign Class 3 Public Primary Certification Authority - G5
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Wed Jul 16 15:59:59 2036
SHA1 hash: 4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5
```

```
Issued to: VeriSign Class 3 Code Signing 2010 CA
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Fri Feb 07 15:59:59 2020
SHA1 hash: 495847A93187CFB8C71F840CB7B41497AD95C64F
```

```
Issued to: Contoso, Inc
Issued by: VeriSign Class 3 Code Signing 2010 CA
Expires: Thu Dec 04 15:59:59 2014
SHA1 hash: EFC77FA6BA295580C2A2CD25B56C00606CA21269
```

The signature is timestamped: Mon Jan 27 14:48:55 2014

#### **Timestamp Verified by:**

```
Issued to: Thawte Timestamping CA
Issued by: Thawte Timestamping CA
Expires: Thu Dec 31 15:59:59 2020
SHA1 hash: BE36A4562FB2EE05DBB3D32323ADF445084ED656
```

```
Issued to: Symantec Time Stamping Services CA - G2
Issued by: Thawte Timestamping CA
Expires: Wed Dec 30 15:59:59 2020
SHA1 hash: 6C07453FFDDA08B83707C09B82FB3D15F35336B1
```

```
Issued to: Symantec Time Stamping Services Signer - G4
Issued by: Symantec Time Stamping Services CA - G2
Expires: Tue Dec 29 15:59:59 2020
SHA1 hash: 65439929B67973EB192D6FF243E6767ADF0834E4
```

#### **Cross Certificate Chain:**

```
Issued to: Microsoft Code Verification Root
Issued by: Microsoft Code Verification Root
Expires: Sat Nov 01 05:54:03 2025
SHA1 hash: 8FBE4D070EF8AB1BCCAF2A9D5CCAEC7282A2C66B3
```

```
Issued to: VeriSign Class 3 Public Primary Certification Authority - G5
Issued by: Microsoft Code Verification Root
Expires: Mon Feb 22 11:35:17 2021
SHA1 hash: 57534CCC33914C41F70E2CBB2103A1DB18817D8B
```

```
Issued to: VeriSign Class 3 Code Signing 2010 CA
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Fri Feb 07 15:59:59 2020
SHA1 hash: 495847A93187CFB8C71F840CB7B41497AD95C64F
```

```
Issued to: Contoso, Inc
Issued by: VeriSign Class 3 Code Signing 2010 CA
Expires: Thu Dec 04 15:59:59 2014
SHA1 hash: EFC77FA6BA295580C2A2CD25B56C00606CA21269
```

Successfully verified: toaster.inf

```
Number of files successfully Verified: 1
Number of warnings: 0
Number of errors: 0
```

Note the presence of Microsoft Code Verification Root in the certificate chain.

Next, check embed signing of the toaster.sys file.

```
signtool verify /v /kp toaster.sys
```

Command output:

```
Verifying: toaster.sys Hash of file (sha1): CCF5F5C02FEDE87D92FCB7B536DBF5D5EFDB7B41

Signing Certificate Chain:
Issued to: VeriSign Class 3 Public Primary Certification Authority - G5
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Wed Jul 16 15:59:59 2036
SHA1 hash: 4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5

Issued to: VeriSign Class 3 Code Signing 2010 CA
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Fri Feb 07 15:59:59 2020
SHA1 hash: 495847A93187CFB8C71F840CB7B41497AD95C64F

Issued to: Contoso, Inc
Issued by: VeriSign Class 3 Code Signing 2010 CA
Expires: Thu Dec 04 15:59:59 2014 SHA1 hash:
EFC77FA6BA295580C2A2CD25B56C00606CA21269

The signature is timestamped: Mon Jan 27 14:48:55 2014 Timestamp Verified by:
Issued to: Thawte Timestamping CA Issued by: Thawte Timestamping CA Expires: Thu Dec 31 15:59:59
2020 SHA1 hash: BE36A4562FB2EE05DBB3D32323ADF445084ED656
Issued to: Symantec Time Stamping Services CA - G2 Issued by: Thawte Timestamping CA
Expires: Wed Dec 30 15:59:59 2020 SHA1 hash: 6C07453FFDDA08B83707C09B82FB3D15F35336B1
Issued to: Symantec Time Stamping Services Signer - G4 Issued by: Symantec Time
Stamping Services CA - G2 Expires: Tue Dec 29 15:59:59 2020 SHA1 hash:
65439929B67973EB192D6FF243E6767ADF0834E4

Cross Certificate Chain:
Issued to: Microsoft Code Verification Root Issued by: Microsoft Code Verification Root Expires:
Sat Nov 01 05:54:03 2025 SHA1 hash: 8FBE4D070EF8AB1BCCAF2A9D5CCA7282A2C66B3
Issued to: VeriSign Class 3 Public Primary Certification Authority - G5
Issued by: Microsoft Code Verification Root
Expires: Mon Feb 22 11:35:17 2021
SHA1 hash: 57534CCC33914C41F70E2CBB2103A1DB18817D8B

Issued to: VeriSign Class 3 Code Signing 2010 CA
Issued by: VeriSign Class 3 Public Primary Certification Authority - G5
Expires: Fri Feb 07 15:59:59 2020
SHA1 hash: 495847A93187CFB8C71F840CB7B41497AD95C64F

Issued to: Contoso, Inc Issued by: VeriSign Class 3 Code Signing 2010 CA
Expires: Thu Dec 04 15:59:59 2014 SHA1 hash: EFC77FA6BA295580C2A2CD25B56C00606CA21269

Successfully verified: toaster.sys
Number of files successfully Verified: 1 Number of warnings: 0 Number of errors: 0
```

Again, note the presence of Microsoft Code Verification Root in the certificate chain.

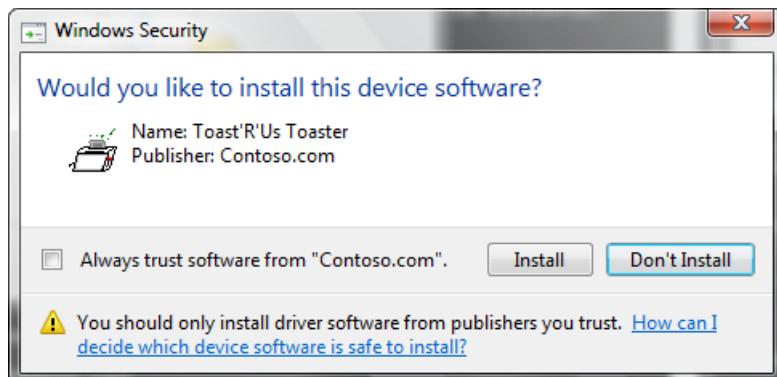
# Troubleshooting Driver Signing Installation

11/2/2020 • 7 minutes to read • [Edit Online](#)

Installing a release-signed driver is the same as described in [Installing, Uninstalling and Loading the Test-Signed Driver Package in Test Signing](#), except for two additional steps needed when installing using any of the four methods described there. Using the Add Hardware Wizard to install a release-signed driver shows the additional two steps, including some common installation issues.

1. Open an elevated command window
2. Run hdwwiz.cpl to start the Add Hardware Wizard, and select Next to go to the second page
3. Select Advanced Option and select Next
4. Select Show all devices from the list box and select Next
5. Select the Have Disk option
6. Enter the path to the folder that contains the C:\toaster driver package
7. Select the inf file and select Open
8. Select OK
9. Select Next in the next two pages, and then select Finish to complete the installation
10. Select Install on the dialog box that asks "Would you like to install this device software?"
11. Select Finish to complete the installation.

Step 10 shows the following Windows Security dialog box.



Selecting the check box will not show this dialog box again on the computer if the driver is installed again or if the driver is removed for any reason.

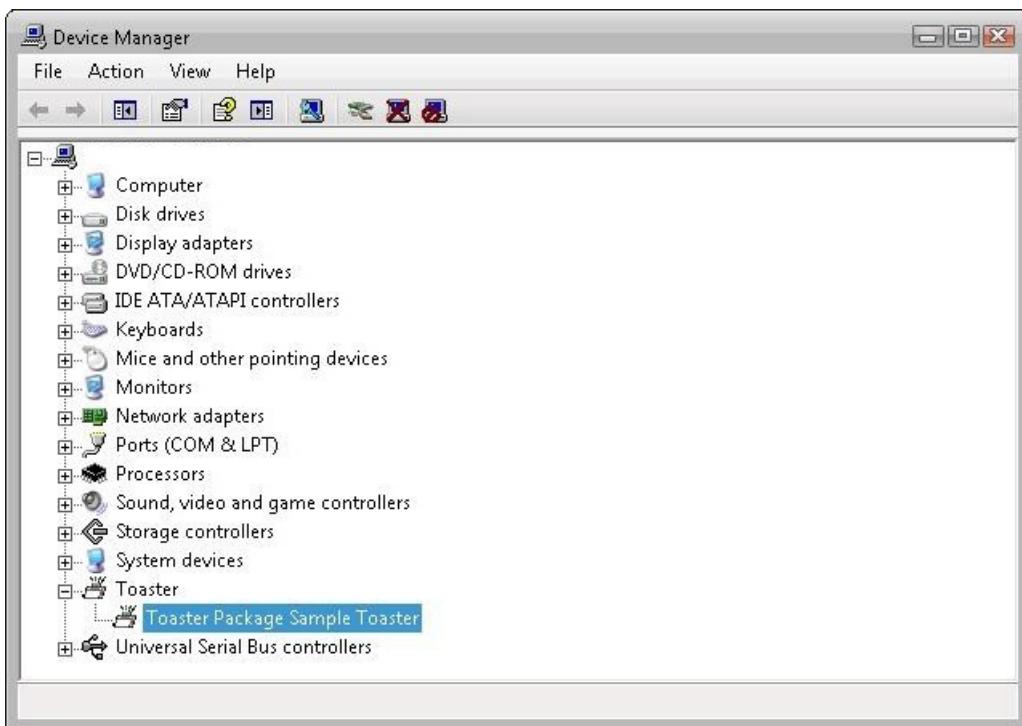
**Note** The system verifies that publisher information is accurate based on the SPC that was used to sign the catalog. If the publisher trust level is unknown—as will be true for Contoso.com—the system displays the dialog box. For the installation to proceed, the user must select Install. For more information on trust and driver installation, see [Code-Signing Best Practices](#).

An unsigned driver will show the following dialog, which allows a user to install an unsigned driver (this may not work in x64 version of Windows).



## Verify that the Release-Signed Driver is Operating Correctly

Use Device Manager to view the driver Properties (described earlier for the test-signed driver). Below is the screen shot to show if the driver is working.



## Troubleshoot Release-Signed Drivers

Several common ways to troubleshoot problems with loading signed or test signed drivers are listed below:

- Use the Add Hardware Wizard or Device Manager to check whether the driver is loaded and signed, as described in [Verify that the Test-Signed Driver Is Operating Correctly](#) of [Test Signing](#).
- Open the setupapi.dev.log file created in the Windows\inf directory after driver install. Refer to the section on setting the registry entry and renaming of the setupapi.dev.log file before installing the driver.
- Check the Windows security audit log and Code Integrity event logs.

## Analyzing the Setupapi.dev.log File

As explained before, any driver installation information will be logged (appended) to the setupapi.dev.log file in the Windows\inf directory. If the file is renamed before a driver is installed, a new log file will be generated. A new log file will be easier to search for important logs from a new driver install. The log file opens in the Notepad

application.

The first left most column may have a single exclamation mark “!” or multiple exclamation marks “!!!”. The single mark is a warning message, but the triple exclamation mark indicates a failure.

You will see the following single exclamation mark when you install a driver package release signed with a CA vendor provided SPC certificate. These are warnings that the cat file has not been verified yet.

```
!    sig:          Verifying file against specific (valid) catalog failed! (0x800b0109)
!    sig:          Error 0x800b0109: A certificate chain processed, but terminated in a root certificate
which is not trusted by the trust provider.
    sig:          Success: File is signed in Authenticode(tm) catalog.
    sig:          Error 0xe0000242: The publisher of an Authenticode(tm) signed catalog has not yet
been established as trusted.
```

Refer to step 10 of driver install and select the “Install” button. You will see the log below, which in most cases means the driver will install and load fine. The Device Manager will not report any errors or a yellow exclamation mark for the driver.

```
!    sto:          Driver package signer is unknown but user trusts the signer.
```

If you also see the following error log in the log file, the driver may not be loaded.

The setupapi.dev.log file has also reported the following error:

```
!!!  dvi:          Device not started: Device has problem: 0x34: CM_PROB_UNSIGNED_DRIVER.
```

Note that 0x34 is Code 52.

To troubleshoot, review the log file and look for exclamation marks next to a driver binary. Run the `signtool verify` command on the cat file and other embedded signed binaries.

Usually the log file information is sufficient to resolve the issue. If the above checks fail to find the root cause, then check the Windows security audit log and code Integrity event logs, described in the next section.

If the service binary file copy has not been committed but the OS tries to start the service, the driver service will fail to start. If this happens, use the setupapi.dev.log file to check driver file copy and commit time information. A restart should successfully start the service. See the example sequence of operation below in the log file.

```
>>> Section start 2014/02/08 14:54:56.463
```

Next at a later time:

```
!    inf:          Could not start service 'toaster'.
```

Then we have the file copy operation:

```

<snip>
flq:           {FILE_QUEUE_COPY}
    flq:           CopyStyle      - 0x00000000
    flq:           SourceRootPath -
'C:\Windows\System32\DriverStore\FileRepository\Toaster.inf_amd64_d9b35403a0fe4391\' 
    flq:           SourceFilename - 'toaster.exe'
    flq:           TargetDirectory- 'C:\Windows\System32\drivers'
    flq:           TargetFilename - 'toaster.exe'

```

The file is committed next. Compare the end time with the start time.

```

flq:           {_commit_file_queue} 14:54:56.711
<snip>
    flq:           {_commit_copyfile}
    flq:           Copying
'C:\Windows\System32\DriverStore\FileRepository\Toaster.inf_amd64_d9b35403a0fe4391\o2flash.exe' to
'C:\Windows\System32\drivers\o2flash.exe'.
    flq:           CopyFile:
'C:\Windows\System32\DriverStore\FileRepository\Toaster.inf_amd64_d9b35403a0fe4391\o2flash.exe' to
'C:\Windows\System32\drivers\SETDA81.tmp'
    flq:           MoveFile: 'C:\Windows\System32\drivers\SETDA81.tmp' to
'C:\Windows\System32\drivers\toaster.exe'
    cpv:           Applied 'OEM Legacy' protection on file
'C:\Windows\System32\drivers\toaster.exe'.
    flq:           Caller applied security to file 'C:\Windows\System32\drivers\toaster.exe'.
    flq:           {_commit_copyfile exit OK}
<snip>
    sto:   {Configure Driver Package: exit(0x00000bc3)}
    ndv:   Restart required for any devices using this driver.
<snip>
<<< Section end 2014/02/08 14:54:57.024
<<< [Exit status: SUCCESS]

```

The above example is a driver update that worked fine in Windows 7, but failed in Windows 8.0 and 8.1, which led to the discovery of a bug.

## Using the Windows Security Audit Log

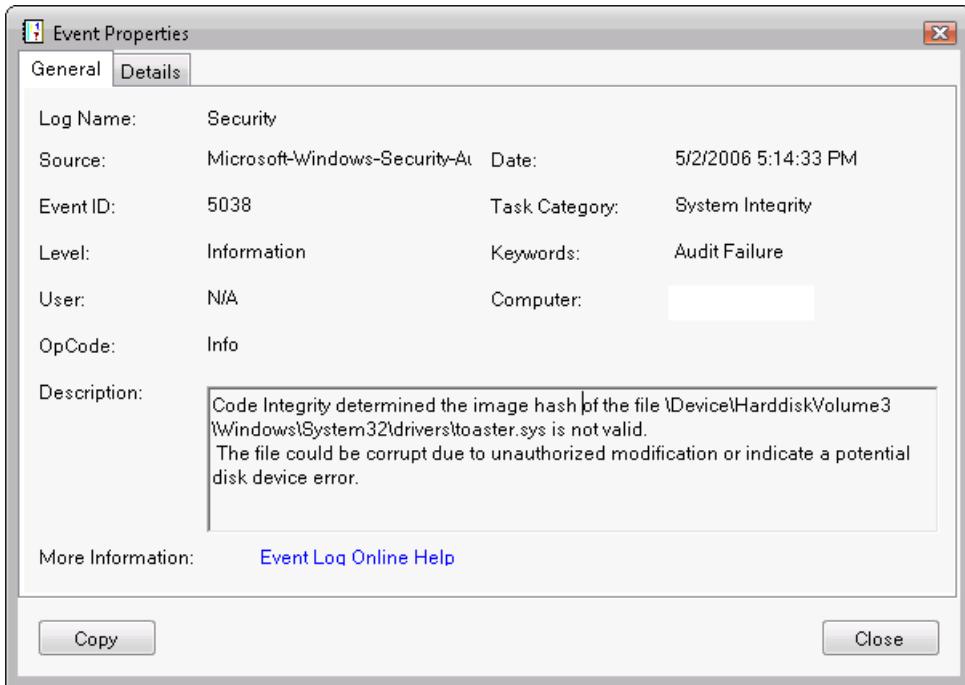
If the driver failed to load because it lacked a valid signature, it will be recorded as an audit failure event. Audit failure events are recorded in the Windows security log, indicating that Code Integrity could not verify the image hash of the driver file. The log entries include the driver file's full path name. The security log audit events are generated only if the local security audit policy enables logging of system failure events.

**Note** The security audit log must be explicitly enabled. For more information, see [Appendix 3: Enable Code Integrity Event Logging and System Auditing](#).

To examine the security log:

1. Open an elevated command window.
2. To start Windows Event Viewer, run Eventvwr.exe. Event Viewer can also be started from the Control Panel Computer Management application.
3. Open the Windows security audit log.
4. Check the log for system integrity events with an event ID of 5038.
5. Select and hold (or right-click) the log entry and select Event Properties to display its Event Properties dialog box, which provides a detailed description of the event.

The screen shot below shows the Event Properties dialog box for a security audit log event that was caused by an unsigned Toaster.sys file.



## Using the Code Integrity Event Operational Event Log

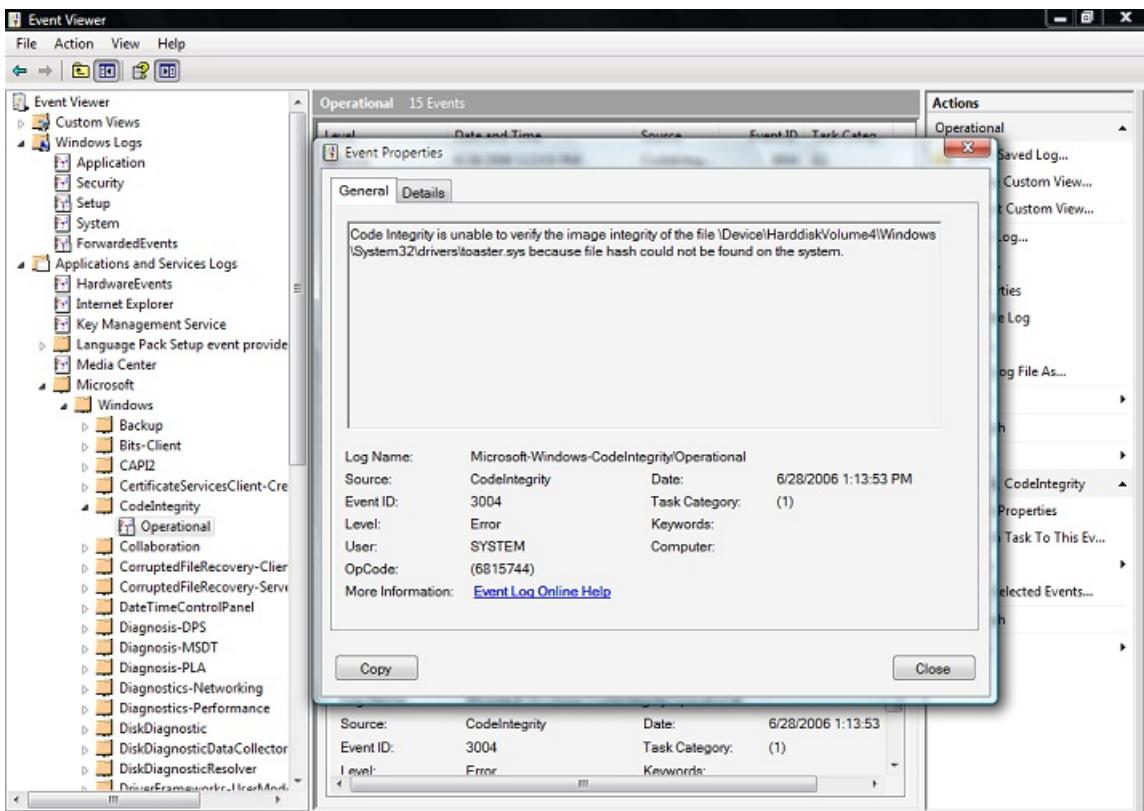
If the driver failed to load because it was not signed or generated an image verification error, Code Integrity records the events in the Code Integrity operational event log. Code Integrity operational events are always enabled.

The Code Integrity events can be viewed with Event Viewer.

### To examine the Code Integrity operational log

1. Open an elevated command window.
2. To start Windows Event Viewer, run Eventvwr.exe. Event Viewer can also be started from the Computer Management Control Panel application.
3. Open the Windows Code Integrity log.
4. Select and hold (or right-click) a log entry and select Event Properties to display its Event Properties dialog box, which provides a detailed description of the event.

The screen shot below shows the Event Properties dialog box for a Code Integrity operational log event that was caused by an unsigned Toaster.sys file.



## Using the Informational Events in the Code Integrity Verbose Log

The Code Integrity informational log's verbose view tracks events for all kernel-mode image verification checks. These events show successful image verification of all drivers that are loaded on the system.

To enable the Code Integrity verbose view:

1. Start Event Viewer, as in the previous example.
2. Select the Code Integrity node to give it focus.
3. Select and hold (or right-click) Code Integrity and select the View item from the shortcut menu.
4. Select Show Analytic and Debug Logs. This creates a sub tree with two additional nodes: Operational and Verbose.
5. Select and hold (or right-click) the Verbose node and select the Properties from the shortcut menu.
6. On the General tab, select Enable Logging to enable the verbose logging mode.
7. Reboot the system to reload all kernel-mode binaries.
8. After rebooting, open the MMC Computer Management snap-in and view the Code Integrity verbose event log.

A few additional known driver signing issues are described in [Appendix 4: Driver Signing Issues](#).

# Appendix 1: Enforcing Kernel-Mode Signature Verification in Kernel Debugging Mode

11/2/2020 • 2 minutes to read • [Edit Online](#)

## Enforcing Kernel-Mode Signature Verification in Kernel Debugging Mode

*Excerpt from [Installing an Unsigned Driver during Development and Test](#):*

In certain cases, developers might have to enable load-time signature enforcement when a kernel debugger is attached. An example of this is when a driver stack has an unsigned driver (such as a filter driver) that fails to load, which may invalidate the entire stack. Because attaching a debugger allows the unsigned driver to load, the problem appears to vanish as soon as the debugger is attached. Debugging this type of issue may be difficult.

In order to facilitate debugging these situations, the kernel-mode code signing policy supports the following registry value:

```
HKLM\SYSTEM\CurrentControlSet\Control\CI\DebugFlags
```

This registry value is of type [REG\\_DWORD](#), and can be assigned a value based on a bitwise OR of one or more of the following flags.

```
0x00000001
```

This flag value configures the kernel to break into the debugger if a driver is unsigned. The developer or tester can then choose to load the unsigned driver by entering g at the debugger prompt.

```
0x00000010
```

This flag value configures the kernel to ignore the presence of the debugger and to always block an unsigned driver from loading.

If this registry value does not exist in the registry or has a value that is not based on the flags described previously, the kernel always loads a driver in kernel debugging mode regardless of whether the driver is signed.

**Note** This registry value does not exist in the registry by default. You must create the value in order to debug the kernel-mode signature verification.

# Appendix 2: Signing Drivers with Visual Studio

11/2/2020 • 2 minutes to read • [Edit Online](#)

## Driver signing using Visual Studio development environment

Newer versions of the WDK integrated with Visual Studio support driver signing directly in the IDE:

[Signing a Driver During Development and Testing](#) [Signing a Driver for Public Release](#)

# Appendix 3: Enable Code Integrity Event Logging and System Auditing

12/5/2018 • 2 minutes to read • [Edit Online](#)

## Enable Code Integrity Event Logging and System Auditing

*Excerpt from [Code Integrity Event Logging and System Auditing](#):*

Code Integrity is the kernel-mode component that implements driver signature verification. It generates system events that are related to image verification and logs the information in the Code Integrity log:

- The Code Integrity operational log view shows only image verification error events.
- The Code Integrity verbose log view shows the events for successful signature verifications.

The following procedure shows how to enable Code Integrity verbose event logging to view all successful operating system loader and kernel-mode image verification events:

### To enable Code Integrity verbose event logging

*Excerpt from [Enabling the System Event Audit Log](#):*

To enable verbose logging, follow these steps:

1. Open an elevated Command Prompt window.
2. Run `Eventvwr.exe` on the command line.
3. Under the **Event Viewer** folder in the left pane of the Event Viewer, expand the following sequence of subfolders:
  - a. Applications and Services Logs
  - b. Microsoft
  - c. Windows
4. Expand the **Code Integrity** subfolder under the **Windows** folder to display its context menu.
5. Select **View**.
6. Select **Show Analytic and Debug Logs**. Event Viewer will then display a subtree that contains an **Operational** folder and a **Verbose** folder.
7. Right-click **Verbose** and then select **Properties** from the pop-up context menu.
8. Select the **General** tab on the **Properties** dialog box, and then select the **Enable Logging** option near the middle of the property page. This will enable verbose logging.
9. Restart the computer for the changes to take effect.

System event records can also be enabled, which include Code Integrity image verification failure events. These events are generated when the Windows kernel fails to load a driver because of a signature failure. Similar events are also recorded in the Code Integrity operational event log view

## To enable the audit policy to generate audit events in the system category for failed operations

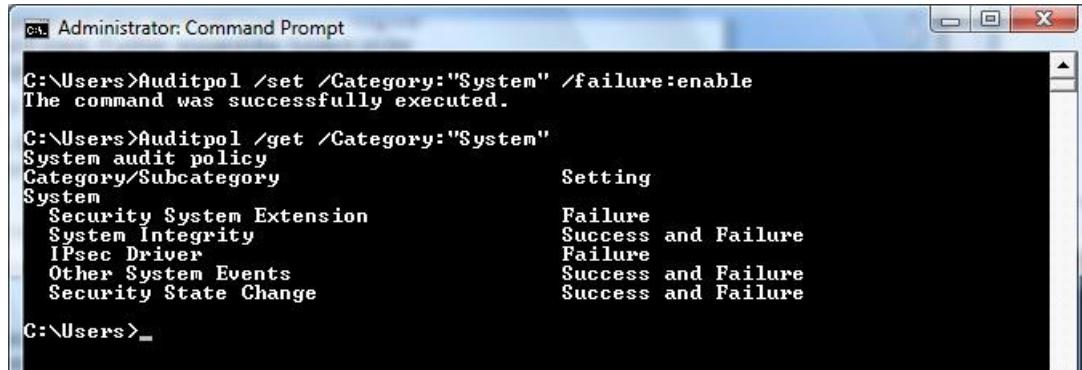
To enable security audit policy to capture load failures in the audit logs, follow these steps:

1. Open an elevated Command Prompt window. To open an elevated Command Prompt window, create a desktop shortcut to *Cmd.exe*, right-click the *Cmd.exe* shortcut, and select **Run as administrator**.
2. In the elevated Command Prompt window, run the following command:

```
Auditpol /set /Category:System /failure:enable
```

3. Restart the computer for the changes to take effect.

The following screen shot shows an how to use Auditpol to enable security auditing.



The screenshot shows a Windows Command Prompt window titled "Administrator: Command Prompt". The window contains the following text:

```
C:\Users>Auditpol /set /Category:"System" /failure:enable
The command was successfully executed.

C:\Users>Auditpol /get /Category:"System"
System audit policy
Category/Subcategory           Setting
System
  Security System Extension    Failure
  System Integrity             Success and Failure
  IPsec Driver                 Failure
  Other System Events          Success and Failure
  Security State Change        Success and Failure

C:\Users>_
```

# Appendix 4: Driver Signing Issues

11/2/2020 • 2 minutes to read • [Edit Online](#)

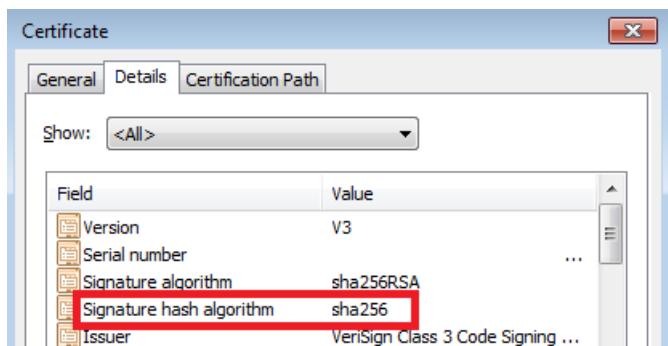
Two known driver signing issues are described below.

## Signing issue with previous OS

Every new release of Windows and subsequently in the released Service Packs, root certificates from Microsoft certified CA vendors, new or existing vendors with new certificates are added to the OS image. For example, Vista, XP, etc. OSes may have unknown signing issues or driver not signed issues if the computer under test is not connected to the internet. If the computer under test is connected to the internet, then the new certificates are automatically downloaded when a driver is installed and there will not be any issues. Sometimes the CA vendors are also able to help out in resolving the issues when the computer under test is not connected to the internet.

## Code 52 error

There is a known bug for Windows 7 x64 OS, when a catalog file (.cat) is signed using a new VeriSign released signing certificate which uses the SHA256 has algorithm. If you open the signed cat file and view signature and select the Details tab you will notice the following:



To resolve the issue, you may ask VeriSign to provide a replacement certificate at no cost signed with the SHA1 hash algorithm.

Alternatively, you can buy another SHA1 certificate and sign the file with two signatures as shown below if you want to keep both certificates. Note that only .sys files can be dual signed because they are PE files.

```
signtool sign /fd sha256 /ac C:\MyCrossCert\Crosscert.cer /s my /n "MyCompany Inc. " /ph /as /sha1 XX...XX  
C:\DriverDir\toaster.SYS
```

Where XX...XX is the hash of the certificate you are using for the secondary signature. Add /tr to timestamp signing.

**Note** Please review Microsoft Security Advisory ([2880823](#)) "Deprecation of SHA-1 Hashing Algorithm for Microsoft Root Certificate Program" which describes a policy change wherein Microsoft will no longer allow root certificate authorities to issue X.509 certificates using the SHA-1 hashing algorithm for the purposes of SSL and code signing after January 1, 2016.

Use of SHA1 certificate will be deprecated by Microsoft starting from January 1, 2016. All CA vendors must to issue signing certificates with the SHA256 hash algorithm.

Windows will stop accepting SHA1 code signing certificates without time stamps after 1 January 2016.

# Overview of Device Metadata Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, device metadata packages contain XML documents that represent the properties of the device and its hardware functions. The Devices and Printers user interface displays device-specific information to the user based on the XML documents from the device's metadata package.

A device metadata package consists of multiple XML documents, and each document specifies various components of the device's attributes.

Starting with Windows 7, a new user interface for devices, Devices and Printers, shows most of the devices that are typically connected to a computer in one window. This window is called the gallery view. For each device that is displayed in the gallery view, Devices and Printers displays device-specific information to the user based on the XML documents from the device's metadata package. By using these XML documents, the OEM can customize which information is included and how this information appears. For example, a device in the gallery view can be represented by a custom icon and descriptive text that the OEM provides.

The XML documents that are contained within device metadata packages specify the information that describes the physical device. The following list shows the type of information that the XML documents can specify:

- The name of the OEM.
- The model name and description of the device.
- One or more functional categories that the device supports.

Each device metadata package consists of the following components:

## [PackageInfo XML document](#)

This document contains data that specifies the contents of the device metadata package. The operating system uses this data to install the package and reference its contents.

This data is formatted based on the [PackageInfo XML schema](#).

## [DeviceInfo XML document](#)

This document contains data that specifies the device's properties, such as device category and model name. The Devices and Printers user interface uses this data to show detailed information about the device.

This data is formatted based on the [DeviceInfo XML schema](#).

## [Device icon file](#)

This file contains a photo-realistic image that represents the device in the Devices and Printers user interface.

## [WindowsInfo XML document](#)

This document contains data that specifies the display actions that the Devices and Printers user interface performs for the specified device in the device metadata package.

This data is formatted based on the [WindowsInfo XML Schema](#).

Each device metadata package has its components compressed into a single file by using the Cabarc (*Cabarc.exe*) tool. For more information about this tool, see to the [Cabarc Overview](#) website.

The file name of the device metadata package uses the following naming convention:

```
<GUID>.devicemetadata-ms
```

The *<GUID>* file prefix is a globally unique identifier (GUID) that is created for the device metadata package. The GUID for each metadata package file name must be unique. When you create a new or revised metadata package, you must create a new GUID, even if the changes are minor.

For more information, see [Building Device Metadata Packages](#).

# Windows Metadata and Internet Services

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Windows Metadata and Internet Services (WMIS) manages device metadata packages that OEMs submit to the [Windows Quality Online Services \(Winqual\)](#) website over the Internet. By using the Winqual site, you can certify hardware devices and software applications for Windows.

When the OEM submits a device metadata package, Winqual completes the following process:

1. Validates the XML documents that are contained within a device metadata package, and digitally signs those packages that pass validation.
2. Makes the package available so that WMIS can distribute and install on remote computers.

In Windows 7 and later versions of Windows, the operating system uses WMIS to discover, index, and match device metadata packages for specific devices that are connected to the computer. For more information about this process, see [Installing Device Metadata Packages from WMIS](#).

# Device Metadata Retrieval Client

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Device Metadata Retrieval Client (DMRC) is the operating system component that matches devices to device metadata packages. When the user opens the gallery view window of the Devices and Printers user interface, the DMRC tries to obtain device metadata for the devices that Devices and Printers will display. First, it checks the local computer's [device metadata cache](#) and [device metadata store](#). If the device is newly installed, or if the device is scheduled for a periodic metadata update, DMRC queries the [Windows Metadata and Internet Services](#) (WMIS) website to determine whether a device metadata package is available for the device. If a device metadata package is available, DMRC automatically downloads the package from WMIS, extracts the package's device metadata components, and saves them within the device metadata cache.

The [PackageInfo XML document](#) (Packageinfo.xml), which is a component of a device metadata package, contains the information that the DMRC needs in order to match a device to the package. The file includes a [MetadataKey](#) XML element that specifies the device-matching information, which comes from one of the following sources:

- A list of one or more hardware IDs that identifies a hardware function that is supported by the device. The list of hardware IDs is specified in the [HardwareIDList](#) child XML element.
- A list of one or more model IDs that identifies a hardware function that is supported by the device. Each model ID is a globally unique identifier (GUID), and the list of model IDs is specified in the [ModelIDList](#) child XML element.

For more information about the XML schema that is referenced by the [PackageInfo XML document](#), see [PackageInfo XML Schema](#).

# Device Metadata Store

12/5/2018 • 2 minutes to read • [Edit Online](#)

The device metadata store is the directory where device metadata packages are stored on the local computer. The device metadata store is accessed from the following directory:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore\<locale>
```

The <locale> subdirectory represents the locale of the device metadata package. The name of this subdirectory can be in the following format:

```
<Language>[-<Region>]
```

For example, a value of EN-US specifies the subdirectory that contains device metadata package that are localized for the English language of the United States.

If only <Language> is specified, the subdirectory that contains device metadata packages that are localized for the specified language in all locations where the language exists. For example, a value of 'EN' applies to 'EN-US' and 'EN-BR'.

A device metadata package is copied to the device metadata store in one of the following ways:

- The OEM or developer copies the device metadata package. For more information, see [Manually Adding Device Metadata Packages](#).
- The device metadata package is copied by using an application that is provided by the OEM. For more information, see [Installing Device Metadata Packages through an Application](#).

**Note** We do not recommend that end-users copy device metadata packages to the device metadata store. Instead, end-users should install device metadata packages by using either the [Windows Metadata and Internet Services \(WMIS\)](#) or an application provided by the OEM.

# Device Metadata Cache

12/5/2018 • 2 minutes to read • [Edit Online](#)

The device metadata cache is the directory where device metadata packages are stored on the local computer.

On Windows 7, the device metadata cache is accessed from the following directory:

```
%LOCALAPPDATA%\Local\Microsoft\Device Metadata\
```

On Windows 8 and later, the device metadata cache is accessed from the following directory:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataCache\
```

When the [Device Metadata Retrieval Client](#) (DMRC) downloads device metadata packages from the Windows Metadata and Services (WMIS) website, it saves them within the device metadata cache. For more information about this process, see [Installing Device Metadata Packages from WMIS](#).

**Note** The device metadata cache is reserved for only the operating system to use. Device metadata packages that are not installed by DMRC, such as through an application that is provided by an OEM, must be copied to the [device metadata store](#) instead.

# Device Metadata Package Structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

Each device metadata package has the following directory structure:

PackagelInfo.xml

DeviceInformation DeviceInfo.xml DeviceIcon.ico

WindowsInformation WindowsInfo.xml

**DeviceStage** When you create a device metadata package, XML documents and icon files are stored in the following directories:

- The [PackagelInfo XML document](#) is at the root of the directory. The name of this XML document must be PackagelInfo.xml.
- The DeviceInformation subdirectory contains the [DeviceInfo XML document](#) and the optional device icon file. The name of the XML document must be DeviceInfo.xml.

If your device metadata package includes a device icon file, it can have any name but must end with a suffix of *.ico*. For more information, see [Device Icon File](#).

- The WindowsInformation subdirectory contains the [WindowsInfo XML document](#). The name of the XML document must be WindowsInfo.xml.
- The DeviceStage subdirectory contains the specific files that are used by Windows Device Stage to present the Device Stage experience. Device Stage is a rich platform for developing and distributing device-specific experiences. With Device Stage, a device maker can create experiences that match the branding, functionality, and services of its device by defining only a few XML files and graphics.

If the device maker uses the Device Stage experience for the device, Windows requires the DeviceStage directory to be in the device metadata package. Otherwise, Windows ignores the directory if it is in the package.

**Note** Device Stage is supported for a limited number of device classes.

More information about Windows Device Experience, Device Stage, and the Device Stage XML schema can be found in the [Microsoft Device Experience Development Kit](#).

When you create a device metadata package, you should follow these guidelines:

- Each XML document must be saved by using UTF-8 encoding.
- The device metadata package is not required to include a device icon. However, we highly recommend that the device metadata package contain a [device icon file](#). This is used to display the photo-realistic image of the device in Devices and Printers.

Starting with the Windows 7 version of the Windows Driver Kit (WDK), the [Toaster Sample](#) provides a sample device metadata package. The XML documents for this package are located in the *src\general\toaster\devicemetadatapackage* subdirectory of the WDK.

# PackagelInfo XML Document

11/2/2020 • 2 minutes to read • [Edit Online](#)

The PackagelInfo XML document contains data that specifies the contents of the device metadata package. The operating system uses this data to install the package and reference its contents.

Components of the device metadata system, such as the [Windows Metadata and Internet Services \(WMIS\)](#) and [Device Metadata Retrieval Client \(DMRC\)](#) use the PackagelInfo XML document to provide the Devices and Printers user interface with the current and most appropriate information for a device, such as the following:

- The hardware or model ID of the device. This information is specified by the [HardwareID](#) and [ModelID](#) XML elements within the PackagelInfo XML document.
- A localized version of the device metadata package that matches the locale of the computer. This information is specified by the [Locale](#) XML element within the PackagelInfo XML document.
- The last modified date of the device metadata package that matches the locale of the computer. This information is specified by the [LastModifiedDate](#) XML element within the PackagelInfo XML document.

Each device metadata package must contain only one PackagelInfo XML document. The name of the document must be *PackagelInfo.xml*.

The data in the PackagelInfo XML document is formatted based on the [PackagelInfo XML schema](#).

# DeviceInfo XML Document

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Devices and Printers user interface displays detailed information about the device that is based on the DeviceInfo XML document from the device's metadata package. This XML document contains data that specifies the device's properties, such as the following:

- The functional category of the device. This information is specified by a **DeviceCategory** XML element, which is a child element of the **DeviceCategoryList** XML element within the DeviceInfo XML document.

The first **DeviceCategory** XML element within the **DeviceCategoryList** XML element is considered the *primary functional category* of the device. When the Devices and Printers user interface first displays the gallery view of devices that are connected to the computer, or if the end-user has filtered devices by category, devices are displayed based on their primary functional category.

For multifunction devices, additional functional categories are specified by a separate **DeviceCategory** XML child element that follows the first **DeviceCategory** element within the **DeviceCategoryList** XML element. Again, if the end-user filters devices within the Devices and Printers user interface based on device category, devices are displayed based on their primary and additional functional categories.

- The model name of the device. This information is specified by the **ModelName** XML element within the DeviceInfo XML document.
- The description of the device. This information is specified by the **DeviceDescription1** and **DeviceDescription2** XML elements within the DeviceInfo XML document.
- The manufacturer of the device. This information is specified by the **Manufacturer** XML elements within the DeviceInfo XML document.
- An icon that represents the device. This information is specified by the **DeviceIconFile** XML element within the DeviceInfo XML document.

For more information about device icons, see [Device Icon File](#).

Each device metadata package must contain only one DeviceInfo XML document. The name of the document must be DeviceInfo.xml.

The data in the DeviceInfo XML document is formatted based on the [DeviceInfo XML schema](#).

# Device Icon File

12/1/2020 • 2 minutes to read • [Edit Online](#)

A device metadata package can contain one photo-realistic image, or icon, that represents the device in the Devices and Printers user interface. The image is stored in an icon file, and the file name must be specified in the [DeviceIconFile](#) element of the package's [DeviceInfo XML document](#).

If the device metadata package does not contain a device icon file and [DeviceIconFile](#) element, the Devices and Printers user interface displays a default icon for the device. This icon is based on the device's category type that is specified in the [DeviceCategory](#) element of the [DeviceInfo XML document](#).

**Note** We highly recommend that the device metadata package contain a device icon file, which is used to display the photo-realistic image of the device in the Devices and Printers user interface. For more information about how to create icons that have the same display qualities of Windows graphical elements, refer to [Icons](#) in the Microsoft SDK.

# WindowsInfo XML Document

11/2/2020 • 2 minutes to read • [Edit Online](#)

This document contains data that specifies the display actions that the operating system performs for the specified device in the device metadata package. These actions include the following:

- Whether the **device icon** is displayed when the device is in a disconnected state, such as when the user removes the device. This action is specified by the [ShowDeviceInDisconnectedState](#) XML element within the WindowsInfo XML document.
- Whether a Device Stage user interface appears when the device transitions from a disconnected state to a connected state, such as when the user plugs in the device. This action is specified by the [LaunchDeviceStageOnDeviceConnect](#) XML element within the WindowsInfo XML document.
- Whether a Device Stage user interface is started when the user double-clicks the **device icon** that appears in either the Devices and Printers user interface or in Windows Explorer. This action is specified by the [LaunchDeviceStageFromExplorer](#) XML element within the WindowsInfo XML document.

Each device metadata package must contain only one WindowsInfo XML document. The name of the document must be *WindowsInfo.xml*.

The data in the WindowsInfo XML document is formatted based on the [WindowsInfo XML Schema](#).

# SoftwareInfo XML Document

11/2/2020 • 2 minutes to read • [Edit Online](#)

For Windows 8 or later, a metadata package may contain one Softwareinfo.xml file, which contains information for downloading an app that is associated with the device and giving privileged access to applications.

The data within the Softwareinfo.xml file is formatted based on the [SoftwareInfo XML schema](#).

# Building Device Metadata Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic provides guidelines on how to build device metadata packages.

## Device metadata package file names

Before you create the device metadata package file, you must first create a globally unique identifier (GUID) for the metadata package. To do this, use the Guidgen tool (*Guidgen.exe*) that is described in the [GUID Generation](#) website.

The file name of the device metadata package must use the following naming convention:

```
<GUID>.devicemetadata-ms
```

For example, if you create a GUID that has the value of {20f001a99-4675-8707-248ca-187dfd9}, you use that GUID to create the following device metadata package file:

```
20f001a99-4675-8707-248ca-187dfd9.devicemetadata-ms
```

**Note** The operating system recognizes device metadata packages only if it has a suffix of *.devicemetadata-ms*.

The following rules apply to device metadata package files:

- The GUID for each metadata package file name must be unique. When you create a new or revised metadata package, you must create a new GUID, even if the changes are minor.
- Each metadata package can support only one locale. If you support more than one locale for your device, you must create separate metadata packages for each locale, with each metadata package having its own GUID. For more information, see [Locale XML element](#).

**Note** If you require multiple locale-specific device metadata package files for your device, you can group all the files by creating a language-neutral identifier. This identifier is a GUID, and the same GUID can be specified in the [LanguageNeutralIdentifier](#) XML element within all metadata packages for the same device.

- The *<GUID>* prefix of the device metadata package file name must specify the GUID without the '{' or '}' delimiters.

## Creating a device metadata package file

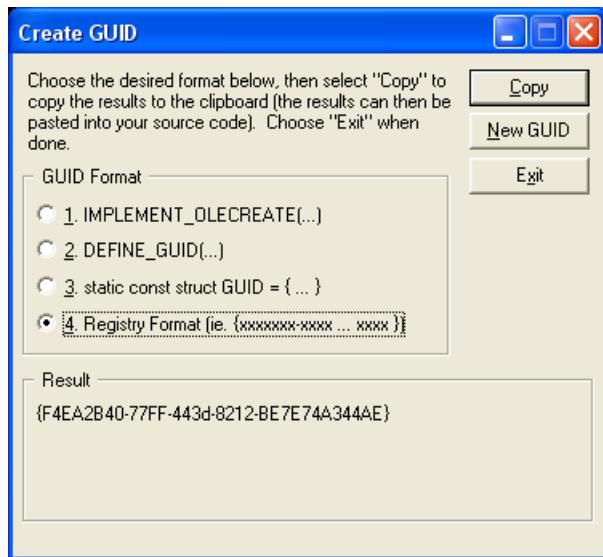
The [components of a device metadata package](#) are stored in a file compressed by using the Cabarc (*Cabarc.exe*) tool. For more information about this tool, refer to the [Cabarc Overview](#) website.

The following code example shows how to use the Cabarc tool to create a device metadata package file. In this example, the components of the metadata package are located in a local directory named *MyMetadataPackage*. The following list shows the subdirectories and files within the *MyMetadataPackage* directory:

```
. \MyMetadataPackages  
. \MyMetadataPackage\PackageInfo.xml  
. \MyMetadataPackage\DeviceInformation\DeviceInfo.xml  
. \MyMetadataPackage\DeviceInformation\MyIcon.ico  
. \MyMetadataPackage\WindowsInformation\WindowsInfo.xml
```

First, a GUID with the value of {f4ea2b40-77ff-443d-8212-be7e74a344ae} is created for the device metadata

package. The following figure shows how to use the Guidgen tool to create the GUID:



Then, the following command uses the Cabarc tool to create a new device metadata package file in a local directory named *MyDeviceMetadataPackage*.

```
Cabarc.exe -r -p -P .\MyMetadataPackage\  
N .\MyDeviceMetadataPackage\f4ea2b40-77ff-443d-8212-be7e74a344ae.devicemetadata-ms  
.\\MyMetadataPackage\\PackageInfo.xml  
.\\MyMetadataPackage\\DeviceInformation\\DeviceInfo.xml  
.\\MyMetadataPackage\\DeviceInformation\\MyIcon.ico  
.\\MyMetadataPackage\\WindowsInformation\\WindowsInfo.xml
```

**Note** Each metadata package can support only one locale. If you support more than one locale for your device, you must create separate metadata packages for each locale, with each metadata package having its own GUID.

# Installing Device Metadata Packages to an Offline Windows Image

12/5/2018 • 2 minutes to read • [Edit Online](#)

Computer OEMs can add device metadata packages to an offline image of Windows by copying the packages to the local device metadata store. This store is in the following location:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore\<locale>
```

You must first create the `<locale>` subdirectory based on the target locale of the device metadata package. Then you must copy the metadata package to the appropriate `<locale>` subdirectory.

For example, a device metadata package that is localized for the English language of Great Britain must be copied to the following location:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore\EN-GB
```

A device metadata package that is localized for the Japanese language must be copied to the following location:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore\JA
```

For more information, see [Device Metadata Store](#).

# Installing Device Metadata Packages from WMIS

11/2/2020 • 2 minutes to read • [Edit Online](#)

When the operating system detects a new device, it queries an online service called the [Windows Metadata and Internet Services](#) (WMIS) for a metadata package for the device. If a device metadata package is available, the Device Metadata Retrieval Client (DMRC) that runs on the local computer downloads the package from WMIS and installs the package on the local computer.

**Note** Device metadata packages are not downloaded from WMIS if the current user is logged in by using any account with only guest privileges, such as the built-in Guest account.

If you submit your device metadata package to [Windows Quality Online Services \(Winqual\)](#) when you submit your [driver package](#) to the [Hardware Certification Kit \(HCK\)](#) for digital signing, your package will be available to WMIS for download requests made by DMRC on any computer that runs Windows 7 and later versions of Windows.

**Important** We highly recommended that OEMs distribute device metadata packages only through WMIS. Distribution of device metadata packages through WMIS supports the *hardware-first* installation scenario. In this scenario, a new device is installed before the driver and device-specific software for the device is installed. For more information about this scenario, see [Hardware-First Installation](#).

A device metadata package is installed through WMIS in the following way:

1. When the user opens the gallery-view window of the Devices and Printers user interface, the [Device Metadata Retrieval Client](#) (DMRC) tries to obtain device metadata for the devices that the Devices and Printers user interface displays.

DMRC first searches the local computer's [device metadata cache](#) and [device metadata store](#) for device metadata. If the device is newly installed, or if the device is scheduled for a periodic metadata update, DMRC queries WMIS for available metadata packages for the device.

2. If a device metadata package is available, DMRC automatically downloads the package from WMIS, extracts the package's device metadata components, and saves them within the [device metadata cache](#).

# Installing Device Metadata Packages through an Application

11/2/2020 • 2 minutes to read • [Edit Online](#)

To install device metadata packages in the [device metadata store](#) by using an application, such as a device installation application, follow these steps:

1. The application first queries the path of the [device metadata store](#) by calling the [SHGetKnownFolderPath](#) function. The [KNOWNFOLDERID](#) GUID for the device metadata store is {5CE4A5E9-E4EB-479D-B89F-130C02886155}.
2. The application then copies the device metadata package to the device metadata store by calling the [CopyFile](#) function.

**Note** The application must be running with administrator privileges or started from an elevated command prompt window.

When your application copies the device metadata package to the [device metadata store](#), it must complete the following steps:

1. If a subdirectory does not exist in the device metadata store for the locale of your device metadata package, the application must create the subdirectory by using the name of the target locale.  
For example, if the locale of the package is EN-US, the application must create the *EN-US* subdirectory under the path of the device metadata store if the subdirectory does not currently exist.
2. Copy the device metadata package to the appropriate *</locale>* subdirectory within the [device metadata store](#).

**Note** If you use the [CopyFile](#) function to copy the device metadata package, specify the full path name, which includes the appropriate *</locale>* subdirectory. By doing this, [CopyFile](#) creates the associated subdirectories for your package if they do not exist on the local computer.

After the device metadata package is installed in the [device metadata store](#), the [Device Metadata Retrieval Client](#) (DMRC) accesses the device metadata package and presents the device information to the Devices and Printers user interface.

# Installing Device Metadata Packages through a Driver Package

5/8/2019 • 2 minutes to read • [Edit Online](#)

A [driver package](#) can install device metadata packages by copying them to the [device metadata store](#). This is accomplished by using [INF CopyFiles directives](#) within the [DestinationDirs](#) and [DDInstall](#) sections of the [INF file](#) for the driver package.

**Note** We highly recommend that you install device metadata packages from the WMIS server instead of through driver packages. For more information, see [Installing Device Metadata Packages from WMIS](#).

To install device metadata packages through a [driver package](#), you must follow these guidelines:

- The device metadata packages must be copied to the [device metadata store](#) by using INF directives. The metadata packages must not be copied by a [co-installer](#).
- If your driver package is used to install devices on versions of Windows earlier than Windows 7, you must use a separate [INF DDInstall section](#) that contains your metadata-related INF directives. You must specify this section name in the [INF Models section](#) by using a *TargetOSversion* decoration that specifies an *OSMajorVersion* and *OSMinorVersion* value for Windows 7 or later versions of Windows.

**Note** If you do not use a separate INF *DDInstall* section that is decorated for Windows 7 or later versions of Windows, the installation of your digitally signed [driver package](#) will result in a signature alert when installed on versions of Windows earlier than Windows 7.

For more information, see [Combining Platform Extensions with Operating System Versions](#).

- All metadata packages in the driver package must be copied to the correct locale-specific folder in the [device metadata store](#). This is needed in order to support dynamic changes to locale.
- The COPYFLG\_NODECOMP flag (0x00000800) is required in the [INF CopyFiles directives](#) that specify the device metadata packages. This flag guarantees that the binary integrity of the device metadata package is retained and avoids a decompression of the device metadata package when the driver package is installed.
- You must first digitally sign the device metadata package before you digitally sign the driver package. For more information about digital signing, see [Driver Signing](#).
- Any failure during the installation of the metadata package installation causes the driver installation to fail.

The following example shows how to copy device metadata packages to locale-specific directory paths by using an INF file for the device metadata store within the [DestinationDirs Section](#) and [DDInstall](#) INF sections:

```
[SourceDisksNames]
1 = %Media_Description%,,\MetadataPackage ;

[SourceDisksFiles.NTx86]
GUID1.devicemetadata-ms= 1,, ;A metadata package file for EN-US
GUID2.devicemetadata-ms= 1,, ;A metadata package file for AR-SA
GUID3.devicemetadata-ms= 1,, ;A metadata package file for JA-JP

[DestinationDirs]
COPYMETADATA_EN-US = 24, \ProgramData\Microsoft\Windows\DeviceMetadataStore\EN-US ;
COPYMETADATA_AR-SA = 24, \ProgramData\Microsoft\Windows\DeviceMetadataStore\AR-SA ;
COPYMETADATA_JA-JP = 24, \ProgramData\Microsoft\Windows\DeviceMetadataStore\JA-JP ;
. . .

[DeviceInstall.ntx86]
CopyFiles=COPYMETADATA_EN-US
CopyFiles=COPYMETADATA_AR-SA
CopyFiles=COPYMETADATA_JA-JP

[COPYMETADATA_EN-US]
GUID1.devicemetadata-ms,,,0x00000800 ;COPYFLG_NODECOMP
[COPYMETADATA_AR-SA]
GUID2.devicemetadata-ms,,,0x00000800 ;COPYFLG_NODECOMP
[COPYMETADATA_JA-JP]
GUID3.devicemetadata-ms,,,0x00000800 ;COPYFLG_NODECOMP
```

# Manually Adding Device Metadata Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

Device metadata packages can be installed on a computer in the following ways:

- An OEM can use the Windows 7 OEM Preinstall Kit (OPK) to update an offline image of the operating system. By using the OPK, the OEM copies the device metadata package to the image's [device metadata store](#).
- Developers can copy their device metadata packages to the [device metadata store](#) to test and debug the installation of a package. For more information, see [Debugging Device Metadata Packages](#).

**Note** We do not recommend that end-users copy device metadata packages to the device metadata store. Instead, end-users should install device metadata packages by using either the Windows Metadata and Internet Services (WMIS) or an installation application that is provided by the OEM.

The following path shows the location of the [device metadata store](#):

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore
```

To copy device metadata packages to the [device metadata store](#), complete the following steps:

1. If a subdirectory does not exist in the device metadata store for the locale of your device metadata package, you must create the subdirectory by using the name of the target locale.

For example, if the locale of the package is EN-US, you must first create the following directory if it does not currently exist:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataStore\EN-US
```

2. Copy your device metadata package to the appropriate *<locale>* subdirectory of the [device metadata store](#).

After the device metadata package is installed in the [device metadata store](#), the [Device Metadata Retrieval Client](#) (DMRC) accesses the device metadata package and presents the device information to the Devices and Printers user interface.

# How the DMRC Selects a Device Metadata Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

When the Devices and Printers or Device Stage user interfaces are opened, the operating system starts the device metadata retrieval client ([DMRC](#)) to search its cache for the most appropriate and current metadata package for a device. The DMRC also searches for a newer metadata package for the device on the Windows Metadata and Internet Services ([WMIS](#)) server. If one is found, the DMRC downloads the package and installs it on the computer.

**Note** If the DMRC recently downloaded a metadata package for a device, it uses the cached metadata package for the device instead of searching the WMIS server for a newer package. For more information, see [How the DMRC Determines When to Search the WMIS Server](#).

The DMRC uses the following metadata XML elements, which are specified in the package, to select the appropriate package for a device. The order of these XML elements reflects the priority that the DMRC uses to select a metadata package:

- [ModelID](#) and [ModelIDList](#)
- [HardwareID](#) and [HardwareIDList](#)
- [Locale](#)
- [LastModifiedDate](#)

The [DMRC](#) follows these steps when it selects a metadata package for a device:

1. If the device has a model ID, the DMRC searches device metadata packages for a match between a [ModelID](#) entry in the package's [ModelIDList](#) XML element and the device's model ID value.
2. If the device does not have a model ID, the DMRC searches device metadata packages for a match between the [HardwareID](#) entries in the package's [HardwareIDList](#) XML element and the device's hardware IDs.
3. The DMRC creates a list of device metadata packages that meet the search criteria described in steps 1 and 2. From this list, the DMRC then searches the list entries for a match between the package's [Locale](#) XML element and the list of preferred user locales on the computer.

If no entries in the list match this search criterion, the DMRC searches the entries in the list for a device metadata package that contains a [Locale](#) XML element that has the **default** attribute set to **true**. If the DMRC finds a match, it selects that metadata package.

4. If the DMRC finds more than one device metadata package during step 3, it selects the package that has a [LastModifiedDate](#) XML element that has the most recent time stamp.

The following points are relevant to the selection algorithm that is used by the [DMRC](#):

- If the DMRC selects a metadata package that is based on hardware IDs, it uses the same ranking of hardware IDs that the operating system uses during driver installation. The DMRC ranks more-specific hardware IDs larger than less-specific hardware IDs. For example, the following hardware IDs are listed in ranking order:

```
<HardwareID>D0ID:USB\VID_XXXX&PID_YYYY&REV_0000</HardwareID>
<HardwareID>D0ID:USB\VID_XXXX&PID_YYYY</HardwareID>
```

For the information about hardware IDs, see [Hardware IDs](#).

- Only one metadata package for a device should set the **default** attribute of the [Locale](#) XML element to

true. You should only set this attribute to true in the package that contains a hardware ID with the highest ranking value.

- The **LastModifiedDate** XML element is used for versioning purposes and is used to select a newer version of a device metadata package for a device.
- If two or more device metadata packages in the local metadata store contain the same values for the **ModelIDList**, **HardwareIDList**, **Locale**, or **LastModifiedDate** XML elements, the DMRC selects only one of them for the device. In this case, the DMRC selects one of these packages in a nondeterministic manner.

For more information about the device metadata XML schema and elements, see [Device Metadata Schema Reference](#).

# How the DMRC Determines When to Search the WMIS Server

11/2/2020 • 3 minutes to read • [Edit Online](#)

The device metadata retrieval client ([DMRC](#)) maintains a cache of device metadata packages. This cache is populated with metadata packages that the DMRC downloaded from the Windows Metadata and Internet Services ([WMIS](#)) server.

When the Devices and Printers or Device Stage user interfaces (UIs) are opened, the DMRC searches its cache for the most appropriate and current metadata package of every device that is displayed in the UI.

Periodically, before it selects a metadata package from its cache, the DMRC searches for a newer metadata package for a device on the [WMIS](#) server. If one is found, the DMRC downloads the package and installs it in its cache on the computer. Then, the DMRC selects the newer version of the metadata package from its cache.

Based on the following values, the DMRC determines when to search the [WMIS](#) server for a newer metadata package for a device:

## LastCheckedDate

This value indicates the most recent date when the DMRC queried the WMIS server for metadata for a device. This date does not reflect whether the DMRC successfully downloaded a metadata package.

The DMRC manages an index table that contains the properties for the device metadata package of each [device ID](#) in the system. The **LastCheckedDate** value is a field of each row in this index table.

## CheckBackMDNotRetrieved

This registry value indicates the number of days that the DMRC waits before it repeats a query of the WMIS server for a device metadata package. This value applies to devices for which the DMRC has not yet downloaded metadata from WMIS.

The **CheckBackMDNotRetrieved** value is located under the following registry key:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Device Metadata
```

The following table describes the format and value range for the **CheckBackMDNotRetrieved** value.

DATA TYPE	VALUE RANGE	DEFAULT VALUE
<a href="#">REG_DWORD</a>	0 to 256, inclusive	5

## CheckBackMDRetrieved

This registry value indicates the number of days that the DMRC waits before it queries for updated device metadata packages. This value applies to devices for which the DMRC previously downloaded metadata packages.

The **CheckBackMDRetrieved** value is located under the following registry key:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Device Metadata
```

The following table describes the format and value range for the **CheckBackMDRetrieved** value.

DATA TYPE	VALUE RANGE	DEFAULT VALUE
REG_DWORD	0 to 256, inclusive	8

**Note** The WMIS server, in cooperation with the DMRC, sets the values for the **CheckBackMDRetrieved** and **CheckBackMDNotRetrieved** registry values on the client system. These values are set based on network conditions and load balancing. Every response from the WMIS server contains the client configuration data and controls the DMRC behavior.

The DMRC follows these steps to determine whether it has to search the WMIS server for a newer metadata package for a device:

1. If the target device's **device ID** is not listed in the DMRC index table, the DMRC is searching for the device's metadata package for the first time. In this case, the DMRC queries the WMIS server.
2. If the target device's **device ID** is listed in the DMRC index table, the DMRC calculates whether it is time to query the WMIS server again for a metadata package for the device. In this case, the DMRC queries the WMIS server in the following way:
  - a. If the DMRC has previously downloaded a device metadata package for the device, the DMRC compares the value of the **CheckBackMDRetrieved** registry key to the value of today's date minus the **LastCheckedDate** value. If the **CheckBackMDRetrieved** value is smaller, the DMRC queries the WMIS server.
  - b. If the DMRC has not previously downloaded a device metadata package for the device, the DMRC compares the value of the **CheckBackMDNotRetrieved** registry key to the value of today's date minus the **LastCheckedDate** value. If the **CheckBackMDNotRetrieved** value is smaller, the DMRC queries the WMIS server.

# Debugging Device Metadata Packages By Using Event Viewer

11/2/2020 • 3 minutes to read • [Edit Online](#)

Starting with Windows 7, the Event Tracing for Windows (ETW) service supports the DeviceMetadata/Debug channel for events that are related to the processing of device metadata packages.

The DeviceMetadata/Debug channel stores error and informational events that occur during the download or processing of device metadata packages. This channel also stores warning and informational events that provide additional status about the detection and query of device metadata packages within the [device metadata store](#).

## **Viewing Device Metadata/Debug ETW Events through Event Viewer**

You use Event Viewer to view events that are logged for device metadata packages. Follow these steps to open the DeviceMetadata/Debug ETW channel in Event Viewer to view these events:

1. On the **Start** menu, right-click **Computer**, and then click **Manage**.
2. Expand the **System Tools** node.
3. Expand and then click the **Event Viewer** node.
4. On the **View** menu, click **Show Analytic and Debug Logs**.
5. Expand the **Applications and Services Logs** node.
6. Expand the **Microsoft** node.
7. Expand the **Windows** node.
8. Expand the **UserPnP** node.
9. Click the **DeviceMetadata/Debug** node.

**Note** You must first enable logging on the DeviceMetadata/Debug ETW channel to receive and view events. To do this, right-click the **DeviceMetadata/Debug** node and select **Properties**. Then, click **Enable Log**.

## **Device Metadata/Debug ETW Events**

The operating system logs the following error, warning, and informational events during the download or processing of a device metadata package:

Event ID: 7900 Error: Device metadata package error

An error was detected with one of the components of a device metadata package.

This event log message contains the following information:

- A description of the error, which includes a description of the source of the error. The error source is either the [device metadata store](#) or the [device metadata cache](#).
- The name of the device metadata package.
- An application-specific error code. For more information about these error codes, see [Device Metadata Error Codes](#).
- A Win32 error code.

Event ID: 7901 Information: Device metadata package downloaded from WMIS.

A device metadata package was downloaded from the [Windows Metadata and Internet Services](#) (WMIS) by the [Device Metadata Retrieval Client](#) (DMRC), which extracts the components from the package and saves them within the [device metadata cache](#).

This event log message contains the following information:

- A description of the event.
- The location of the unpacked device metadata package within the [device metadata cache](#).
- The name of the device metadata package.

Event ID: 7902 Error: Device metadata package not signed.

An installed device metadata package was not signed by the [Windows Quality Online Services \(Winqual\)](#).

**Note** The signature of the device metadata package is verified only when it is downloaded from WMIS.

This event log message contains the following information:

- A description of the error.
- The name of the device metadata package.
- An application-specific error code. For more information about these error codes, see [Device Metadata Error Codes](#).
- A Win32 error code.

Event ID: 7950 Information: New device metadata package discovered in the local metadata store.

The DMRC has detected a new device metadata package that is installed on the local computer.

This event log message contains the following information:

- A description of the event.
- The source of the device metadata package, which is either the [device metadata store](#) or the [device metadata cache](#).
- The name of the device metadata package.

Event ID: 7951 Information: Query for metadata packages in progress.

The DMRC queries installed device metadata packages for a particular device.

This event log message contains the following information:

- A description of the event.
- A device lookup key, such as the device's hardware ID or model ID. For more information, see [HardwareID](#) and [ModelID](#).

**Note** Only the most specific hardware ID is logged when a list of hardware IDs are passed as a parameter.

Event ID: 7952 Warning: Network-related errors.

The DMRC encountered a network error during the download of a device metadata packaged from the WMIS.

**Note** This warning is not generated when the network is not available.

This event log message contains the following information:

- A detailed description of the error.
- An application-specific error code. For more information about these error codes, see [Device Metadata Error Codes](#).

- The HTTP status code at the time of the network error.

# Debugging Device Metadata Packages by Using Problem Reports

12/5/2018 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, the operating system sends problem reports about device metadata package errors (error code 0x50000xx) to the Windows Error Report (WER) server. These reports provide useful debug information to help diagnose problems with your device metadata package.

For more information about the device metadata package errors, see [Device Metadata Error Codes](#).

You can use either Action Center or Event Viewer to view the problem reports that the operating system has sent or will soon send to the Windows Error Report Server.

## **Viewing problem reports by using Action Center**

Follow these steps to view device metadata package error reports by using Action Center:

1. On the **Start** menu, type "view all problem reports" and press ENTER.
2. Select a problem report that you want to review.

The report contains detailed information for the error.

## **Viewing error reports by using Event Viewer**

You can view problem reports in Event Viewer. Follow these steps to view device metadata package problem reports by using Event Viewer:

1. On the **Start** menu, right-click **Computer**, and then click **Manage**.
2. Expand the **System Tools** node.
3. Expand and then click the **Event Viewer** node.
4. Expand the **Windows Logs** node.
5. Right-click **Application**, and then click **Filter Current Log**.
6. Type "1001" in the **Event ID** text box, and then click **OK**.

The **Event ID** text box is the unlabeled text box in the middle of the dialog box that has the default contents of "<All Event Ids>".

## **Interpreting a problem report**

Every Device Metadata Retrieval Client problem report contains the following information:

- An application-specific error code. For more information about these error codes, see [Device Metadata Error Codes](#).
- A Win32 error code.
- The source of the device metadata package, which is either the [device metadata store](#) or the [device metadata cache](#).
- The name of the device metadata package.

# Device Metadata Error Codes

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, the operating system logs the following error codes within events that are related to the download and processing of device metadata packages. These events are managed by the Event Tracing for Windows (ETW) service and can be viewed by using Event Viewer. For more information about these events, see [Debugging Device Metadata Packages By Using Event Viewer](#).

## Windows Metadata and Internet Services (WMIS) Errors (200000xx)

ERROR CODE	EXPLANATION
20000021	The request does not contain a device metadata request.
20000022	The request batch size exceeds the maximum allowed value.
20000023	The locale value is invalid.
20000024	The request does not contain valid header information.
20000025	The request format is invalid.
20000031	An error occurred at the service side when processing the request.

## Device Metadata Retrieval Client (DMRC) Errors (0x400000xx)

ERROR CODE	EXPLANATION
40000011	There is no local metadata cache.
40000012	The structure (folders) in the local metadata cache is not correct.
40000021	There is no local metadata store.
40000022	The structure (folders) in the local metadata store is corrupted.
40000031	The DMRC index data is missing.

ERROR CODE	EXPLANATION
40000032	The DMRC index data is corrupted.

#### Device Metadata Package Errors (0x500000xx)

ERROR CODE	EXPLANATION
50000011	The <i>.devicemetadata-ms</i> cabinet ( <i>cab</i> ) file is corrupted.
50000012	The <i>.devicemetadata-ms</i> cab file does not have correct device metadata structure.
50000021	<i>PackageInfo.xml</i> is missing.
50000022	<i>PackageInfo.xml</i> is not well-formed and cannot be parsed. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <b>Note</b> This error code includes the cases where either the <i>PackageInfo.xml</i> document is missing required elements, or one or more of its elements are not valid based on the syntax of the <a href="#">PackageInfo XML Schema</a>.           </div>
50000031	<i>DeviceInfo.xml</i> is missing.
50000032	<i>DeviceInfo.xml</i> is not well-formed and cannot be parsed.
50000033	<i>DeviceInfo.xml</i> is missing required elements.
50000034	Elements in <i>DeviceInfo.xml</i> are not valid based on the XML schema definition.
50000041	<i>WindowsInfo.xml</i> is missing.
50000042	<i>WindowsInfo.xml</i> is not well-formed and cannot be parsed.
50000043	<i>WindowsInfo.xml</i> is missing required elements.
50000044	Elements in <i>WindowsInfo.xml</i> are not valid based on the XML schema definition.

#### WMIS Query (0x7000xxxx)

ERROR CODE	EXPLANATION
70000408	The WMIS server is not down, but the request timed out.
70000500	The WMIS server returned an internal error, but a detailed error code is not available.
70000503	The WMIS server is busy and cannot service the request.

# Best Practices for Specifying Hardware IDs

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **HardwareIDList** element specifies one or more hardware identification strings for the device. Each string is specified by a **HardwareID** XML element. The operating system first queries the device for its identification string and then loads the device metadata package that has a **HardwareID** value that matches this string.

Each **HardwareID** element value must be a [hardware ID](#) that is unique to the device. You must not use the [compatible ID](#) for the device, such as USB Class ID, 'HID\_DEVICE', or 'HID\_DEVICE\_SYSTEM\_KEYBOARD'.

You must ensure that each **HardwareID** element that is specified in the metadata package has a value that accurately correlates to the [hardware ID](#) that the physical device reports. For example, if a hardware ID is reused across a family of physical devices, the device metadata package that specifies that **HardwareID** element value is associated with the entire family of devices.

This section includes the following topics on best practices for specifying [hardware IDs](#) for certain types of devices:

[Specifying Hardware IDs for a Bluetooth Device](#)

[Specifying Hardware IDs for a Computer](#)

[Specifying Hardware IDs for a Multifunction Device](#)

For more information about the format requirements of the **HardwareID** XML element, see [HardwareID](#).

# Specifying Hardware IDs for a Bluetooth Device

12/21/2018 • 2 minutes to read • [Edit Online](#)

To specify [hardware IDs](#) for a Bluetooth device within a device metadata package, the device must support the Device Identification (DID) Profile. Otherwise, the operating system cannot select and load the most appropriate device metadata package for the Bluetooth device. We highly recommend that all Bluetooth devices support the DID profile.

For more information about the DID Profile, refer to the [Bluetooth Device Identification Profile specification version 1.3](#).

For information about supporting Bluetooth Low Energy (LE) Device IDs, refer to the [Device Information Service version 1.1](#).

# Specifying Hardware IDs for a Computer

11/2/2020 • 6 minutes to read • [Edit Online](#)

Devices and Printers recognizes the computer as a [device container](#). As a result, the computer can be identified within a device metadata package by using a [HardwareID](#) XML element that specifies a unique [hardware ID](#) value. This hardware ID value for the computer (sometimes referred to as a computer hardware ID, or CHID) can specify a combination of the System Management BIOS (SMBIOS) field data.

Unlike [hardware IDs](#) for other device containers, the hardware ID for the computer is generated by Windows every time the system boots. The hardware IDs for a computer can be generated by running the ComputerHardwareIds tool (ComputerHardwareIDs.exe), which is included in the Windows Driver Kit (WDK) for Windows 7, Windows 8 and Windows 8.1. Beginning with Windows 10, the ComputerHardwareIds tool is included in the Software Development Kit (SDK).

The ComputerHardwareIds tool generates a set of hardware IDs for the computer that is based on information from the fields in the system's System Management BIOS (SMBIOS). The following table describes these SMBIOS fields.

FIELD NAME	STRUCTURE NAME AND TYPE	SMBIOS SPECIFICATION VERSION	OFFSET	LENGTH	VALUE	DESCRIPTION
Manufacturer	System Information (Type 1)	2.0+	04h	BYTE	STRING	The index of a null-terminated string within the dmiStrucBuffer array. This string specifies the name of the computer manufacturer.

FIELD NAME	STRUCTURE NAME AND TYPE	SMBIOS SPECIFICATION VERSION	OFFSET	LENGTH	VALUE	DESCRIPTION
Family	System Information (Type 1)	2.4+	1Ah	BYTE	STRING	<p>The index of a null-terminated string within the dmiStrucBuffer array. This string specifies the family to which a particular computer belongs. A family refers to a set of computers that are similar but not identical from a hardware or software point of view.</p> <p>Typically a family is composed of different computer models, which have different configurations and pricing points.</p> <p>Computers in the same family often have similar branding and cosmetic features.</p>
Product Name	System Information (Type 1)	2.0+	05h	BYTE	STRING	<p>The index of a null-terminated string within the dmiStrucBuffer array. This string specifies the product name of the computer.</p>

FIELD NAME	STRUCTURE NAME AND TYPE	SMBIOS SPECIFICATION VERSION	OFFSET	LENGTH	VALUE	DESCRIPTION
Vendor	BIOS Information (Type 0)	2.0+	04h	BYTE	STRING	The index of a null-terminated string within the dmiStrucBuffer array. This string specifies the name of the BIOS vendor.
BIOS Version	BIOS Information (Type 0)	2.+0	05h	BYTE	STRING	The index of a null-terminated string within the dmiStrucBuffer array. This string can contain information about the processor core and OEM version.
System BIOS Major Release	BIOS Information (Type 0)	2.4+	14h	BYTE	Varies.	The major release of the system BIOS.
System BIOS Minor Release	BIOS Information (Type 0)	2.4+	15h	BYTE	Varies	The minor release of the system BIOS.
Enclosure type	System Enclosure (Type 3)	2.0+	05h	BYTE	Varies	The system enclosure or chassis types.
SKU Number	SKU Number (Type 1)	2.4+	19h	BYTE	STRING	The identification of a particular computer configuration for sale.

FIELD NAME	STRUCTURE NAME AND TYPE	SMBIOS SPECIFICATION VERSION	OFFSET	LENGTH	VALUE	DESCRIPTION
Baseboard Manufacturer	Manufacturer (Type 2)		04h	BYTE	STRING	Number of null-terminated string. This string identifies the Manufacturer of the Baseboard, where the Baseboard – Board Type is 0Ah (Motherboard)
Baseboard Product	Product (Type 2)		05h	BYTE	STRING	Number of null-terminated string. This string identifies the Product name of the Baseboard, where the Baseboard – Board Type is 0Ah (Motherboard)

For more information about the *dmiStrucBuffer* array and the SMBIOS fields, see the [System Management BIOS \(SMBIOS\)](#) specification on the Distributed Management Task Force (DMTF) website.

When the ComputerHardwareIDs tool runs, it creates unique hardware IDs from the SMBIOS information. Each hardware ID is a *GUID* and is created by concatenating the values from the SMBIOS fields.

The following tables show the SMBIOS fields used to form each hardware ID in Windows 7, Windows 8, Windows 8.1, and Windows 10.

**Important** Each Computer HardwareID is only generated if each individual SMBIOS field used to generate the HardwareID is populated in the SMBIOS data for the system.

HWID	WINDOWS 7
HardwareID-0	Manufacturer + Family + Product Name + Vendor + BIOS Version + System BIOS Major Release + System BIOS Minor Release
HardwareID-1	Manufacturer + Product Name + BIOS Vendor + BIOS Version + System BIOS Major Release + System BIOS Minor Release
HardwareID-2	Manufacturer + Family + ProductName
HardwareID-3	Manufacturer + ProductName

<b>HWID</b>	<b>WINDOWS 7</b>
HardwareID-4	Manufacturer + Family
HardwareID-5	Manufacturer + Enclosure Type
HardwareID-6	Manufacturer
<b>HWID</b>	<b>WINDOWS 8, WINDOWS 8.1</b>
HardwareID-0	Manufacturer + Family + Product Name + SKU Number + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-1	Manufacturer + Family + Product Name + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-2	Manufacturer + Product Name + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-3	Manufacturer + Family + ProductName + SKU Number
HardwareID-4	Manufacturer + Family + ProductName
HardwareID-5	Manufacturer + SKU Number
HardwareID-6	Manufacturer + ProductName
HardwareID-7	Manufacturer + Family
HardwareID-8	Manufacturer + Enclosure Type
HardwareID-9	Manufacturer
<b>HWID</b>	<b>WINDOWS 10</b>
HardwareID-0	Manufacturer + Family + Product Name + SKU Number + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-1	Manufacturer + Family + Product Name + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-2	Manufacturer + Product Name + BIOS Vendor + BIOS Version + BIOS Major Release + BIOS Minor Release
HardwareID-3	Manufacturer + Family + Product Name + SKU Number + Baseboard Manufacturer + Baseboard Product
HardwareID-4	Manufacturer + Family + Product Name + SKU Number
HardwareID-5	Manufacturer + Family + Product Name

HWID	WINDOWS 10
HardwareID-6	Manufacturer + SKU Number + Baseboard Manufacturer + Baseboard Product
HardwareID-7	Manufacturer + SKU Number
HardwareID-8	Manufacturer + Product Name + Baseboard Manufacturer + Baseboard Product
HardwareID-9	Manufacturer + Product Name
HardwareID-10	Manufacturer + Family + Baseboard Manufacturer + Baseboard Product
HardwareID-11	Manufacturer + Family
HardwareID-12	Manufacturer + Enclosure Type
HardwareID-13	Manufacturer + Baseboard Manufacturer + Baseboard Product
HardwareID-14	Manufacturer

Each hardware ID string is converted into a GUID by using the SHA-1 hashing algorithm.

### Using Computer HardwareIDs with PC Device Metadata packages

For Windows 7 systems, we highly recommend that vendors do the following when selecting a [hardware ID](#) value to use as the [HardwareID](#) XML element value for the computer.

- Use [HardwareID-3](#) or [HardwareID-4](#) as the first choice if the device metadata package matches a computer that has a specific make, family, and model. This allows a metadata package to match the specified computer, which provides the most precise metadata for the computer.
- Use [HardwareID-5](#), as the second choice if the device metadata package covers the entire family of computers. In this case, the computer family is unique and is not branded with more than one product line.
- Use [HardwareID-6](#) or [HardwareID-7](#) as the third choice if the device metadata package covers all of your computers or those computers with a specific enclosure type.

**Note** For Windows 7 PC Device Metadata, do not use [HardwareID-1](#) or [HardwareID-2](#) for the computer's hardware ID. [Hardware ID-1](#) or [HardwareID-2](#) is reserved for future use.

**Note** For Windows 8 PC Device Metadata, we strongly recommend that vendors not use [HardwareID-1](#), [HardwareID-2](#), [HardwareID-3](#) for the computer's hardware ID. [HardwareID-1](#), [HardwareID-2](#), [HardwareID-3](#) are reserved for future use. Instead, vendors can use [HardwareID-4](#), [HardwareID-5](#), [HardwareID-6](#), [HardwareID-7](#), [HardwareID-8](#), [HardwareID-9](#), and [HardwareID-10](#).

To specify that the hardware ID is for a computer device container, use the following rules:

- Delimit the hardware ID string with '{' and '}' characters.
- Add the prefix 'ComputerMetadata\' in front of the hardware ID string.

The following is an example of a [HardwareID](#) XML element for the computer:

DOIID:ComputerMetadata\{c20d5449-511e-4cb5-902a-a541239322aa}

For more information about the format requirements of the **HardwareID** XML element, see [HardwareID](#).

## Related topics

[Windows 10 Driver Publishing Workflow](#)

# Specifying Hardware IDs for a Multifunction Device

12/1/2020 • 2 minutes to read • [Edit Online](#)

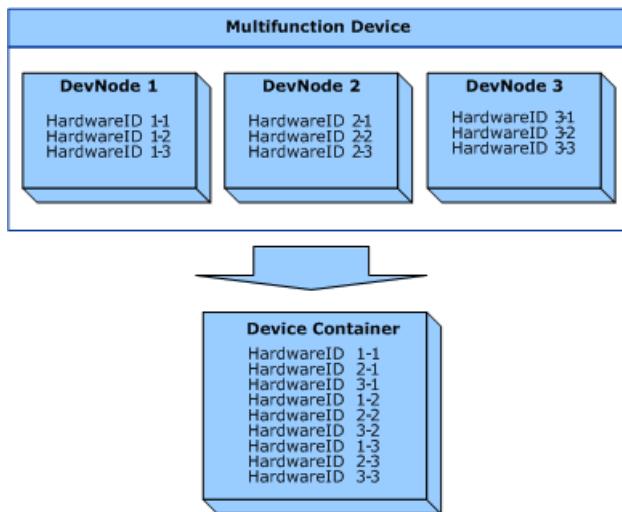
You can specify more than one **HardwareID** element for a physical device. This is done by specifying multiple **HardwareID** element values within the parent **HardwareIDList** element. Each value must specify a unique **hardware ID** for the device.

For example, consider a single-function USB printer from the company Contoso, Ltd. The following **HardwareID** elements can be used to define the device:

```
<HardwareIDList>
  <HardwareID>DOID:USB\VID_1234&PID_1234&REV_0000</HardwareID>
  <HardwareID>DOID:USB\VID_1234&PID_1234</HardwareID>
  <HardwareID>DOID:USBPRINT\Contoso_Ltd_Co9999/HardwareID>
</HardwareIDList>
```

If the device is a multifunction device, a device container combines all the **hardware IDs** from the device nodes (*devnodes*) for each hardware function on the device. For more information about device containers and container IDs, see [Container IDs](#).

The following figure shows the relationship between a multifunction device's devnodes and device container.



Depending on your multifunction device, you can decide which **hardware ID** values are specified by using separate **HardwareID** elements in the **HardwareIDList** element. Multiple hardware IDs can be specified in any order in the **HardwareIDList** element. However, you should be aware of the following points:

- Within a given devnode, the ranking of hardware IDs by the operating system is deterministic. For example, in the previous figure, *HardwareID1-1* is always ranked higher than *HardwareID1-2* and *HardwareID1-3*.
- The operating system does not consistently rank a hardware ID from one devnode higher than a hardware ID from another devnode. For example, in the previous figure, the operating system might not always rank *HardwareID1-1* higher than *HardwareID2-1*.

Therefore, make sure that your metadata package does not rely on the order or ranking of **hardware IDs** across the devnodes for a device. You should use all relevant hardware IDs for your multifunction device in the **HardwareIDList** element of your device metadata package. This guarantees that the operating system selects your metadata package regardless of the ranking of hardware IDs.

Based on the *devnode* topology shown in the previous figure, consider the following suggestions:

- Specify *HardwareID1-1*, *HardwareID2-1*, and *Hardware ID3-1* in a new device metadata package if you have already published another metadata package that specifies only *HardwareID2-1*.

If the operating system ranks *HardwareID2-1* higher than *HardwareID1-1* and finds that *HardwareID2-1* is specified in both the old and new metadata package, the operating system selects the metadata package based on the value of the **LastModifiedDate** XML element. In this case, the operating system selects the new metadata package.

- If your new metadata package lists only *HardwareID1-1*, the operating system will not select the new package if *HardwareID2-1* is ranked higher than *HardwareID1-1*.

For more information about metadata package selection and ranking, see [How the DMRC Selects a Device Metadata Package](#).

# Best Practices for Specifying Model IDs

11/2/2020 • 2 minutes to read • [Edit Online](#)

Model IDs are based on the business definition or stock keeping unit (SKU) of the physical device. Each model ID must be unique for all makes and models of the physical device.

The following list describes the differences between hardware IDs and model IDs for a physical device:

- **Hardware IDs** are specified by using one or more **HardwareID** XML elements within the **HardwareIDList** XML element. Each **HardwareID** value specifies a hardware function based on a bus-specific value. Hardware IDs could be used to map device drivers to device instances.

For example, two devices with the same hardware ID share a functional interface that is used by the same driver.

- Model IDs are specified by using one or more **ModelID** XML elements within the **ModelIDList** XML element. Model IDs allow the original equipment manufacturer (OEM) or independent hardware vendor (IHV) to uniquely identify the physical device independent of bus or interface technologies.

For example, two devices with different model IDs might have the same hardware IDs for their components.

- **Hardware IDs** are used to map device metadata packages to device instances on a specific bus or interface.
- Model IDs are used to map device metadata packages to physical devices, regardless of how the device is connected to the computer.

The **ModelIDList** XML element is required only if the **HardwareIDList** element is not specified in the **PackageInfo** XML data. If it is specified, the **ModelIDList** element must contain one or more **ModelID** elements to specify the unique model ID for each function that the device supports.

If the **PackageInfo** XML data contains the **HardwareIDList** and **ModelIDList** elements, the operating system uses the following rules when it determines whether a device is specified by a device metadata package:

- If the device has a model ID, the operating system does not search for a match in the **HardwareIDList** element.
- Otherwise, the operating system searches the **HardwareIDList** element for a match of the hardware ID for the device.

If your device metadata package supports multiple device models or model IDs, you can specify a **ModelID** element for each device model.

The following is an example of a **ModelIDList** element that has multiple **ModelID** elements:

```
<ModelIDList>
<ModelID>825AAB98-18EE-4FE2-9472-197D1D00FE31</ModelID>
<ModelID>23F64715-AC4A-4DC4-B554-C8D56E43FE8B</ModelID>
</ModelIDList>
```

For more information about the format requirements of the **ModelID** XML element, see [ModelID](#).

# Best Practices for Testing the Download of Device Metadata Packages

12/5/2018 • 2 minutes to read • [Edit Online](#)

Because of how the device metadata retrieval client ([DMRC](#)) caches metadata packages, a delay can occur between the time when a device metadata package is available on the Windows Metadata and Internet Services ([WMIS](#)) server and the time when the DMRC downloads the metadata package to a client system. To test the download of a device metadata package, you can force a download in one of the following ways:

- Delete the folders in the [device metadata cache](#). This cache is located in the following directory:

Windows 7:

```
%LOCALAPPDATA%\Microsoft\Device Metadata\
```

Windows 8:

```
%PROGRAMDATA%\Microsoft\Windows\DeviceMetadataCache\
```

Deleting these folders resets the value of **LastCheckedDate** and forces the DMRC to query the WMIS server for all devices.

- Set the **CheckBackMDRetrieved** and **CheckBackMDNotRetrieved** registry keys to 0. When these values are zero, the DMRC immediately queries the WMIS server for a target device.

Be aware that the WMIS server overwrites these values every time that the DMRC receives a response from WMIS. Therefore, these parameters can change if the DMRC receives a response for any other device before it queries the WMIS server for your target device.

**Note** You must make these changes only when you test metadata packages. You must not provide end-users with any tools that change the registry values that are used by DMRC.

# SetupAPI

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Setup application programming interface (SetupAPI) is a system component that provides two sets of functions:

- [General setup functions](#)
- [Device installation functions](#)

Device installation software can use these functions to perform custom operations in *class installers*, *co-installers*, and *device installation applications*.

For device installation applications, Driver Install Frameworks (DIFx) provides high-level tools that abstract the low-level SetupAPI operations that install Plug and Play (PnP) function drivers and manage the association between application software and the drivers. If the DIFx tools provide the functionality that an installation application requires to install PnP drivers and application software for devices, the installation application should use the DIFx tools instead of directly calling SetupAPI functions. However, co-installers and class installers are Microsoft Win32 DLLs that assist the default installation operation by performing custom operations for a device or all devices in a [device setup class](#). These operations typically require direct calls to Win32 functions and SetupAPI functions.

This section contains the following topics, which provide general information about how to use the [general Setup functions](#) and [device installation functions](#) that are provided by SetupAPI:

[Using General Setup Functions](#)

[Using Device Installation Functions](#)

[Guidelines for Using SetupAPI](#)

**Note** This section describes only a subset of the Setup functions in SetupAPI. For more information about this API, see [Setup API](#).

# Using General Setup Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section summarizes the general Setup functions. *Device installation applications* can use these functions to do the following:

- Read and process INF files.
- Determine the amount of free space that is required on the installation's target system.
- Move files from installation source media to media on the installation's target system, while requesting user intervention as needed.
- Create a log of files moved during an installation.
- Write log entries to the [SetupAPI text logs](#).

Installation software typically uses these functions together with [device installation functions](#) and [PnP configuration manager functions](#).

The general Setup functions listed in this section are described in detail in the Microsoft Windows SDK documentation.

This section includes the following topics:

[INF File Processing Functions](#)

[Disk Prompting and Error Handling Functions](#)

[File Queuing Functions](#)

[Default Queue Callback Routine Functions](#)

[Cabinet File Function](#)

[Disk-Space List Functions](#)

[MRU Source List Functions](#)

[File Log Functions](#)

[User Interface Functions](#)

[SetupAPI Logging Functions](#)

# INF File Processing Functions

11/2/2020 • 3 minutes to read • [Edit Online](#)

The INF file processing functions provide setup and installation functionality that includes the following:

- Opening and closing an INF file.
- Retrieving information about an INF file.
- Retrieving information about source files and target directories for copy operations.
- Performing the installation actions specified in an INF file section.

The following table lists the functions that are used for processing INF files. For detailed function descriptions, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">InstallHinfSection</a>	Executes a specified section in a specified INF file.
<a href="#">SetupCloseInfFile</a>	Frees resources and closes the INF handle.
<a href="#">SetupCopyOEMInf</a>	Copies a file into %SystemRoot%\Inf
<a href="#">SetupDecompressOrCopyFile</a>	Copies a file and, if necessary, decompresses it.
<a href="#">SetupFindFirstLine</a>	Finds a pointer to the first line in a section of an INF file or, if a key is specified, the first line that matches the key.
<a href="#">SetupFindNextLine</a>	Returns a pointer to the next line in an INF file section.
<a href="#">SetupFindNextMatchLine</a>	Returns a pointer to the next line in an INF file section or, if a key is specified, the next line that matches the key.
<a href="#">SetupGetBinaryField</a>	Retrieves binary data from a field in a specified line, in an INF file.
<a href="#">SetupGetFieldCount</a>	Returns the number of fields in a line.
<a href="#">SetupGetFileCompressionInfo</a>	Retrieves file compression information from an INF file.
<a href="#">SetupGetInfDriverStoreLocation</a>	Retrieves the fully qualified file name (directory path and file name) of an INF file in the <a href="#">driver store</a> that corresponds to a specified INF file in the system INF file directory or a specified INF file in the driver store.

FUNCTION	DESCRIPTION
<a href="#">SetupGetInfFileList</a>	Returns a list of the INF files in a specified directory.
<a href="#">SetupGetInfInformation</a>	Returns information about an INF file.
<a href="#">SetupGetIntField</a>	Obtains the integer value of a specified field in a specified line, in an INF file.
<a href="#">SetupGetInfPublishedName</a>	Retrieves the fully qualified name (directory path and file name) of an INF file in the system INF file directory that corresponds to a specified INF file in the system INF file directory or a specified INF file in the <a href="#">driver store</a> .
<a href="#">SetupGetLineByIndex</a>	Returns a pointer to the line associated with a specified index value in a specified section.
<a href="#">SetupGetLineCount</a>	Returns the number of lines in the specified section.
<a href="#">SetupGetLineText</a>	Retrieves the contents of a specified line from an INF file.
<a href="#">SetupGetMultiSzField</a>	Returns multiple strings, starting at a specified field in a line.
<a href="#">SetupGetSourceFileLocation</a>	Returns the location of a source file that is listed in an INF file.
<a href="#">SetupGetSourceFileSize</a>	Returns the size of a specified file or a set of files that are listed in a specified section of an INF file.
<a href="#">SetupGetSourceInfo</a>	Retrieves the path, tag file, or description for a source.
<a href="#">SetupGetStringField</a>	Retrieves string data from a field in a specified line, in an INF file.
<a href="#">SetupGetTargetPath</a>	Determines the target path for the files that are listed in a specified INF file section.
<a href="#">SetupInstallFile</a>	Installs a specified file into a specific target directory.
<a href="#">SetupInstallFileEx</a>	Installs a specified file into a specific target directory. The installation is postponed if an existing version of the file is in use.

FUNCTION	DESCRIPTION
<a href="#">SetupInstallFilesFromInfSection</a>	Queues the files in a specified INF file section for copying. (Same as <a href="#">SetupQueueCopySection</a> .)
<a href="#">SetupInstallFromInfSection</a>	Performs the directives specified in an INF <i>DD\Install</i> section.
<a href="#">SetupInstallServicesFromInfSection</a>	Performs service installation and deletion operations as specified in an INF <i>DD\Install.Services</i> section.
<a href="#">SetupOpenAppendInfFile</a>	Opens an INF file and appends it to an existing INF handle.
<a href="#">SetupOpenInfFile</a>	Opens an INF file and returns a handle to it.
<a href="#">SetupOpenMasterInf</a>	Opens the master INF file that contains file and layout information for files that were included with the default installation of the operating system.
<a href="#">SetupQueryInfFileInformation</a>	Returns the name of one of the constituent INF files of a specified INF file.
<a href="#">SetupQueryInfVersionInformation</a>	Returns the version number of one of the constituent INF files of a specified INF file.
<a href="#">SetupSetDirectoryId</a>	Assigns a directory ID (DIRID) to a specified directory.
<a href="#">SetupUninstallOEMInf</a>	Uninstalls a specified INF file, and deletes the associated .pnf and .cat files, if they exist.
<a href="#">SetupVerifyInfFile</a>	Verifies that a digitally-signed INF file has not been modified. (Windows XP and later.)

# Disk Prompting and Error Handling Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the general Setup functions to prompt the user to insert new media, or to handle errors that arise when files are being copied, renamed, or deleted.

The following table lists functions that provide dialog boxes for requesting installation media and reporting errors. For detailed function descriptions, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupCopyError</a>	Generates a dialog box that informs the user of a copy error.
<a href="#">SetupDeleteError</a>	Generates a dialog box that informs the user of a delete error.
<a href="#">SetupPromptForDisk</a>	Generates a dialog box that prompts the user for an installation medium or source file location.
<a href="#">SetupRenameError</a>	Generates a dialog box that informs the user of a rename error.

# File Queuing Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Using the general Setup functions, you can queue files for various operations. File queues can be established for copying, renaming, and deleting files. Typically, an application queues all the file operations necessary for an entire installation and then "commits" the queue so the operations are performed in a single batch.

The following table provides a summary of file queuing functions. For detailed function descriptions, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupCloseFileQueue</a>	Destroys a file queue together with any uncommitted file operations.
<a href="#">SetupCommitFileQueue</a>	Commits (performs) all queued operations.
<a href="#">SetupOpenFileQueue</a>	Creates and returns a handle to a file queue.
<a href="#">SetupPromptReboot</a>	Prompts the user to restart his or her computer, if necessary.
<a href="#">SetupQueueCopy</a>	Queues a specified file for copying.
<a href="#">SetupQueueCopyIndirect</a>	Queues a specified file for copying, and provides file location and security information.
<a href="#">SetupQueueCopySection</a>	Queues the files in a specified INF file section for copying.
<a href="#">SetupQueueDefaultCopy</a>	Queues a specified file for copying, using default source and destination settings contained in the INF file.
<a href="#">SetupQueueDelete</a>	Queues a specified file for deletion.
<a href="#">SetupQueueDeleteSection</a>	Queues the files in an INF file section for deletion.
<a href="#">SetupQueueRename</a>	Queues a specified file for renaming.
<a href="#">SetupQueueRenameSection</a>	Queues the files in an INF section for renaming.
<a href="#">SetupScanFileQueue</a>	Scans a file queue and performs a specified operation on each queue entry.

FUNCTION	DESCRIPTION
<a href="#">SetupSetPlatformPathOverride</a>	Sets the value that is used for overriding the default platform-specific source path.

# Default Queue Callback Routine Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

If you associate a callback routine with a file queue, the callback routine will be called every time that the system performs one of the queued file operations. Typically, you can use the default queue callback routine, [SetupDefaultQueueCallback](#), to handle these notifications.

The following table lists functions associated with the default queue callback routine. For detailed function descriptions, and for more information about how to use callback routines with file queues, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupDefaultQueueCallback</a>	Handles notifications sent by the system when queued file operations are performed.
<a href="#">SetupInitDefaultQueueCallback</a>	Initializes context information that is needed by <a href="#">SetupDefaultQueueCallback</a> .
<a href="#">SetupInitDefaultQueueCallbackEx</a>	Initializes context information that is needed by <a href="#">SetupDefaultQueueCallback</a> , and provides a separate window for displaying progress messages.
<a href="#">SetupTermDefaultQueueCallback</a>	Notifies the system that the <i>device installation application</i> will not commit any additional file queue operations.

# Cabinet File Function

11/2/2020 • 2 minutes to read • [Edit Online](#)

A cabinet (CAB) file is a single file, usually with a *.cab* extension, that contains several compressed files as a file library. CAB files are used to organize the installation files that will be copied to the user's system. A compressed file can be spread over several CAB files.

The following function is used with CAB files. For a detailed function description, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupIterateCabinet</a>	Sends a notification to a callback function for each file that is stored in a CAB file.

# Disk-Space List Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Disk-space list functions are used to create and modify disk-space lists. These lists can be used to calculate the total disk space that is required to handle the files that will be copied or deleted during the installation procedure.

The following table lists the functions that can be used to manipulate disk-space lists. For detailed function descriptions, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupAddInstallSectionToDiskSpaceList</a>	Searches for <b>CopyFile</b> and <b>DelFile</b> directives in a <i>DD\Install</i> section of an INF file, then adds the file operations specified in those sections to a disk-space list.
<a href="#">SetupAddSectionToDiskSpaceList</a>	Adds to a disk-space list all the file copy or delete operations that are listed in a specified section of an INF file.
<a href="#">SetupAddToDiskSpaceList</a>	Adds a single delete or copy operation to a disk-space list.
<a href="#">SetupCreateDiskSpaceList</a>	Creates a disk-space list.
<a href="#">SetupDestroyDiskSpaceList</a>	Destroys a disk-space list and releases the resources allocated to it.
<a href="#">SetupQueryDrivesInDiskSpaceList</a>	Fills a caller-supplied buffer with a list of the drives referenced by the file operations that are listed in the disk-space list.
<a href="#">SetupQuerySpaceRequiredOnDrive</a>	Examines a disk-space list to determine the space that is required to perform all the file operations that are listed for a particular drive.
<a href="#">SetupRemoveFromDiskSpaceList</a>	Removes a file copy or delete operation from a disk-space list.
<a href="#">SetupRemoveInstallSectionFromDiskSpaceList</a>	Searches for <b>CopyFiles</b> and <b>DelFiles</b> directives in a <i>DD\Install</i> section of an INF file, and removes the file operations specified in those sections from a disk-space list.
<a href="#">SetupRemoveSectionFromDiskSpaceList</a>	Removes from a disk-space list the file copy or delete operations that are listed in a specified section of an INF file.

# MRU Source List Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Most recently used (MRU) source lists are resident on the user's computer and contain information about source paths used in previous installations. This information can be used when prompting the user for a source path.

The *device installation application* can access a user-specific source list and, if the application has administrator privilege, the system-wide source list. The device installation application can also create a temporary source list that is discarded when the device installation application exits. By calling **SetupSetSourceList**, the device installation application identifies which source list it will use during the installation.

The following table lists the functions that can be used to manipulate source lists. For detailed function descriptions, see the Microsoft Windows SDK documentation.

FUNCTION	DESCRIPTION
<a href="#">SetupAddToSourceList</a>	Adds an entry to a source list.
<a href="#">SetupCancelTemporarySourceList</a>	Cancels use of a temporary list.
<a href="#">SetupFreeSourceList</a>	Frees resources allocated by a previous call to <a href="#">SetupSetSourceList</a> .
<a href="#">SetupQuerySourceList</a>	Queries the current list of installation sources.
<a href="#">SetupRemoveFromSourceList</a>	Removes an entry from an installation source list.
<a href="#">SetupSetSourceList</a>	Sets the installation source list to the system MRU list, the user MRU list, or a temporary list.

# File Log Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use a log file to record information about the files copied to a system during an installation. The log file can be either the system log or your own installation log file.

The following table lists the functions that can be used to manipulate log files. For more information about function descriptions, see the [Microsoft Windows SDK documentation](#).

FUNCTION	DESCRIPTION
<a href="#">SetupInitializeLogFile</a>	Initializes a log file for use.
<a href="#">SetupLogError</a>	Writes an error message to a log file. (It should be used only during the installation of the operating system.)
<a href="#">SetupLogFile</a>	Adds an entry to the log file.
<a href="#">SetupQueryLogFile</a>	Retrieves information from a log file.
<a href="#">SetupRemoveLogFileEntry</a>	Removes an entry from a log file.
<a href="#">SetupTerminateLogFile</a>	Releases resources allocated to a log file.

# User Interface Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the following general Setup functions in class installers and co-installers to determine whether the current process can interact with a user.

FUNCTION	DESCRIPTION
<a href="#">SetupGetNonInteractiveMode</a>	Returns the value of a SetupAPI non-interactive flag that indicates whether the caller's process can interact with a user through user interface components, such as dialog boxes.
<a href="#">SetupSetNonInteractiveMode</a>	Sets a non-interactive SetupAPI flag that determines whether SetupAPI can interact with a user in the caller's context.

# SetupAPI Logging Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, Plug and Play (PnP) device installation applications, class installers, and co-installers can use the following functions to write log entries to the [SetupAPI text logs](#).

FUNCTION	DESCRIPTION
<a href="#">SetupGetThreadLogToken</a>	Retrieves the <a href="#">log token</a> for the thread that called <a href="#">SetupGetThreadLogToken</a> .
<a href="#">SetupSetThreadLogToken</a>	Sets the log token for the thread that called <a href="#">SetupSetThreadLogToken</a> .
<a href="#">SetupWriteTextLog</a>	Writes a log entry in a <a href="#">SetupAPI text log</a> .
<a href="#">SetupWriteTextLogError</a>	Writes information about a SetupAPI-specific error or a Win32 error in SetupAPI text log.
<a href="#">SetupWriteTextLogInfLine</a>	Writes a log entry in a SetupAPI text log that contains the text of a specified INF file line.

# Using Device Installation Functions

11/2/2020 • 10 minutes to read • [Edit Online](#)

This section summarizes the [device installation functions](#). By using the device installation functions, the installation software can perform the following types of operations:

- Install drivers
- Handle DIF codes.
- Manage device information sets.
- Manage driver lists.
- Manage device interfaces.
- Manage icons and other bitmaps.

To perform device installation operations that are not supported by the SetupAPI functions described in this section, call the appropriate [general Setup functions](#) or [PnP Configuration Manager functions](#) (CM\_XXX functions).

The following tables provide summaries of the following types of functions:

[Driver Installation Functions](#)

[Device Information Functions](#)

[Driver Information Functions](#)

[Driver Selection Functions](#)

[Device Installation Handlers](#)

[Device Installation Customization Functions](#)

[Setup Class Functions](#)

[Bitmap and Icon Functions](#)

[Device Interface Functions](#)

[Device Property Functions \(Windows Vista and Later\)](#)

[Registry Functions](#)

[Other Functions](#)

## **Driver Installation Functions**

<a href="#">DiInstallDevice</a>	Installs a specified driver that is preinstalled in the <a href="#">driver store</a> on a PnP device that is present in the system. (Windows Vista and later versions of Windows)
<a href="#">DiInstallDriver</a>	Preinstalls a driver in the driver store and then installs the driver on matching PnP devices that are present in the system. (Windows Vista and later versions of Windows)

<a href="#">DiRollbackDriver</a>	Rolls back the driver that is installed on a specified device to the backup driver set for the device. (Windows Vista and later versions of Windows)
<a href="#">DiShowUpdateDevice</a>	Displays the Hardware Update wizard for a specified device. (Windows Vista and later versions of Windows)
<a href="#">DiUninstallDevice</a>	Uninstalls a device and removes its device node ( <i>devnode</i> ) from the system. (Windows 7 and later versions of Windows)
<a href="#">InstallSelectedDriver</a>	Installs a selected driver on a selected device.
<a href="#">UpdateDriverForPlugAndPlayDevices</a>	Updates the function driver that is installed for matching PnP devices that are present in the system.

## Device Information Functions

<a href="#">SetupDiCreateDeviceInfoList</a>	Creates an empty <a href="#">device information set</a> . This set can be associated with a class GUID.
<a href="#">SetupDiCreateDeviceInfoListEx</a>	Creates an empty device information set. This set can be associated with a class GUID and can be for devices on a remote computer.
<a href="#">SetupDiCreateDeviceInfo</a>	Creates a new device information element and adds it as a new member to the specified device information set.
<a href="#">SetupDiOpenDeviceInfo</a>	Retrieves information about an existing device instance and adds it to the specified device information set.
<a href="#">SetupDiEnumDeviceInfo</a>	Returns a context structure for a device information element of a device information set.
<a href="#">SetupDiGetDeviceInstanceId</a>	Retrieves the device instance ID associated with a device information element.
<a href="#">SetupDiGetDeviceInfoListClass</a>	Retrieves the class GUID associated with a device information set if it has an associated class.
<a href="#">SetupDiGetDeviceInfoListDetail</a>	Retrieves information associated with a device information set including the class GUID, remote computer handle, and remote computer name.

<a href="#">SetupDiGetClassDevPropertySheets</a>	Retrieves handles to the property sheets of a specified device information element or of the <a href="#">device setup class</a> of a specified device information set.
<a href="#">SetupDiGetClassDevs</a>	Returns a device information set that contains all devices of a specified class.
<a href="#">SetupDiGetClassDevsEx</a>	Returns a device information set that contains all devices of a specified class on a local or remote computer.
<a href="#">SetupDiSetSelectedDevice</a>	Sets the specified device information element to be the currently-selected member of a device information set. This function is typically used by an installation wizard.
<a href="#">SetupDiGetSelectedDevice</a>	Retrieves the currently-selected device for the specified device information set.
<a href="#">SetupDiRegisterDeviceInfo</a>	Registers a newly created device instance with the Plug and Play manager.
<a href="#">SetupDiDeleteDeviceInfo</a>	Deletes a member from the specified device information set. This function does not delete the actual device.
<a href="#">SetupDiDestroyDeviceInfoList</a>	Destroys a device information set and frees all associated memory.

## Driver Information Functions

<a href="#">SetupDiBuildDriverInfoList</a>	Builds a list of drivers associated with a specified device instance or with the device information set's global class driver list.
<a href="#">SetupDiEnumDriverInfo</a>	Enumerates the members of a driver information list.
<a href="#">SetupDiGetDriverInfoDetail</a>	Retrieves detailed information for a specified driver information element.
<a href="#">SetupDiSetSelectedDriver</a>	Sets the specified member of a driver list as the currently selected-driver. It can also be used to reset the driver list so that there is no currently-selected driver.
<a href="#">SetupDiGetSelectedDriver</a>	Retrieves the member of a driver list that was selected as the driver to install.
<a href="#">SetupDiCancelDriverInfoSearch</a>	Cancels a driver list search that is currently underway in a different thread.

<a href="#">SetupDiDestroyDriverInfoList</a>	Destroys a driver information list.
--	-------------------------------------

## Driver Selection Functions

<a href="#">SetupDiAskForOEMDisk</a>	Displays a dialog that asks the user for the path of an OEM installation disk.
<a href="#">SetupDiSelectOEMDrv</a>	Selects a driver for a device by using an OEM path supplied by the user.
<a href="#">SetupDiSelectDevice</a>	Default handler for the DIF_SELECTDEVICE request.

## Device Installation Handlers

<a href="#">SetupDiCallClassInstaller</a>	Calls the appropriate class installer, and any registered co-installers, with the specified installation request.
<a href="#">SetupDiChangeState</a>	The default handler for the DIF_PROPERTYCHANGE request. It can be used to change the state of an installed device.
<a href="#">SetupDiRegisterCoDeviceInstallers</a>	Registers the device-specific co-installers listed in the INF file for the specified device. This function is the default handler for DIF_REGISTER_COINSTALLERS.
<a href="#">SetupDiInstallDevice</a>	The default handler for the DIF_INSTALLDEVICE request.
<a href="#">SetupDiInstallDriverFiles</a>	The default handler for the DIF_INSTALLDEVICEFILES request.
<a href="#">SetupDiInstallDeviceInterfaces</a>	The default handler for the DIF_INSTALLINTERFACES request. It installs the interfaces that are listed in a <i>DDInstallInterfaces</i> section of a device INF file.
<a href="#">SetupDiMoveDuplicateDevice</a>	This function is obsolete and cannot be used in any version of Microsoft Windows.
<a href="#">SetupDiRemoveDevice</a>	The default handler for the DIF_REMOVEDEVICE request.
<a href="#">SetupDiUnremoveDevice</a>	The default handler for the DIF_UNREMOVE request.
<a href="#">SetupDiRegisterDeviceInfo</a>	The default handler for the DIF_REGISTERDEVICE request.
<a href="#">SetupDiSelectDevice</a>	The default handler for the DIF_SELECTDEVICE request.

<a href="#">SetupDiSelectBestCompatDrv</a>	The default handler for the DIF_SELECTBESTCOMPATDRV request.
--	--

## Device Installation Customization Functions

<a href="#">SetupDiGetClassInstallParams</a>	Retrieves class install parameters for a device information set or a particular device information element.
<a href="#">SetupDiSetClassInstallParams</a>	Sets or clears class install parameters for a device information set or a particular device information element.
<a href="#">SetupDiGetDeviceInstallParams</a>	Retrieves device install parameters for a device information set or a particular device information element.
<a href="#">SetupDiSetDeviceInstallParams</a>	Sets device install parameters for a device information set or a particular device information element.
<a href="#">SetupDiGetDriverInstallParams</a>	Retrieves install parameters for the specified driver.
<a href="#">SetupDiSetDriverInstallParams</a>	Sets the installation parameters for the specified driver.

## Setup Class Functions

<a href="#">SetupDiBuildClassInfoList</a>	Returns a list of setup class GUIDs that includes every class installed on the system.
<a href="#">SetupDiBuildClassInfoListEx</a>	Returns a list of setup class GUIDs that includes every class installed on the local system or a remote system.
<a href="#">SetupDiGetClassDescription</a>	Retrieves the class description associated with the specified setup class GUID.
<a href="#">SetupDiGetClassDescriptionEx</a>	Retrieves the description of a setup class installed on a local or remote computer.
<a href="#">SetupDiGetINFClass</a>	Retrieves the class of a specified device INF file.
<a href="#">SetupDiClassGuidsFromName</a>	Retrieves the GUIDs associated with the specified class name. This list is built based on what classes are currently installed on the system.
<a href="#">SetupDiClassGuidsFromNameEx</a>	Retrieves the GUIDs associated with the specified class name. This resulting list contains the classes currently installed on a local or remote computer.

<a href="#">SetupDiClassNameFromGuid</a>	Retrieves the class name associated with the class GUID.
<a href="#">SetupDiClassNameFromGuidEx</a>	Retrieves the class name associated with a class GUID. The class can be installed on a local or remote computer.
<a href="#">SetupDiInstallClass</a>	Installs the <b>ClassInstall32</b> section of the specified INF file.
<a href="#">SetupDiInstallClassEx</a>	Installs a class installer or an interface class.
<a href="#">SetupDiOpenClassRegKey</a>	Opens the <a href="#">device setup class</a> registry key, or a specific subkey of the class.
<a href="#">SetupDiOpenClassRegKeyEx</a>	Opens the device setup class registry key, the device interface class registry key, or a specific subkey of the class. This function opens the specified key on the local computer or on a remote computer.

## Bitmap and Icon Functions

<a href="#">SetupDiGetClassImageList</a>	Builds an image list that contains bitmaps for every installed class and returns the list in a data structure.
<a href="#">SetupDiGetClassImageListEx</a>	Builds an image list of bitmaps for every class installed on a local or remote computer.
<a href="#">SetupDiGetClassImageIndex</a>	Retrieves the index within the class image list of a specified class.
<a href="#">SetupDiGetClassBitmapIndex</a>	Retrieves the index of the mini-icon supplied for the specified class.
<a href="#">SetupDiDrawMinIcon</a>	Draws the specified mini-icon at the location requested.
<a href="#">SetupDiLoadClassIcon</a>	Loads both the large and mini-icon for the specified class.
<a href="#">SetupDiLoadDeviceIcon</a>	Loads a device icon for a specified device. (Windows Vista and later versions of Windows)
<a href="#">SetupDiDestroyClassImageList</a>	Destroys a class image list.

## Device Interface Functions

<a href="#">SetupDiCreateDeviceInterface</a>	Registers device functionality (a device interface) for a device.
<a href="#">SetupDiOpenDeviceInterface</a>	Retrieves information about an existing device interface and adds it to the specified device information set.
<a href="#">SetupDiGetDeviceInterfaceAlias</a>	Returns an alias of the specified device interface.
<a href="#">SetupDiGetClassDevs</a>	Returns a device information set that contains all devices of a specified class.
<a href="#">SetupDiGetClassDevsEx</a>	Returns a device information set that contains all devices of a specified class on a local or remote computer.
<a href="#">SetupDiEnumDeviceInterfaces</a>	Returns a context structure for a device interface element of a device information set. Each call returns information about one device interface.  The function can be called repeatedly to obtain information about several interfaces exposed by one or more devices.
<a href="#">SetupDiGetDeviceInterfaceDetail</a>	Returns details about a particular device interface.
<a href="#">SetupDiCreateDeviceInterfaceRegKey</a>	Creates a registry subkey for storing information about a device interface instance and returns a handle to the key.
<a href="#">SetupDiOpenDeviceInterfaceRegKey</a>	Opens the registry subkey that is used by applications and drivers to store information that is specific to a device interface instance and returns a handle to the key.
<a href="#">SetupDiDeleteDeviceInterfaceRegKey</a>	Deletes the registry subkey that was used by applications and drivers to store information that is specific to a device interface instance.
<a href="#">SetupDiInstallDeviceInterfaces</a>	Is the default handler for the DIF_INSTALLINTERFACES request. It installs the interfaces that are listed in a <i>DDInstall.Interfaces</i> section of a device INF file.
<a href="#">SetupDiRemoveDeviceInterface</a>	Removes a registered device interface from the system.
<a href="#">SetupDiDeleteDeviceInterfaceData</a>	Deletes a device interface from a device information set.
<a href="#">SetupDiSetDeviceInterfaceDefault</a>	Sets a specified device interface as the default interface for a device class.

<a href="#">SetupDiInstallClassEx</a>	Installs a class installer or an interface class.
<a href="#">SetupDiOpenClassRegKeyEx</a>	Opens the <a href="#">device setup class</a> registry key, the device interface class registry key, or a specific subkey of the class. This function opens the specified key on the local computer or on a remote computer.

## Device Property Functions (Windows Vista and Later)

<a href="#">SetupDiGetClassProperty</a>	Retrieves a device property that is set for a device setup class or a device interface class.
<a href="#">SetupDiGetClassPropertyEx</a>	Retrieves a class property for a device setup class or a device interface class on a local or remote computer.
<a href="#">SetupDiGetClassPropertyParams</a>	Retrieves an array of the device property keys that represent the device properties that are set for a device setup class or a device interface class.
<a href="#">SetupDiGetClassPropertyParamsEx</a>	Retrieves an array of the device property keys that represent the device properties that are set for a device setup class or a device interface class on a local or a remote computer.
<a href="#">SetupDiGetDeviceInterfaceProperty</a>	Retrieves a device property that is set for a device interface.
<a href="#">SetupDiGetDeviceInterfacePropertyParams</a>	Retrieves an array of device property keys that represent the device properties that are set for a device interface.
<a href="#">SetupDiGetDeviceProperty</a>	Retrieves a device instance property.
<a href="#">SetupDiGetDevicePropertyParams</a>	Retrieves an array of the device property keys that represent the device properties that are set for a device instance.
<a href="#">SetupDiSetClassProperty</a>	Sets a class property for a device setup class or a device interface class.
<a href="#">SetupDiSetClassPropertyParams</a>	Sets a device property for a device setup class or a device interface class on a local or remote computer.
<a href="#">SetupDiSetDeviceInterfaceProperty</a>	Sets a device property of a device interface.
<a href="#">SetupDiSetDevicePropertyParams</a>	Sets a device instance property.

## Registry Functions

<a href="#">SetupDiCreateDevRegKey</a>	Creates a registry storage key for device-specific configuration information and returns a handle to the key.
<a href="#">SetupDiOpenDevRegKey</a>	Opens a registry storage key for device-specific configuration information and returns a handle to the key.
<a href="#">SetupDiDeleteDevRegKey</a>	Deletes the specified user-accessible registry key(s) associated with a device information element.
<a href="#">SetupDiOpenClassRegKey</a>	Opens the setup class registry key, or a specific subkey of the class.
<a href="#">SetupDiOpenClassRegKeyEx</a>	<p>Opens the device setup class registry key, the device interface class registry key, or a specific subkey of the class.</p> <p>This function opens the specified key on the local computer or on a remote computer.</p>
<a href="#">SetupDiCreateDeviceInterfaceRegKey</a>	Creates a nonvolatile registry subkey for storing information about a device interface instance and returns a handle to the key.
<a href="#">SetupDiOpenDeviceInterfaceRegKey</a>	Opens the registry subkey that is used by applications and drivers to store information that is specific to a device interface instance and returns a handle to the key.
<a href="#">SetupDiDeleteDeviceInterfaceRegKey</a>	Deletes the registry subkey that was used by applications and drivers to store information that is specific to a device interface instance.
<a href="#">SetupDiSetDeviceRegistryProperty</a>	Sets the specified Plug and Play device property.
<a href="#">SetupDiGetDeviceRegistryProperty</a>	Retrieves the specified Plug and Play device property.
<a href="#">SetupDiGetClassRegistryProperty</a>	Retrieves a specified device class property from the registry.
<a href="#">SetupDiSetClassRegistryProperty</a>	Sets a specified device class property in the registry.

## Other Functions

<a href="#">SetupDiGetActualModelsSection</a>	Retrieves the appropriate decorated <b>INF Models section</b> to use when installing a device from a device INF file.
---	---

<a href="#">SetupDiGetActualSectionToInstall</a>	Retrieves the appropriate <i>DD\Install</i> section to use when installing a device from a device INF file.
<a href="#">SetupDiGetActualSectionToInstallEx</a>	Retrieves the name of the INF <i>DD\Install</i> section that installs a device for a specified operating system and processor architecture.
<a href="#">SetupDiGetHwProfileFriendlyName</a>	Retrieves the friendly name associated with a hardware profile ID.
<a href="#">SetupDiGetHwProfileFriendlyNameEx</a>	Retrieves the friendly name associated with a hardware profile ID on a local or remote computer.
<a href="#">SetupDiGetHwProfileList</a>	Retrieves a list of all currently defined hardware profile IDs.
<a href="#">SetupDiGetHwProfileListEx</a>	Retrieves a list of all currently defined hardware profile IDs on a local or remote computer.
<a href="#">SetupDiRestartDevices</a>	Restarts a specified device or, if necessary, starts all devices that are operated by the same function and filter drivers as the specified device.

# Basic Installation Operations

11/2/2020 • 2 minutes to read • [Edit Online](#)

Installers can use the [general Setup functions](#) and [device installation functions](#) that are provided by SetupAPI to perform installation operations. These functions allow installers to search INF files for compatible drivers, to display driver choices to the user through selection dialog boxes, and to perform the actual driver installation.

Most of the device installation functions rely on information in the [SP\\_DEVINFO\\_DATA](#) structure to perform installation tasks. Each device is associated with an SP\_DEVINFO\_DATA structure. 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, to set the configuration of the device hardware, and to copy the driver files to the appropriate directories.

An installer might have 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 [device setup class](#) by using the [SetupDiInstallClass](#) function. This function installs the new setup class from an INF file that contains an [INF ClassInstall32 section](#).

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

# Functions that Simplify Driver Installation

11/2/2020 • 3 minutes to read • [Edit Online](#)

An installation application can use the following functions to simplify the installation of a PnP function driver.

## **DiInstallDevice (Windows Vista and later versions of Windows)**

The [DiInstallDevice](#) function installs a specific driver that is preinstalled in the [driver store](#) on a specific device present in the system.

An installation application should only use this function if both of the following are true:

- The application incorporates more than one device instance of the same type, that is, all the device instances have the same hardware IDs and compatible IDs.
- The application requires that device-instance-specific drivers be installed on the device instances.

Otherwise, an installation application should use [DiInstallDriver](#) or [UpdateDriverForPlugAndPlayDevices](#) to install the driver that is the best match for a device.

A caller can also call [DiInstallDevice](#) to do the following:

- Search for a preinstalled driver that is the best match to the device, and if one is not found, display the Found New Hardware wizard for the device.
- Suppress invoking finish-install pages and finish-install actions.
- Install a null driver on a specific device.
- Notify the caller whether a system restart is required to complete the installation.

## **DiInstallDriver (Windows Vista and later versions of Windows)**

The [DiInstallDriver](#) function preinstalls a [driver package](#) in the [driver store](#) and then installs the driver on all devices present in the system that have a hardware ID or a compatible ID that matches the driver package.

Calling [DiInstallDriver](#) or [UpdateDriverForPlugAndPlayDevices](#) is the simplest way for an installation application to install a new driver for a device. [DiInstallDriver](#) and [UpdateDriverForPlugAndPlayDevices](#) perform the same basic installation operations. However [UpdateDriverForPlugAndPlayDevices](#) supports additional installation options.

By default, [DiInstallDriver](#) only installs the driver on a device if the driver is a better match to the device than the driver that is currently installed on the device. For information about how Windows selects a driver for device, see [How Windows Selects Drivers](#).

A caller can also call [DiInstallDriver](#) to do the following:

- Force the installation of the specified driver regardless of whether the driver is a better match to the device than the driver that is currently installed on the device.

**Caution** Forcing the installation of the driver can result in replacing a more compatible or newer driver with a less compatible or older driver.

- Indicate to the caller whether a system restart is required to complete the installation.

## **DiRollbackDriver (Windows Vista and later versions of Windows)**

The [DiRollbackDriver](#) function replaces the driver that is currently installed on a device with the previously installed backup driver that is set for a device. This function is provided primarily to restore a device to a working

condition if a device fails after updating the driver for the device. This function performs the same operation that would be performed if a user clicked **Roll Back Driver** on the Driver page for the device in Device Manager.

Windows maintains at most one backup driver for a device. Windows sets a driver as the backup driver for a device immediately after the driver is successfully installed on the device and Windows determines that the device is functioning correctly. However, if a driver does not install successfully on a device or the device does not function correctly after the installation, Windows does not set the driver as the backup driver for the device.

A caller can also call **DiRollbackDriver** to do the following:

- Suppress the display of any user interface component that is associated with the driver rollback.
- Indicate to the caller whether a system restart is required to complete the installation.

For more information about driver rollback, see information about Device Manager in Help and Support Center.

### **UpdateDriverForPlugAndPlayDevices**

The **UpdateDriverForPlugAndPlayDevices** function installs the driver on all devices present in the system that have a hardware ID or compatible ID that matches the driver package.

Calling this function or **DiInstallDriver** is the simplest way for an installation application to install a new driver that is the best match for devices in the system. The basic operation of **UpdateDriverForPlugAndPlayDevices** is similar to the operation of **DiInstallDriver**. However **UpdateDriverForPlugAndPlayDevices** supports additional installation options.

By default, **UpdateDriverForPlugAndPlayDevices** only installs the driver on a device if the driver is a better match to the device than the driver that is currently installed on a device.

A caller can also optionally call **UpdateDriverForPlugAndPlayDevices** to do the following:

- Force the installation of the specified driver regardless of whether the driver is a better match to the device than the driver that is currently installed on the device.

**Caution** Forcing the installation of the driver can result in replacing a more compatible or newer driver with a less compatible or older driver.

- Suppress copying, renaming, or deleting installation files.
- Suppress the display of user interface components.
- Indicate to the caller whether a system restart is required to complete the installation.

# Using SetupAPI To Verify Driver Authenticode Signatures

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the following procedures to verify that a driver has a valid Authenticode [digital signature](#). These procedures are supported starting with Microsoft Windows Server 2003.

## To determine whether a driver has a valid Authenticode signature

Check the DNF\_AUTHENTICODE\_SIGNED flag.

If a driver has a valid Authenticode signature, Windows sets this flag in the **Flags** member of the driver node's **SP\_DRVINSTALL\_PARAMS** structure. (Also be aware that Windows sets the DNF\_INF\_IS\_SIGNED flag if the driver has a [WHQL release signature](#), if it is a system-supplied driver, or if it has an Authenticode signature.)

## To verify that an INF file has a valid Authenticode signature

1. Call the [INF file processing function](#) **SetupVerifyInfFile**.
2. Check the error code that was returned by the function.

If the INF file is not system-supplied and does not have a valid WHQL digital signature, but it does have a valid Authenticode signature, **SetupVerifyInfFile** returns FALSE and [GetLastError](#) returns one of the following error codes:

**ERROR\_AUTHENTICODE\_TRUSTED\_PUBLISHER**

Indicates that the publisher is trusted because the publisher's certificate is installed in the [Trusted Publishers certificate store](#).

**ERROR\_AUTHENTICODE\_TRUST\_NOT\_ESTABLISHED**

Indicates that trust cannot be automatically established because the publisher's signing certificate is not installed in the trusted publisher certificates store. However, this does not necessarily indicate an error. Instead it indicates that the caller must apply a caller-specific policy to establish trust in the publisher.

If the INF file has a valid Authenticode signature, **SetupVerifyInfFile** also returns the following information in the **SP\_INF\_SIGNER\_INFO** output structure:

- The **DigitalSigner** member is set to the name of the signer.
- The **CatalogFile** member is set to the full path of the corresponding signed catalog file.

However, be aware that **SetupVerifyInfFile** does not return the version in the **DigitalSignerVersion** member.

## To verify that a file has a valid Authenticode signature

Call the [SetupAPI function](#) **SetupScanFileQueue** by using the **SPQ\_SCAN\_USE\_CALLBACK\_SIGNERINFO** flag.

**SetupScanFileQueue** sends an **SPFILENOTIFY\_QUEUESCAN\_SIGNERINFO** request to the caller's callback routine and passes a pointer to a **FILEPATHS\_SIGNERINFO** structure. If a file is signed with a valid Authenticode signature, the function sets the error code to the appropriate **ERROR\_AUTHENTICODE\_Xxx** value before calling the callback routine for a file. The function also sets the following information in the **FILEPATHS\_SIGNERINFO** structure:

- The **DigitalSigner** member is set to the name of the signer.
- The **CatalogFile** member is set to the full path of the corresponding signed catalog file.

However, be aware that the version is not set in the **Version** member.

**SetupScanFileQueue** sets the ERROR\_AUTHENTICODE\_Xxx error code in the same way as described earlier in this topic for **SetupVerifyInfFile**.

# Determining the Parent of a Device

12/1/2020 • 2 minutes to read • [Edit Online](#)

Sometimes it is necessary to access the parent of a device. For example, the operation of some types of hardware devices depends on a fixed relationship between a specific parent and set of child devices. To uninstall such a hardware device, you must uninstall the parent in addition to all the child devices. To uninstall the parent, you must obtain a [SP\\_DEVINFO\\_DATA](#) structure for the parent. A Universal Serial Bus (USB) composite device, such as, a multifunction printer, is such a device. It is represented in the system by a parent composite device and one or more child interface devices (see [USB Driver Stack Architecture](#)). To uninstall a multifunction printer, you must uninstall its parent composite device in addition to all its child interface devices.

When the [Plug and Play](#) (PnP) manager configures a device in the system, it adds a device node (*devnode*) for the device to the [device tree](#). When the PnP manager removes a device from the system, it deletes the devnode for the device from the device tree, and the device becomes a *nonpresent device*. The approach that you use to determine the parent of a device depends on how the device is currently configured in the system, as follows:

- If a device has a devnode in the device tree, use [CM\\_Get\\_Parent](#) to obtain a device instance handle for its parent. Given a device instance handle, you can obtain a [device instance ID](#) and an [SP\\_DEVINFO\\_DATA](#) structure for a device. For more information, see [Obtaining the Parent of a Device in the Device Tree](#).
- If a device has a fixed relationship with its parent, you can save and retrieve the device instance ID of its parent. When a device becomes nonpresent, you can use its device instance handle to obtain an [SP\\_DEVINFO\\_DATA](#) structure for the device. For more information, see [Determining the Parent of a Nonpresent Device](#).

# Obtaining the Parent of a Device in the Device Tree

12/1/2020 • 2 minutes to read • [Edit Online](#)

This topic describes how to obtain an **SP\_DEVINFO\_DATA** structure for the parent of a device that has a device node (*devnode*) in the device tree.

## To obtain an SP\_DEVINFO\_DATA structure for the immediate parent of a device in the device tree

1. Verify that the device has a devnode in the [device tree](#) by calling [\*\*CM\\_Get\\_DevNode\\_Status\*\*](#) for the device:
  - If the device has a devnode, the function returns CR\_SUCCESS.
  - If the device does not have a devnode, the function returns CR\_NO\_SUCH\_DEVINST.
2. If the device has a devnode, call [\*\*CM\\_Get\\_Parent\*\*](#) to obtain a device instance handle for the parent of the device.  
(If a device does not have a devnode, [\*\*CM\\_Get\\_Parent\*\*](#) returns a device instance handle for the root device).
3. Using the device instance handle for the parent device, call [\*\*CM\\_Get\\_Device\\_ID\*\*](#) to obtain the device instance ID for the parent device.
4. Using the device instance ID for the parent device, call [\*\*SetupDiOpenDeviceInfo\*\*](#) to obtain an **SP\_DEVINFO\_DATA** structure for the parent device.

## To obtain the SP\_DEVINFO\_DATA structures for a connected sequence of ancestors of a device in the device tree

- Starting with the device's immediate parent and ending with a given ancestor, call [\*\*CM\\_Get\\_Parent\*\*](#) repeatedly until all the handles are obtained. For each call to [\*\*CM\\_Get\\_Parent\*\*](#), supply the device instance handle obtained from the previous call.
- For each device instance handle that you obtain, first call [\*\*CM\\_Get\\_Device\\_ID\*\*](#) to obtain a device instance ID and then call [\*\*SetupDiOpenDeviceInfo\*\*](#) to obtain the **SP\_DEVINFO\_DATA** structure for the device.

For information about how to work with nonpresent devices, see [Determining the Parent of a Nonpresent Device](#).

# Determining the Parent of a Nonpresent Device

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the approach described in this section to determine the parent of a *nonpresent device* only if the relationship between the nonpresent device and its parent is fixed. (If the relationship between a nonpresent device and its parent is not fixed, you cannot use this method because the nonpresent device does not have a specific parent).

For example, this method applies to a USB composite device, such as a multifunction printer, that has one or more interfaces, each of which is represented as a child device. Because all the child interface devices depend on the presence of a specific composite device as their parent, the relationship between the device and its parent is fixed.

The following topics describe this method:

[Saving the Parent/Child Relationship](#)

[Retrieving the Parent/Child Relationship](#)

[Handling a Chain of Ancestors for a Nonpresent Device](#)

## Saving the Parent/Child Relationship

To save the parent/child relationship of a device, supply a *device co-installer* that saves the device instance ID of the device's parent in a user-created entry value under the hardware registry key of the device. You should use a device instance ID because it remains constant across system restarts and between system processes, whereas a device instance handle does not. When you process a **DIF\_INSTALLDEVICE** request in the co-installer, follow these steps to save the device instance ID.

### *To save the device instance ID of the immediate parent in the registry*

1. Call [CM\\_Get\\_Parent](#) to obtain a device instance handle for the parent of the device.
2. Using the device instance handle for the parent device, call [CM\\_Get\\_Device\\_ID](#) to obtain the device instance ID for the parent device.
3. Call [SetupDiOpenDevRegKey](#) by using the DIREG\_DEV flag to obtain a handle to the hardware registry key of the device.
4. Call [RegSetValueEx](#) to save the device instance ID of the parent device in a user-created entry value under the hardware registry key of the device.

## Retrieving the Parent/Child Relationship

After a device co-installer has saved the device instance ID of the parent device in an entry value under a device's hardware registry key, you can retrieve the device instance ID.

### *To retrieve the device instance ID of the parent from the registry*

1. Call [SetupDiOpenDevRegKey](#) using the DIREG\_DEV flag to obtain a handle to the hardware registry key for the device.
2. Call [RegQueryValueEx](#) to retrieve the device instance ID of the parent device that you saved in the entry value that you set in your device co-installer.

After you retrieve the device instance ID of the parent device, call [SetupDiOpenDeviceInfo](#) to obtain an [SP\\_DEVINFO\\_DATA](#) structure for the parent device.

## **Handling a Chain of Ancestors for a Nonpresent Device**

If you require the device instance IDs of a connected sequence of ancestors for a given device, you should save the device instance ID for each ancestor in the registry in a way that you can retrieve them. Be aware that this is valid only if the relationship between the device and all the ancestors is fixed.

One way to do this is for your device co-installer to use **CM\_Get\_Parent** to obtain all the device instance IDs for all the ancestors and save each instance ID in a different entry value under the hardware registry key of the device. You can use the method described in [Saving the Parent/Child Relationship](#) to save the device instance ID of each ancestor. You can then retrieve each device instance ID in the same way as is described in [Retrieving the Parent/Child Relationship](#).

# Obtaining the Original Source Path of an Installed INF File

3/5/2019 • 3 minutes to read • [Edit Online](#)

This topic describes how you can retrieve the original source path of an INF file that is installed in the system INF directory. Although there is no [SetupAPI](#) function to perform this retrieval directly, you can perform the retrieval indirectly by including entries in an INF file so that the SetupAPI functions that access INF file entries can be used to retrieve the original source path information from an installed INF file.

This method only works for INF files that are installed in the system INF file directory. You cannot use this method for INF files that are present in the [driver store](#).

The co-installer that is provided with the toaster sample in the Windows Driver Kit (WDK) uses this method and this topic includes excerpts from the toaster sample that show this method. For more information about the toaster sample, see *toasterpkg.htm*, which is provided in the *src\general\toaster* directory of the WDK.

To use this method to retrieve the original source path of an installed INF file, do the following:

1. Include in the INF file a section that includes an entry whose first field is the %1% string key token. By default, the %1% string key token represents the original source path of the INF file. When Windows installs such an INF file, it saves the original source path string with the installed version of the INF file. Be aware that this method only works if %1% is used as shown in this example. In general, what %1% resolves to is context-dependent. For example, a %1% field in an *add-registry section* entry does not resolve to the original source path; instead %1% in this context resolves to the path of the corresponding INF file in the [driver store](#).
2. Use [SetupOpenInfFile](#), [SetupFindFirstLine](#), and [SetupGetStringField](#) to retrieve the original source path from the entry that includes the %1% string key token.

For example, *toasterpkg.inf* includes the following [ToasterCoInfo] section with a custom OriginalInfSourcePath entry whose first field is the %1% string key token.

```
[ToasterCoInfo]
; Used by the toaster co-installer to figure out where the original media is
; located (so it can start value-added setup programs).
OriginalInfSourcePath = %1%
```

If an INF is configured as illustrated by the toaster sample, you can then retrieve the original source path after installing the INF file in the system INF directory. To retrieve the original source path, first call [SetupOpenInfFile](#) to open the installed INF file. For example, the following code example from *toastco.c* opens the installed *toasterpkg.inf* file.

```
// Since the INF is already in %SystemRoot%\Inf, we need to find out where it
// originally came from. There is no direct way to ascertain an INF's
// path of origin, but we can indirectly determine it by retrieving a field
// from our INF that uses a string substitution of %1% (DIRID_SRCPATH).
//
hInf = SetupOpenInfFile(DriverInfoDetailData->InfFileName,
                       NULL,
                       INF_STYLE_WIN4,
                       NULL
                     );
```

After opening the installed INF file, call **SetupFindFirstLine** to retrieve the first line of the section that contains the entry whose first field is the %1% string key token. Next, call **SetupGetStringField** to retrieve the first field of this entry and retrieve the original source path of the INF file. For example, the following code example from *toastco.c* retrieves the line that contains the custom OriginalInfSourcePath entry and then retrieves the first field of this entry. Because the first field in the original INF is the %1% string key token, **SetupGetStringField** returns the original source path of the INF file.

```
// Contained within our INF should be a [ToastCoInfo] section with the
// following entry:
//
//     OriginalInfSourcePath = %1%
//
// If we retrieve the value (i.e., field 1) of this line, we'll get the
// full path where the INF originally came from.
//
if(!SetupFindFirstLine(hInf, L"ToastCoInfo", L"OriginalInfSourcePath", &InfContext)) {
    goto clean0;
}
if(!SetupGetStringField(&InfContext, 1, *MediaRootDirectory, MAX_PATH, &PathLength) ||
    (PathLength <= 1)) {
    goto clean0;
```

# Guidelines for Using SetupAPI

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following are guidelines for using the [general Setup functions \(SetupXxx\)](#) and [device installation functions \(SetupDiXxx\)](#) that are provided by SetupAPI:

- Never assume that installation file contents are error-free, or that an installation file that you provided hasn't been maliciously modified. Therefore, always validate all information received from SetupAPI functions. Verify that strings are of valid length, that buffers are of valid size, and that index values are within a valid range.
- When writing installation applications for installations on Microsoft Windows XP and later systems, you can call [SetupVerifyInfFile](#) (described in the Windows SDK documentation), which verifies that a digitally signed INF file has not been modified.
- Always test the return value of each SetupAPI function. If the function fails, your code should call [GetLastError](#) to obtain an error code that identifies the failure. Returned error codes can be defined in *Winerror.h* or *Setupapi.h*. Before calling [FormatMessage](#) with [FORMAT\\_MESSAGE\\_FROM\\_SYSTEM](#) to create a text display, always use the [HRESULT\\_FROM\\_SETUPAPI](#) macro (defined in *Winerror.h*) to convert the return value to an HRESULT value. If a SetupAPI function returns successfully, your code must not call [GetLastError](#). (The [GetLastError](#) and [FormatMessage](#) functions, together with system error codes, are described in the Windows SDK documentation.)
- If a SetupAPI function returns a handle, your code must check for a return value of [INVALID\\_HANDLE\\_VALUE](#). Such functions do not return [NULL](#).
- Be aware of the following difference between the [SetupDiXxx](#) and [SetupXxx](#) functions that allow a caller to query for the required size of a buffer:
  - If the caller of a [SetupDiXxx](#) function makes such a query, [GetLastError](#) always returns [ERROR\\_INSUFFICIENT\\_BUFFER](#).
  - If the caller of a [SetupXxx](#) function makes such a query, [GetLastError](#) returns [NO\\_ERROR](#) if no buffer length was specified or [ERROR\\_INSUFFICIENT\\_BUFFER](#) if a buffer was specified that was too small.

# DIFx Guidelines

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting in Windows 10 Version 1607 (Redstone 1), the Driver Install Frameworks (DIFx) tools (`Difxapi.dll`, `Difxapp.dll`, `Difxappa.dll`, and `DPIInst.exe`) are deprecated and are no longer included in the WDK.

Instead, we recommend providing your [driver package](#) as a standalone driver package that does not require an installer. This is a self-contained package that adds its own settings or configuration, rather than depending on an installer to modify system state. Standalone driver packages are required in order to support driver package scenarios such as distributing the driver package through Windows Update and adding the driver package to an offline image. The recommended way to publish a standalone driver package is through Windows Update. The first step to doing this is to submit your driver package to the [Windows Hardware Dev Center](#).

If you choose to use DIFx anyway, you must use an older WDK to get the right tools. The following caveats apply:

- If your driver package specifies only **TargetOSVersion** values of Windows 8.1 or later, you cannot use the DIFxApp MSI custom action due to DIFxApp's dependency on [GetVersionEx](#), an API that changed starting in Windows 8.1. **TargetOSVersion** is specified in the [INF Manufacturer Section](#). DIFxApp exposes MSI custom actions such as `MsiProcessDrivers`, `MsiInstallDrivers`, and `MsiUninstallDrivers`. If your driver package specifies **TargetOSVersion** values of Windows 8.1 or later, you cannot use these custom actions in your MSI.
- Starting in Windows 8.1, applications that link to `Difxapi.dll` must contain an app manifest targeting the OS version on which the application is intended to run. This is due to DIFxAPI's dependency on [GetVersionEx](#), an API that changed starting in Windows 8.1. For more on changes to [GetVersionEx](#) in Windows 8.1, see [Targeting your application for Windows](#).
- If your driver package uses the **BuildNumber** part of **TargetOSVersion** (introduced in Windows 10, version 1607 (Build 14310 and later)), you cannot use the DIFx tools with that driver package. The DIFx tools do not support BuildNumber targeting.
- Use DIFx version 2.1, which is available in the Windows 7 WDK through the Windows 10 Version 1511 WDK. Although a DIFx version of 2.1 was available in earlier versions of the WDK, it was not compatible with Windows 7 and later versions of Windows.

Although it's no longer being updated, you can find API reference documentation for DIFx at [Difxapi.h](#).

If you still need a custom installer to install your driver package, use either the [PnPUtil](#) command line tool or a custom installer that calls [driver installation functions](#).

Similarly, if you need the custom installer to uninstall the driver package, use either [PnPUtil](#) or a custom installer that calls [DiUninstallDriver](#) or [SetupUninstallOEMInf](#).

# Device and Driver Installation Software Components

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section contains the following topics which describe the various software components that can be optionally developed to enhance device and driver installations:

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

- [Writing a Co-installer](#)
- [Calling the Default DIF Code Handlers](#)
- [Finish-Install Actions](#)
- [Providing Device Property Pages](#)
- [Writing a Device Installation Application](#)

# Writing a Co-installer

12/5/2018 • 2 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

A co-installer is a Microsoft Win32 DLL that assists in device installation. Co-installers are called by SetupAPI 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 includes the following topics:

[Co-installer Operation](#)

[Co-installer Interface](#)

[Co-installer Functionality](#)

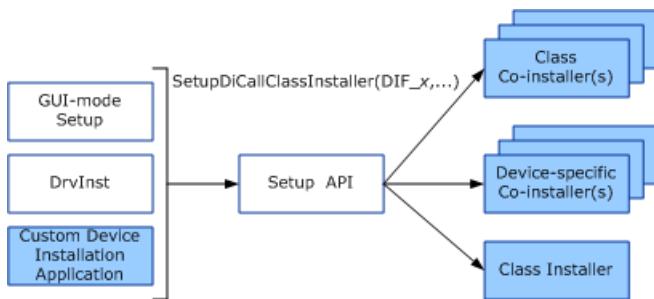
[Handling DIF Codes](#)

[Registering a Co-installer](#)

# Co-installer Operation

12/1/2020 • 3 minutes to read • [Edit Online](#)

Co-installers are called by SetupAPI, as shown in the following figure.



The unshaded boxes represent the components that the operating system supplies for the [system-supplied device setup classes](#). The shaded boxes represent the components that you can provide. If you create a custom device setup class, you can also supply a class installer. However, you rarely need to create a new device setup class, because almost every device can be associated with one of the system-supplied device setup classes. For more information about Windows components, see [Device Installation Overview](#).

Co-installers can be provided for a specific device (*device-specific co-installer*) or for a device setup class (*class co-installer*). SetupAPI calls a device-specific co-installer only when installing the device for which the co-installer is registered. The operating system and vendors can register zero or more device-specific co-installers for a device. SetupAPI calls a class co-installer when installing any device of the device setup class for which the co-installer is registered. The operating system 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.

Windows and [custom device installation applications](#) install devices by calling [SetupDiCallClassInstaller](#) with [device installation function codes](#) (DIF codes).

During GUI-mode setup, the operating system calls [SetupDiCallClassInstaller](#) with DIF codes to detect non-PnP devices that are present in the system. An IHV would typically provide a co-installer to perform this action for non-PnP devices that the IHV releases.

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.

The operating system adds device-specific co-installers 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](#)

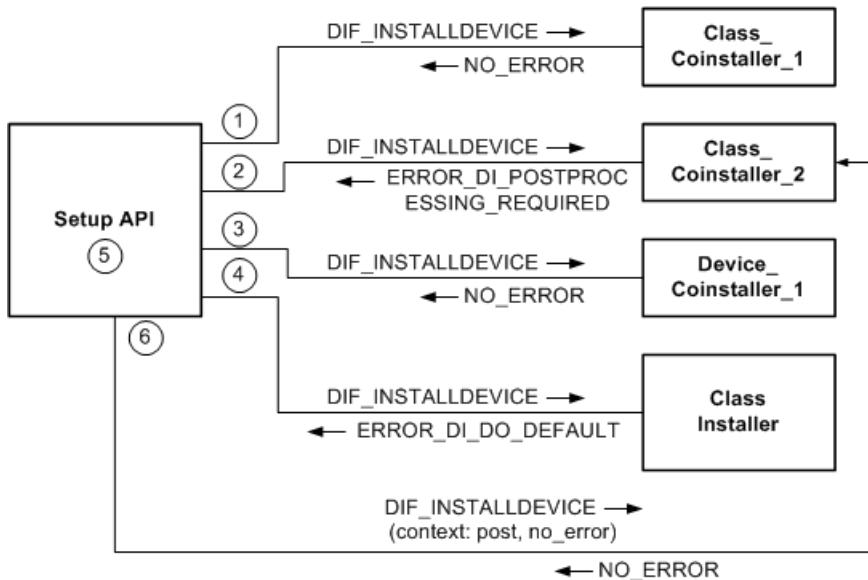
[DIF\\_NEWDEVICEWIZARD\\_SELECT](#)

[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, or 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.



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 SetupAPI.
6. **SetupDiCallClassInstaller** calls any co-installers that requested postprocessing. In this example, the

second class co-installer requested postprocessing.

Co-installer postprocessing 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 postprocessing, 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.

# Co-installer Interface

11/2/2020 • 2 minutes to read • [Edit Online](#)

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* COINSTALLED_PROC) (
    IN DI_FUNCTION InstallFunction,
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData OPTIONAL,
    IN OUT PCOINSTALLED_CONTEXT_DATA Context
);
```

### *InstallFunction*

Specifies the device installation request being processed, in which the co-installer has the option of participating. These requests are specified using DIF codes, such as DIF\_INSTALLDEVICE. For more information, see [Device Installation Function 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-NUL, it identifies a device information element in the device information set. *DeviceInfoData* is non-NUL 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 [COINSTALLED\\_CONTEXT\\_DATA](#) structure.

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.

The co-installer must also return NO\_ERROR if it receives an unrecognized DIF code. (Note that class installers return ERROR\_DI\_DO\_DEFAULT for unrecognized DIF codes.)

### 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 that describes 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;
```

The following list describes the members of this structure:

- **PostProcessing** is TRUE when a co-installer is called after the appropriate class installer, if any, has processed the DIF code specified by *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.

# Co-installer Functionality

11/2/2020 • 2 minutes to read • [Edit Online](#)

A co-installer is a user-mode Win32 DLL that typically writes additional configuration information to the registry, or performs other installation tasks that require 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 device property pages](#), which are displayed by Device Manager so users can modify device parameters.
- Starting with Windows Vista, provide [finish-install actions](#) (in response to a [DIF\\_FINISHINSTALL\\_ACTION](#) request) to install applications.

When called for postprocessing, a co-installer must check the **InstallResult** member of the [COINSTALLED\\_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 can sometimes obtain information from the user. Such information might include additional device parameters, or whether the user wants device-specific applications installed. Co-installers can create user interfaces by providing "finish install" pages and device property pages. No other form of user interface is allowed. Windows displays "finish install" pages at the end of the installation (within the Found New Hardware or Hardware Update). Device Manager displays property pages, and allows users with administrator privilege to modify parameters displayed on these pages.

# Handling DIF Codes

11/2/2020 • 3 minutes to read • [Edit Online](#)

Device installation 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](#)

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

### **When Sent**

Describes the typical times when, and reasons why, a device installation 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](#) ([SetupDiXxx](#) 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 [SetupDiXxx](#) 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](#) structure.

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 values for the DIF code. See the following figure for more information about return values.

#### **Default DIF Code Handler**

Specifies the **SetupDiXxx** function that carries out the system-defined default 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 are registered for postprocessing).

If a class installer successfully handles a DIF code and **SetupDiCallClassInstaller** should subsequently call the default handler, the class installer returns **ERROR\_DI\_DO\_DEFAULT**.

If the class installer successfully handles a DIF code, including directly calling the default handler, the class installer should return **NO\_ERROR** and **SetupDiCallClassInstaller** will not subsequently call the default handler again. Note that the class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

If the class installer encounters an error, the installer should return an appropriate Win32 error code and **SetupDiCallClassInstaller** will not subsequently call the default handler.

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

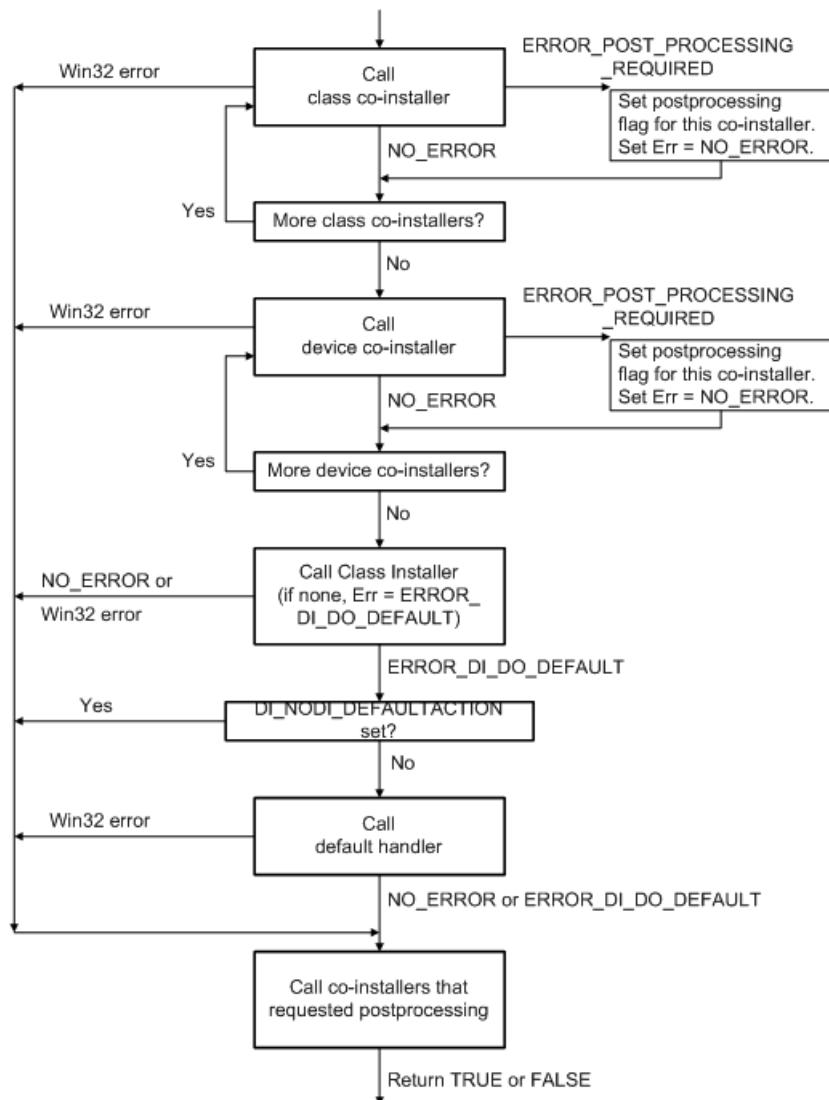
#### **Installer Operation**

Describes typical steps that an installer might take 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.



The operating system 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.

# Registering a Co-installer

12/5/2018 • 2 minutes to read • [Edit Online](#)

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. A class co-installer is 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.

# Registering a Device-Specific Co-installer

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 *XxxCopyFilesSection* 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 *install-section-name* entry for the co-installer can be decorated with an optional OS/architecture extension (for example, cdrom\_install.NTx86.CoInstallers). For more information, see [INF DDInstall Section](#).

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. SetupAPI then calls the co-installers at each subsequent step of the installation process. If more than one co-installer is registered for a device, SetupAPI calls them in the order in which they are listed in the registry.

# Registering a Class Co-installer

11/2/2020 • 2 minutes to read • [Edit Online](#)

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](#).

# Calling the Default DIF Code Handlers

12/1/2020 • 2 minutes to read • [Edit Online](#)

Default DIF code handlers perform system-defined default operations for [DIF codes](#). As described in [Handling DIF Codes](#), [SetupDiCallClassInstaller](#) calls the default handler for a DIF request after the *class installer* and *co-installer* have first processed the DIF request, but before [SetupDiCallClassInstaller](#) recalls the co-installers that registered for post-processing of the request.

**Note** The operation of [SetupDiCallClassInstaller](#) cannot be configured to recall the class installer to post-process a DIF request.

In those situations where a *class installer* must perform operations for a DIF request after the default handler is called, the class installer must directly call the default handler when it processes the DIF request, as follows:

1. Perform operations that must be done before calling the default handler.
2. Call the default handler to perform the default operations.

**Note** The class installer must not attempt to supersede the operation of the default handler.

3. Perform the operations that must be done after the default handler returns.
4. Return NO\_ERROR if the class installer successfully completed processing the DIF request or return a Win32 error if the processing failed.

**Important** Co-installers and device installation applications must not call the default DIF code handlers.

For an example of a situation where this method must be used, see the information about calling the default handler [SetupDiInstallDevice](#) on the [DIF\\_INSTALLDEVICE](#) request reference page.

The following table lists the DIF codes that have default handlers.

DIF CODE	DEFAULT DIF CODE HANDLER FUNCTION
<a href="#">DIF_PROPERTYCHANGE</a>	<a href="#">SetupDiChangeState</a>
<a href="#">DIF_FINISHINSTALL_ACTION</a>	<a href="#">SetupDiFinishInstallAction</a>
<a href="#">DIF_INSTALLDEVICE</a>	<a href="#">SetupDiInstallDevice</a>
<a href="#">DIF_INSTALLINTERFACES</a>	<a href="#">SetupDiInstallDeviceInterfaces</a>
<a href="#">DIF_INSTALLDEVICEFILES</a>	<a href="#">SetupDiInstallDriverFiles</a>
<a href="#">DIF_REGISTER_COINSTALLERS</a>	<a href="#">SetupDiRegisterCoDeviceInstallers</a>
<a href="#">DIF_REGISTERDEVICE</a>	<a href="#">SetupDiRegisterDeviceInfo</a>
<a href="#">DIF_REMOVE</a>	<a href="#">SetupDiRemoveDevice</a>
<a href="#">DIF_SELECTBESTCOMPATDRV</a>	<a href="#">SetupDiSelectBestCompatDrv</a>
<a href="#">DIF_SELECTDEVICE</a>	<a href="#">SetupDiSelectDevice</a>

DIF CODE	DEFAULT DIF CODE HANDLER FUNCTION
DIF_UNREMOVE	<a href="#">SetupDiUnremoveDevice</a>

# Finish-Install Actions

11/2/2020 • 3 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

*Finish-install actions* allow the installer to complete installation operations.

Installers can specify finish-install actions to happen in a class installer, class co-installer, or device co-installer, starting with Windows Vista and later versions. Finish-install actions run in the context of an administrator *after* all other installation operations, including finish-install wizard pages, are completed.

In Windows 7, the default finish-install action is provided by the system-supplied [SetupDiFinishInstallAction](#) function. This function processes, in the interactive context of an administrator, the [RunOnce registry entries](#) that are set for a device. If a device does not have a class installer, or a class installer returns ERROR\_DI\_DO\_DEFAULT in response to a [DIF\\_FINISHINSTALL\\_ACTION](#) request, Windows calls [SetupDiFinishInstallAction](#) after all the installers for a device complete their finish-install actions.

In Windows 8 and later versions, finish-install actions are not automatically run as part of device installation, and the [SetupDiFinishInstallAction](#) function has been removed. Instead, an administrator (or a limited user that can provide administrator credentials to a UAC prompt) must go to the Action Center and address the "Finish installing device software" maintenance item in order for the finish-install action to be run. Until then, the finish-install action will not run. For example, if a user plugs in a device that installs a driver that includes a finish-install action, the finish-install action will not automatically run at that time. Instead, the finish-install action will run at some later point when the user manually initiates it. Thereafter, when Windows runs the finish-install action, the action has that single opportunity to run. If the action fails then it must take appropriate steps to allow the user to try again and finish later. Similarly, installing supporting software that should accompany a driver can still be accomplished with a finish-install action, but it will also not be installed automatically.

Alternatively, depending on your scenario, in Windows 8 and later versions, you may be able to make use of the new device app model. More information about device apps can be found at [Design Great Hardware Experiences](#).

Finish-install actions are useful in the following situations:

- To run a device-specific application installation program that is not designed to run as part of a finish-install wizard page. If such an installation program has its own user interface, using a finish-install action to install the application provides a better user experience.

For example, assume that a device manufacturer wants to install a device-specific application in addition to a driver for a device, and the device-specific application has its own installation program with its own user interface. To provide the best user experience, the device manufacturer would run the installation program as a finish-install action. In this way, when Windows detects the device and finds the driver, Windows first installs the driver and then runs the installation program for the application.

- To run an installation program that can only run in an interactive user context (a client-side installation). For example, such an installation program can be started by using an [InteractiveInstall](#) directive in the [INF ControlFlags Section](#) of a driver package's INF file.

**Note** Starting with Windows Vista, such an installation program cannot be run in the same way as on earlier versions of Windows. This is because Windows Vista and later versions of Windows do not support the installation of devices within a client-side installation. However, such an installation program can be run as a finish-install action if the driver package includes a class installer, class co-installer, or device co-installer that starts the installation program.

This section discusses finish-install actions in more detail and includes the following topics:

[Overview of Finish-Install Actions](#)

[Implementing Finish-Install Actions](#)

[How Finish-Install Actions are Processed](#)

# Overview of Finish-Install Actions

12/5/2018 • 2 minutes to read • [Edit Online](#)

If an *installer* (a class installer, class co-installer, or device co-installer) provides finish-install actions for a device, Windows runs the finish-install actions *after* all other device installation operations for the device have completed, and the device has been started. On a new system, finish-install actions run the first time that the operating system is started after Windows is finished.

Device installation operations include the following:

- The *core device installation* (also known as *server-side installation*), in which the driver for the device is installed in a trusted system context and without user interaction.

Windows processes *finish-install actions* in the same way regardless of whether the installation is initiated by connecting a Plug and Play device to a computer (a *hardware-first installation*) or the installation is initiated by running an installation program such as the Found New Hardware Wizard, the Update Driver Software Wizard, or a vendor-supplied installation program (a *software-first installation*).

Windows runs finish-install actions only in the context of a user who has administrator privileges.

Starting with Windows Vista, User Account Control (UAC) enables users to run at lower privilege most of the time and elevate privilege only when necessary. Therefore, if the finish-install process is invoked, Windows prompts the user for the consent and the credentials that are required to run the finish-install actions. If the user does not supply consent or provide the necessary credentials, the finish-install actions are deferred until a user who does supply consent and the necessary credentials signs in.

**Note** Starting with Windows 7, if UAC is set to the default setting (Notify me only when programs try to make changes to my computer) or a lower setting, the operating system does not display the prompt for users with administrative privileges when it processes finish-install actions.

# How Finish-Install Actions are Processed

11/2/2020 • 3 minutes to read • [Edit Online](#)

Finish-install actions for a device are processed in the same way by an *installer* (a class installer, class co-installer, or device co-installer), regardless of whether the installation was a *hardware-first installation* or the installation is initiated by running an installation program such as the Found New Hardware Wizard, the Update Driver Software Wizard, or a vendor-supplied installation program (a *software-first installation*).

**Note** In Windows 8, Windows 8.1, and Windows 10, finish-install actions must be completed in the Action Center by an administrator (or a limited user that can provide administrator credentials to a UAC prompt). Users must click on "Finish installing device software".

Windows processes finish-install actions after all other installation operations have completed and the device has been started, including:

- Core device installation (also known as *server-side installation*), in which the driver for the device is installed and loaded by the system's Plug and Play (PnP) manager components.

Windows completes the following steps to process an installer's finish-install actions:

- At the end of core device installation, Windows calls [SetupDiCallClassInstaller](#) to send a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request to the installers for the device.

**DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** is the only DIF code that is sent in both the context of core device installation and in the client context. Therefore, a class installer, class co-installer, or device co-installer must indicate that it has finish-install actions during **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** processing, instead of during **DIF\_INSTALLDEVICE** processing.

- If an installer provides finish-install actions, it sets the **DIF\_FLAGSEX\_FINISHINSTALL\_ACTION** flag in response to a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request. If the **DIF\_FLAGSEX\_FINISHINSTALL\_ACTION** flag is set after all the installers have processed a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request, the device is flagged to perform a finish install action.

For more information about this operation, see [Marking a Device as having a Finish-Install Action to Perform](#).

- After core device installation is complete for a device, Windows checks whether the device has been flagged to perform a finish-install action. If it has, Windows queues a finish-install process that performs the finish-install actions specific to the device. The process executes in the user's context.

In Windows 8 and later versions, finish-install actions are not automatically run as part of device installation. Instead, an administrator (or a limited user that can provide administrator credentials to a UAC prompt) must go to the Action Center and address the "Finish installing device software" maintenance item for the finish-install action to run. Until then, the finish-install action will not run. For example, if a user plugs in a device that installs a driver that includes a finish-install action, the finish-install action will not automatically run at that time. The finish-install action runs at a later point when the user manually initiates it. When Windows runs the finish-install action, the action has that single opportunity to run. If the action fails then it must take appropriate steps to allow the user to try again and finish later. Installing supporting software that should accompany a driver can still be accomplished with a finish-install action, but it will also not be installed automatically.

In Windows 7, the finish-install process runs only in the context of a user with administrator credentials at one of the following times:

- The next time that a user who has administrator credentials logs on while the device is attached.
- When the device is reattached.
- When the user selects **Scan for hardware changes** in Device Manager.

If a user is signed in without administrative privileges, Windows prompts the user for consent and credentials to run the finish-install actions in an administrator context.

4. When finish-install operations run, the finish-install process starts and completes any finish-install wizard pages for the device, and then calls **SetupDiCallClassInstaller** to send a **DIF\_FINISHINSTALL\_ACTION** request to all installers for the device, as described in [Running Finish-Install Actions](#).
5. After the installers have completed their finish-install actions, Windows runs the default finish-install action, as described in [Running the Default Finish-Install Action](#).

# Marking a Device as having a Finish-Install Action to Perform

11/2/2020 • 2 minutes to read • [Edit Online](#)

An *installer* (a class installer, class co-installer, or device co-installer) indicates to Windows that it has finish-install actions to perform by setting the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag when the installer processes a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request. This action will cause Windows to flag the device as needing to perform a finish install action. The steps are as follows:

1. When an installer receives a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request, the installer sets the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag if it has finish-install actions to perform.

The installer then returns one of the following error codes:

- ERROR\_DI\_DO\_DEFAULT if the installer is a class installer that has no finish-install wizard pages.
- NO\_ERROR if the installer is a class installer that has finish-install wizard pages or a co-installer that either has or does not have finish-install wizard pages.

2. If the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag is set for a device after all installers have processed the **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request for the device, Windows flags the device as needing to perform a finish install action.

# Running Finish-Install Actions

11/2/2020 • 3 minutes to read • [Edit Online](#)

In Windows 8, and later versions of Windows, finish-install actions do not automatically run as part of device installation. When a device is installed with a driver that includes a finish-install action, the finish-install action will not automatically run. Instead, Windows prompts a user to “Finish installing device software” in the Notifications area or in Windows Action Center. Installing device software requires administrator permissions. If installation fails, the software must prompt the user to try the installation again. Installing supporting software that should accompany a driver can still be accomplished with a finish-install action, but it will not be installed automatically.

Prior to Windows 8, if a device is flagged as needing to perform a finish-install action, Windows initially attempts to complete the finish-install actions by running a finish-install process at one of the following times:

- For a device that is installed during Windows setup, the first time that an administrator logs on to Windows after Windows setup is finished.
- For a device that is installed or reinstalled after Windows is installed, after the core device installation operations have completed, as follows:
  - For a [hardware-first installation](#) of a device, Windows runs the initial finish-install process. If the current user is not an administrator, Windows will first prompt the user to enter the credentials of an administrator before it runs the initial finish-install process.
  - For a [software-first installation](#) of a device, Windows runs the initial finish-install process in the context of the administrator who initiated the installation or reinstallation.

Prior to Windows 8, if the initial attempt to complete the finish-install actions succeeds, the finish-install process clears the device as being flagged to perform a finish install action. If the initial attempt to complete the finish-install actions fails, the finish-install process does not clear the device as being flagged to perform a finish install action and exits. Subsequently, while the device remains flagged to perform a finish install action, Windows repeatedly attempts to complete the finish-install actions by running a new finish-install process whenever the device is enumerated, as follows:

- While the device remains installed, the next time an administrator logs on.
- If an administrator clicks Scan for hardware changes on the **Action** menu of Device Manager or an installation program calls [CM\\_Reenumerate\\_DevNode](#) in the context of an administrator.

If the device is flagged to perform a finish-install action, the finish-install process calls [SetupDiCallClassInstaller](#) to send a [DIF\\_FINISHINSTALL\\_ACTION](#) request to installers for the device.

If an installer has finish-install actions, the installer performs finish-install actions and returns an appropriate error code for the [DIF\\_FINISHINSTALL\\_ACTION](#) request. An installer returns one of the error codes in the following table.

ERROR CODE	MEANING
------------	---------

ERROR CODE	MEANING
ERROR_DI_DO_DEFAULT	<p>Class installer: The class installer has successfully run its finish-install actions and is requesting that Windows perform its default processing.</p> <p>The class installer also returns this error code if it has no finish-install actions, or a finish-install action fails and should not be attempted again.</p> <p>Device or class co-installer: Co-installers do not return this error code.</p>
NO_ERROR	<p>Class installer: The class installer has successfully run its finish-install action. Windows should not perform its default processing.</p> <p>Device or class co-installer: The co-installer has either successfully run its finish-install actions or has no finish-install actions.</p> <p>The co-installer also returns this error code if a finish-install action fails and should not be attempted again.</p>
Microsoft Win32 error	<p>The class installer, device co-installer, or class co-installer encountered an error while processing a finish-install action, but should attempt to process the finish-install action again.</p> <p>By returning a Win32 error code, the installer indicates that Windows should run another finish-install process to complete the finish-install actions the next time the device is enumerated. The installer should also inform the user of this situation.</p>

# Running the Default Finish-Install Action

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows 7, the default finish-install action is provided by the system-supplied [\*\*SetupDiFinishInstallAction\*\*](#) function.

If a device does not have a class installer, or a class installer returns ERROR\_DI\_DO\_DEFAULT in response to a [\*\*DIF\\_FINISHINSTALL\\_ACTION\*\*](#) request, Windows calls [\*\*SetupDiFinishInstallAction\*\*](#) after all the installers for a device complete their finish-install actions.

In Windows 8 and later versions, there are no default finish-install actions.

# Implementing Finish-Install Actions

11/2/2020 • 2 minutes to read • [Edit Online](#)

*Installers* (a class installer, class co-installer, or device co-installer) supply finish-install actions. A finish-install action can run an executable program, create a process, create a thread, or execute code in the device driver installation finish-install process.

To implement finish-install actions, an installer:

1. Sets the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag when the installer processes a **DIF\_NEWDVICEWIZARD\_FINISHINSTALL** DIF code and returns one of the following error codes:
  - ERROR\_DI\_DO\_DEFAULT if it is a class installer without finish-install wizard pages.
  - NO\_ERROR if it is a class installer with finish-install wizard pages or a co-installer with or without finish-install wizard pages.
2. Performs the finish-install actions when it processes a **DIF\_FINISHINSTALL\_ACTION** request.

An installer returns one of the error codes in the following table.

ERROR CODE	MEANING
ERROR_DI_DO_DEFAULT	Class installer: The class installer has successfully run its finish-install actions and is requesting Windows to perform its default processing. A class installer should also return this error code if it has no finish-install actions.  Device or class co-installer: Co-installers do not return this error code.
NO_ERROR	Class installer: The class installer has successfully run its finish-install action. Windows should not perform its default processing.  Device or class co-installer: The co-installer has either successfully run its finish-install actions or has no finish-install actions.
Microsoft Win32 error	The installer encountered an error, but the finish-install action should be attempted again. Returning a Win32 error code indicates that Windows should run another finish-install process to complete the finish-install actions the next time the device is enumerated.

**Note** If a finish-install action fails and should not be attempted again, a class installer returns ERROR\_DI\_DO\_DEFAULT and a device or class co-installer returns NO\_ERROR.

For information about how to develop finish-install actions, see [Guidelines for Implementing Finish-Install Actions](#). For sample code that shows how to implement finish-install actions, see the following topics:

[Code Example: Finish-Install Actions in a Class Installer](#)

[Code Example: Finish-Install Actions in a Co-installer](#)

# Guidelines for Implementing Finish-Install Actions

11/2/2020 • 4 minutes to read • [Edit Online](#)

Finish-install actions can be run by an *installer* (a class installer, class co-installer, or device co-installer). In its finish-install actions, an installer can run an executable program, create a process, create a thread, or execute code in the device driver installation finish-install process.

Consider the following guidelines when you implement finish-install actions in an installer:

- Finish-install actions cannot be used to apply any critical settings that are required for a device to work.
- An installer should wait for the finish-install action to finish if the finish-install action must run to completion.

For example, to avoid being interrupted by a system restart while a finish-install action is still running, an installer should wait for the finish-install action to finish before the installer returns from processing a **DIF\_FINISHINSTALL\_ACTION** request.

- A finish-install action should inform a user of progress.

Windows does not inform a user that finish-install actions are running or that finish-install actions succeed or fail. Therefore, a finish-install action should inform the user that a finish-install action is in progress and then notify the user that the finish-install action succeeded or failed.

- An installer must handle the situation where a system restart is required to complete the finish-install actions.

If a finish-install action requires a system restart before the settings take effect on the device, the installer should set the DI\_NEEDREBOOT flag before it returns from processing a **DIF\_FINISHINSTALL\_ACTION** request. However, a device installation should not force a restart of a computer unless absolutely necessary.

For more information about when a device installation should require a system restart, see [Device Installations and System Reboots](#).

- An installer should handle the situation where a finish-install action fails, but should be attempted again. For example, the installer can fail the finish-install action in this way if the device that is being installed has been removed from the system.

Prior to Windows 8, if a finish-install action fails, but should be attempted again, an installer should notify the user of the temporary failure, perform any necessary cleanup, and return a Win32 error code for the **DIF\_FINISHINSTALL\_ACTION** request. If an installer returns a Win32 error code for a **DIF\_FINISHINSTALL\_ACTION** request, Windows does not clear the device as having been flagged to perform a finish install action for the device node (*devnode*).

However, starting with Windows 8, returning an error code will not prevent the flag being cleared. If the finish-install action has an error, it needs to provide the user with the ability to run it again in the future.

While this flag remains set for the device, Windows runs a new finish-install process.

For more information, see [Running Finish-Install Actions](#).

- An installer should handle the situation where a finish-install action fails and should not be attempted again.

If an error makes it impossible for a finish-install action ever to succeed, an installer should notify the user that the action cannot be completed, and then perform any necessary cleanup. In this situation, a co-installer

should return NO\_ERROR and a device or class installer should return ERROR\_DI\_DO\_DEFAULT. Windows will subsequently clear the device as having been flagged to perform a finish install action for the devnode and call [SetupDiFinishInstallAction](#) to perform the default finish-install operations.

- When the installer processes a [DIF\\_NEWDEVICEWIZARD\\_FINISHINSTALL](#) DIF code, it should check to see if any finish-install actions are needed. The installer should only set the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag if there are finish-install actions that must be performed. If this flag is set unnecessarily, users get an extra device installation prompt during reinstallation of the driver, and the DIF\_FINISHINSTALL\_ACTION request has no finish-install actions to perform.

For example, consider a device co-installer where the finish-install action installs an application that is required for the device to work properly. For instance, the finish-install action for a Microsoft keyboard might install the IntelliType application. When such a co-installer processes the [DIF\\_NEWDEVICEWIZARD\\_FINISHINSTALL](#) DIF code, it should check to see whether the application is already installed. If the application is already installed, there is no finish-install action to perform, and therefore the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag should not be set. In this situation, if the co-installer incorrectly sets the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag, the user gets an undesired User Account Control (UAC) prompt for permission to proceed even though the finish-install action has no action to perform.

**Note** Starting with Windows 7, if UAC is set to the default setting ("Notify me only when programs try to make changes to my computer") or a lower setting, the operating system does not display the prompt for users with administrative privileges when it processes finish-install actions.

- Before you register an installer that implements finish-install actions, you must include and install all the files that are needed to run the finish-install actions in the [CopyFiles directive](#) of the [INF file](#) for the device. This is required so that the files get placed during the installation in a location that is accessible by the installer.

For more information about the registration requirements of a device or class co-installer, see [Registering a Class Co-installer](#).

# Code Example: Finish-Install Actions in a Class Installer

11/2/2020 • 3 minutes to read • [Edit Online](#)

In this example, a class installer performs the following operations to support finish-install actions:

- When the class installer receives a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request, it calls the installer-supplied function *FinishInstallActionsNeeded* to determine whether there are finish-install actions to perform. (The code for the *FinishInstallActionsNeeded* function is not shown in this example.)  
If *FinishInstallActionsNeeded* returns TRUE, the class installer calls **SetupDiGetDeviceInstallParams** to retrieve the device installation parameters for the device. It then calls **SetupDiSetDeviceInstallParams** to set the **FlagsEx** member of the **SP\_DEVINSTALL\_PARAMS** structure for the device with the **DI\_FLAGSEX\_FINISHINSTALL\_ACTION** flag. Setting this flag causes Windows to send a **DIF\_FINISHINSTALL\_ACTION** request to all class installers, class co-installers, and device co-installers that are involved in the installation of this device. This request is sent after all installation operations except the finish-install actions have completed.
- When the class installer receives a **DIF\_FINISHINSTALL\_ACTION** request, it again calls *FinishInstallActionsNeeded* to determine whether it has finish-install actions to perform and, if so, performs the finish-install actions. The class installer notifies the user that finish-install actions are in progress and waits for the finish-install actions to complete before returning from processing the **DIF\_FINISHINSTALL\_ACTION** request.
- If the finish-install actions succeed, the class installer notifies the user that the finish-install actions succeeded.
- If the finish-install actions required a restart of the system to complete the finish-install actions, the class installer calls **SetupDiGetDeviceInstallParams** to retrieve the device installation parameters for the device and then calls **SetupDiSetDeviceInstallParams** to set the **Flags** member of the **SP\_DEVINSTALL\_PARAMS** structure for a device with the **DI\_NEEDREBOOT** flag. The installer also notifies the user that a system restart is required.
- If the finish-install actions fail, but the finish-install actions should be attempted again the next time the device is enumerated, the class installer notifies the user of this situation.

**Note** Starting in Windows 8 a finish-install action is only run once. Windows will not automatically run it again, especially not the next time the device is enumerated because that is not when finish-install actions are run.

- If the finish-install actions fail and the class installer determines that the finish-install actions cannot ever succeed, the class installer notifies the user of this situation.
- By default, the class installer returns **ERROR\_DI\_DO\_DEFAULT** in response to a **DIF\_FINISHINSTALL\_ACTION** request if the finish-install actions succeeded or if the finish-install actions failed and the installer determines that the finish-install actions should not be attempted again. The installer returns a Win32 error code only if the finish-install actions fail and the finish-install actions should be attempted again the next time the device is enumerated in the context of an administrator.

**Note** Starting in Windows 8 a finish-install action is only run once. Windows will not automatically run it again, especially not the next time the device is enumerated because that is not when finish-install actions are run.

The following class installer code example shows the basic structure of class installer code that implements finish-install actions:

```
DWORD CALLBACK
SampleClassInstaller(
    IN DI_FUNCTION InstallFunction,
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData,
)
{
    SP_DEVINSTALL_PARAMS DeviceInstallParams;
    DWORD ReturnValue = ERROR_DI_DO_DEFAULT; // The default return value

    switch(InstallFunction)
    {
        case DIF_NEWDEVICEWIZARD_FINISHINSTALL:
            //
            // Processing for finish-install wizard pages
            // If the class installer has wizard pages,
            // set ReturnValue to NO_ERROR
            //
            // Processing for finish-install actions
            if (FinishInstallActionsNeeded())
            {
                // Obtain the device install parameters for the device
                // and set the DI_FLAGSEX_FINISHINSTALL_ACTION flag
                DeviceInstallParams.cbSize = sizeof(DeviceInstallParams);
                if (SetupDiGetDeviceInstallParams(DeviceInfoSet, DeviceInfoData, &DeviceInstallParams))
                {
                    DeviceInstallParams.FlagsEx |= DI_FLAGSEX_FINISHINSTALL_ACTION;
                    SetupDiSetDeviceInstallParams(DeviceInfoSet, DeviceInfoData, &DeviceInstallParams);
                }
            }
            break;

        case DIF_FINISHINSTALL_ACTION:
            // Processing for finish-install actions
            if (FinishInstallActionsNeeded())
            {
                //
                // Perform the finish-install actions,
                // notify the user that finish install actions
                // are in progress and wait for
                // the finish-install actions to complete
                //
                // If the finish-install actions succeed, notify the user
                //
                // If the finish install actions require a system restart:
                // notify the user, call SetupDiGetDeviceInstallParams
                // to obtain the device install parameters for the device in
                // DeviceInstallParams, and call SetupDiSetInstallParams to set
                // the DI_NEEDREBOOT flag in DeviceInstallParams.Flags
                //
                // If the finish install actions failed, but
                // should be attempted again: clean up,
                // notify the user of the failure, and
                // set ReturnValue to an appropriate Win32 error code
                //
                // If the finish install actions failed and
                // should not be attempted again: clean up and
                // notify the user of the failure
                //
                // Starting with Windows 8, a finish-install action
                // is only run once. Windows will not automatically
                // run it again, especially not the next time
                // the device is enumerated because that is not when
                // finish-install actions are run.
                //
            }
    }
}
```

```
//  
}  
break;  
}  
  
return ReturnValue;  
}
```

# Code Example: Finish-Install Actions in a Co-installer

11/2/2020 • 3 minutes to read • [Edit Online](#)

In this example, a co-installer performs the following operations to support finish-install actions:

- When the co-installer receives a **DIF\_NEWDEVICEWIZARD\_FINISHINSTALL** request, it calls the installer-supplied function *FinishInstallActionsNeeded* to determine whether there are finish-install actions to perform. (The code for the *FinishInstallActionsNeeded* function is not shown in this example).

If *FinishInstallActionsNeeded* returns TRUE, the co-installer calls **SetupDiGetDeviceInstallParams** to retrieve the device installation parameters for the device and then calls **SetupDiSetDeviceInstallParams** to set the **FlagsEx** member of the **SP\_DEVINSTALL\_PARAMS** structure for the device with the DI\_FLAGSEX\_FINISHINSTALL\_ACTION flag. Setting this flag causes Windows to send a **DIF\_FINISHINSTALL\_ACTION** request to all class installers, class co-installers, and device co-installers that are involved in the installation of this device. This request is sent after all installation operations, except the finish-install actions, have completed.
- When the co-installer receives a **DIF\_FINISHINSTALL\_ACTION** request, the co-installer again calls *FinishInstallActionsNeeded* to determine whether it has finish-install actions to perform and, if so, performs the finish-install actions. The co-installer notifies the user that finish-install actions are in progress and waits for the finish-install actions to complete before returning from processing the **DIF\_FINISHINSTALL\_ACTION** request.
  - If the finish-install actions succeed, the co-installer notifies the user that finish-install actions succeeded.
  - If the finish-install actions required a system restart to complete the finish-install actions, the co-installer calls **SetupDiGetDeviceInstallParams** to retrieve the device installation parameters for the device and then calls **SetupDiSetDeviceInstallParams** to set the **Flags** member of the **SP\_DEVINSTALL\_PARAMS** structure for the device with the DI\_NEEDREBOOT flag. The installer also notifies the user that a system restart is required.
  - If the finish-install actions fail and the finish-install actions should be attempted again the next time the device is enumerated, the co-installer notifies the user of this situation.

**Note** Starting in Windows 8 a finish-install action is only run once. Windows will not automatically run it again, especially not the next time the device is enumerated because that is not when finish-install actions are run.

- If the finish-install actions fail and the co-installer determines that the finish-install actions cannot succeed, the co-installer notifies the user of this situation.
- By default, the co-installer returns NO\_ERROR in response to a **DIF\_FINISHINSTALL\_ACTION** request if the finish-install actions succeeded, or if the finish-install actions failed and the co-installer determine that the finish-install actions should not be attempted again. The co-installer returns a Win32 error code only if the finish-install actions fail and the finish-install actions should be attempted again the next time the device is enumerated in the context of an administrator.

**Note** Starting in Windows 8 a finish-install action is only run once. Windows will not automatically run it again, especially not the next time the device is enumerated because that is not when finish-install actions are run.

The following co-installer code example shows the basic structure of co-installer code that implements finish-install actions:

```

DWORD CALLBACK
SampleCoInstaller(
    IN DI_FUNCTION InstallFunction,
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData,
    IN OUT PCOINSTALLER_CONTEXT_DATA Context
)
{
    SP_DEVINSTALL_PARAMS DeviceInstallParams;
    DWORD ReturnValue = NO_ERROR; // The default return value

    switch(InstallFunction)
    {
        case DIF_NEWDEVICEWIZARD_FINISHINSTALL:
            //
            // Processing for finish-install wizard pages
            //
            // Processing for finish-install actions
            if (FinishInstallActionsNeeded())
            {
                // Obtain the device install parameters for the device
                // and set the DI_FLAGSEX_FINISHINSTALL_ACTION flag
                DeviceInstallParams.cbSize = sizeof(DeviceInstallParams);
                if (SetupDiGetDeviceInstallParams(DeviceInfoSet, DeviceInfoData, &DeviceInstallParams))
                {
                    DeviceInstallParams.FlagsEx |= DI_FLAGSEX_FINISHINSTALL_ACTION;
                    SetupDiSetDeviceInstallParams(DeviceInfoSet, DeviceInfoData, &DeviceInstallParams);
                }
            }
            break;

        case DIF_FINISHINSTALL_ACTION:
            if (FinishInstallActionsNeeded())
            {
                //
                // Perform the finish-install actions,
                // notify the user that finish install actions
                // are in progress and wait for
                // the finish-install actions to complete
                //
                // If the finish-install actions succeed, notify the user
                //
                // If the finish install actions require a system restart:
                // notify the user, call SetupDiGetDeviceInstallParams
                // to obtain the device install parameters for the device in
                // DeviceInstallParams, and call SetupDiSetInstallParams to set
                // the DI_NEEDREBOOT flag in DeviceInstallParams.Flags
                //
                // If the finish install actions failed, but
                // should be attempted again: clean up,
                // notify the user of the failure, and
                // set ReturnValue to an appropriate Win32 error code
                //
                // If the finish install actions failed and
                // should not be attempted again: clean up
                // and notify the user of the failure
                //
                // Starting with Windows 8, a finish-install action
                // is only run once. Windows will not automatically
                // run it again, especially not the next time
                // the device is enumerated because that is not when
                // finish-install actions are run.
                //
            }
            break;
    }

    return ReturnValue;
}

```



# Overview of Device Property Pages

11/2/2020 • 2 minutes to read • [Edit Online](#)

A *device property page* is a window that allows the user to view and edit the properties for a device. For most devices, the operating system provides standard device property pages that allow the user to view and edit a common set of parameters for that device. For more information about how property pages are displayed for a device, see [How Device Property Pages are Displayed](#).

Independent hardware vendors (IHVs) typically provide custom device property pages that allow the user to view and edit additional and proprietary properties for a device. These properties are specific to each device that the IHV supplies. These properties might include default playback volume for a CD drive or speaker volume for a modem.

An IHV creates a custom device property page by using a property page provider. A property page provider can be one of the following:

## Class Installers and Co-installers

A co-installer or class installer can provide one or more custom device property pages by supporting the `DIF_ADDPROPERTYPAGE_ADVANCED` device installation function (DIF) code.

## Property Page Extension DLL

A dynamic-link library (DLL) that provides one or more custom device property pages is referred to as a *property page extension DLL*. This type of provider supports custom property pages by implementing the `AddPropSheetPageProc`, `ExtensionPropSheetPageProc`, and other property sheet callback functions.

For more information about these functions, see the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0.

An IHV should supply a provider of custom device property pages in its driver package if its device or device class has any individual properties that a user can set.

**Note** In versions of Windows earlier than Windows 2000, users set such information in Control Panel. Driver software that is written for Windows 2000 and later versions of Windows should provide property pages instead.

For more information about property page providers, see [Types of Device Property Page Providers](#).

The Windows SDK for Windows 7 and .NET Framework 4.0 documentation provides comprehensive guidance about property pages and the Microsoft Win32 functions that manipulate them. For more information about property pages and property sheets, see [Property Sheet](#) in the Windows SDK for Windows 7 and .NET Framework 4.0 documentation.

# How Device Property Pages are Displayed

12/5/2018 • 2 minutes to read • [Edit Online](#)

A device property page is displayed in the following ways:

- Device Manager displays the property page any time that the properties for a device are displayed.
- Starting with Windows 7, **Devices and Printers** displays the property page any time that a user right-clicks a device. To display the property page in **Devices and Printers**, do the following:
  - Right-click the icon in **Devices and Printers** that represents a device that is connected to the system.
  - Click **Properties**.
  - On the **Properties** window, click the **Hardware** tab, and then click **Properties**.

# Types of Device Property Page Providers

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can supply custom device property pages by using the following types of property page providers:

- **Class Installers and Co-installers.**

A [co-installer](#) can provide one or more custom device property pages by supporting the [DIF\\_ADDPROPERTYPAGE\\_ADVANCED](#) device installation function (DIF) code. 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 co-installer that is part of the Toaster sample in the Windows Driver Kit (WDK) supports this type of device property page provider. It is located in the `src\general\toaster\classinstaller` subdirectory of the WDK.

For more information about the requirements for this type of provider, see [Specific Requirements for Device Property Page Providers \(Co-Installers\)](#).

**Note** Although you can write a class installer that provides custom device property pages, it is generally better to provide this functionality in a co-installer, together with other device-specific or device-class-specific features.

- **Property Page Extension DLL.**

A DLL that provides one or more custom device property pages is referred to as a *property page extension DLL*. This type of provider supports custom property pages by implementing the [AddPropSheetPageProc](#), [ExtensionPropSheetPageProc](#), and other property sheet callback functions. For more information about these functions, see the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0 documentation.

This type of provider is installed by specifying an [EnumPropPages32](#) entry in the *add-registry-section* of an [INF AddReg directive](#). This directive is specified within a [INF DDInstall section](#).

The AC97 sample audio driver supports this type of device property page provider. It is located in the `src\audio\ac97` subdirectory of the WDK.

For more information about the requirements for this type of provider, see [Specific Requirements for Device Property Page Providers \(Property Page Extension DLLs\)](#).

**Note** Unless your [driver package](#) requires a class installer or co-installer, it is more efficient to support custom device property pages by using a property page extension DLL.

All types of device property page providers must follow the guidelines described in [General Requirements for Device Property Page Providers](#).

# General Requirements for Device Property Page Providers

12/5/2018 • 2 minutes to read • [Edit Online](#)

To create device property pages, a provider must follow these general requirements in order for the property page to work correctly:

- Handle the request to add a property page.

For providers that are property page extension DLLs, this request is made through the **AddPropSheetPageProc** callback function. For more information, see [Specific Requirements for Device Property Page Providers \(Property Page Extension DLLs\)](#)

- Create the property page by calling the **CreatePropertySheetPage** function. The provider passes the address of the initialized PROPSHEETPAGE structure in this call.
- Supply a **PropSheetPageProc** callback function 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.
- Initialize a PROPSHEETPAGE structure with (among other things) the addresses of the **PropSheetPageProc** callback function and dialog box procedure.

This section includes the following topics that provide more guidance about custom property pages:

[Creating Custom Property Pages](#)

[Property Page Callback Function](#)

[Handling Windows Messages for Property Pages](#)

[Sample Custom Property Page](#)

# Creating Custom Property Pages

11/2/2020 • 2 minutes to read • [Edit Online](#)

When a [device property page provider](#) handles a request to create a property page for its device or device class, the provider 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 )) {  
    ...
```

In this example, the code zero-initializes the structure in which the installation parameters of the device 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.

2. Make sure that the maximum number of dynamic pages for the device has not yet been met by using a statement such as the following:

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

If the test fails, do not initialize or create the page. Instead, 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 memory with the data. The provider must release this memory in its property page callback when the property page is destroyed.

For providers that are [co-installers](#), this device-specific data must include the *DeviceInfoSet* and *DeviceInfoData* passed with the [DIF\\_ADDPROPERTYPAGE\\_ADVANCED](#) device installation function (DIF) code.

For example, a property page provider can define and use a structure as shown in the following example:

```
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 that are 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 the callback, process the PSPCB\_RELEASE message and free the memory that was allocated in step 3.

- In **pfnDigProc**, set a pointer to the dialog box procedure for the property page.

- In **IParam**, pass a pointer to the memory area that was allocated and initialized in step 3.

- Set other members as appropriate for the custom property page. See the Microsoft Windows SDK documentation for more information about 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.

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

# Property Page Callback Function

12/5/2018 • 2 minutes to read • [Edit Online](#)

When a provider creates a property page for its device or device class, it supplies a pointer to a callback function. The callback function is called one time when the property page is created and again when it is about to be destroyed.

The callback is a `PropSheetPageProc` function that is described in the Windows SDK documentation. This function 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 that is associated with the page. The function should return `TRUE` to continue to create the page or `FALSE` if 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 destroyed, the callback is called with a `PSPCB_RELEASE` message. The function should free any data that was allocated when the property page was created. Typically, this involves freeing the data referenced by the `IParam` member of the `PROPSHEETPAGE` structure. The return value is ignored when the page is being destroyed.

# Handling Windows Messages for Property Pages

11/2/2020 • 3 minutes to read • [Edit Online](#)

When a [device property page provider](#) handles a request to create a property page for its device or device class, it returns 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 Microsoft Windows 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 that is 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 provider. For additional functions that manipulate icons and bitmaps see [Device Installation Functions](#). For more information about **SendDlgItemMessage**, **DestroyIcon**, and using icons in dialog boxes, see the Windows SDK documentation.

In addition to an icon that represents 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 Plug and Play (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 **SetupDiXxx** functions to extract that information for display. For more information, see [Device Installation Functions](#).

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.

For more information about the notifications that a dialog box procedure might encounter, such as the PSN\_APPLY or PSN\_HELP notifications, and how the procedure should handle them, see [Notifications](#) in the Windows SDK documentation.

## PSN\_APPLY Notifications

The property sheet sends a PSN\_APPLY notification message when the user clicks **OK**, **Close**, or **Apply**. In response to this message, the dialog box procedure should validate and apply the changes that were made by the

user.

When it receives the PSN\_APPLY notification, the provider must do the following:

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 referenced by the **IParam** member of the PROPSHEETPAGE structure.
  2. Ensure that the user's changes are valid.
  3. If the provider allows a user to set a property that requires Windows to remove and restart the device, the provider must set the DI\_FLAGSEX\_PROPCHANGE\_PENDING flag in the **FlagsEx** field of the returned [SP\\_DEVINSTALL\\_PARAMS](#) structure.
- However, if the provider can ensure that the changes do not require the device's drivers to be stopped and then restarted, it does not have to set this flag.
4. Call [SetupDiSetDeviceInstallParams](#) with the changed [SP\\_DEVINSTALL\\_PARAMS](#) structure to set the new parameters.

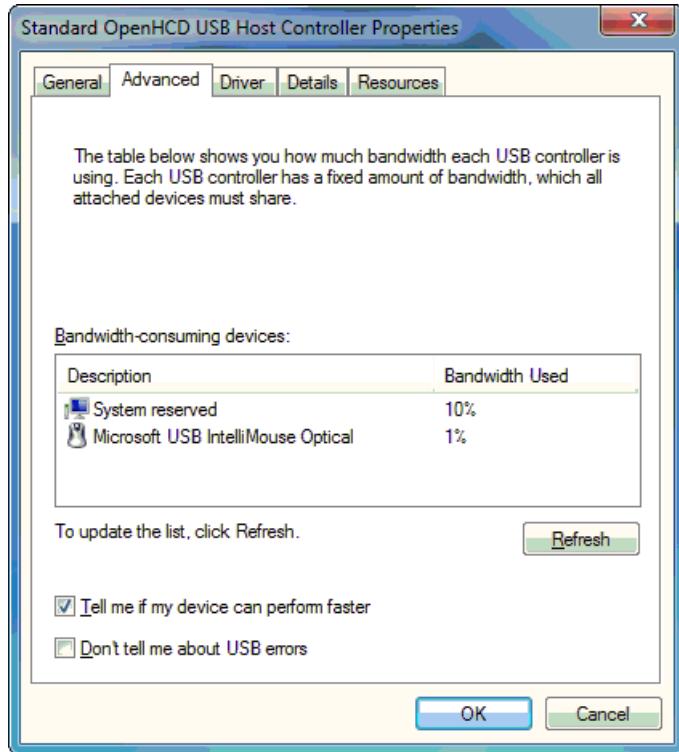
### **PSN\_RESET Notifications**

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 that were made by the user.

# Sample Custom Property Page

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following dialog box contains a sample custom property page:



In the previous dialog box, a provider supplies a property page named Properties. By default, the system supplies the General, Driver, and Details tabs shown in the dialog box. When appropriate, the system also provides Resources and Power tabs.

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

## Related information

- [Specific Requirements for Device Property Page Providers](#)
- [Types of Device Property Page Providers](#)

# Specific Requirements for Device Property Page Providers (Co-Installers)

11/2/2020 • 2 minutes to read • [Edit Online](#)

A [co-installer](#) that provides one or more custom device property pages must handle the [DIF\\_ADDPROPERTYPAGE\\_ADVANCED](#) device installation function (DIF) code. Device Manager issues this request when a user clicks on the **Properties** tab of a device in Device Manager or in Control Panel.

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 does this by initializing and returning an [SP\\_ADDPROPERTYPAGE\\_DATA](#) structure for the class installation parameters of the request.

If the user changes any properties, Device Manager sends a [DIF\\_PROPERTYCHANGE](#) DIF code to the installer after the installer sets the new parameters by calling [SetupDiSetDeviceInstallParams](#).

For more information about how to create a custom device property page by a [co-installer](#), see [General Requirements for Device Property Page Providers](#).

# Specific Requirements for Device Property Page Providers (Property Page Extension DLLs)

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic how to create and install a property page extension DLL.

## Creating a property page extension dll

A property page extension DLL that provides custom property pages must handle the request to add a property page. This request is made through the **AddPropSheetPageProc** callback function.

In response to this request, the DLL 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.

For information about how to create a custom device property page by a property page extension DLL, see [General Requirements for Device Property Page Providers](#).

## Installing a device property page

A property page extension DLL is installed by using the following directives in the **INF file** of a **driver package**:

1. Use the *add-registry-section*, which is specified by an [INF AddReg directive](#) in the [INF DDInstall section](#), to add an **EnumPropPages32** entry for the device. The **EnumPropPages32** entry specifies the following [REG\\_SZ](#) values:

- The name of the DLL that exports the **ExtensionPropSheetPageProc** callback function.
- The name of the **ExtensionPropSheetPageProc** callback function as implemented by the DLL.

The following code example shows an *add-registry-section* that adds the **EnumPropPages32** entry that specifies the name of the DLL (*MyPropProvider.dll*) and callback function (*MyCallbackFunction*):

```
HKR, , EnumPropPages32, 0, "MyPropProvider.dll, MyCallbackFunction"
```

**Important** Both the name of the DLL and callback function must be enclosed together within quotation marks ("").

2. Include an [INF CopyFiles directive](#) that copies the property page extension DLL to the **%SystemRoot%\System32** directory.
3. If the device is a network adapter, you must specify **NCF\_HAS\_UI** as one of the **Characteristics** values in the [INF DDInstall section](#). This value indicates that the adapter supports a user interface.

For more information, see [Specifying Custom Property Pages for Network Adapters](#).

# Invoking a Device Properties Dialog Box

12/21/2018 • 2 minutes to read • [Edit Online](#)

This topic describes how to invoke a device properties dialog box programmatically in an installation application or manually from a command prompt by using the **DeviceProperties\_RunDLL** function provided by Device Manager.

For information about how to call **DeviceProperties\_RunDLL**, see the following topics:

[Invoking a Device Properties Dialog Box Programmatically in an Installation Application](#)

[Invoking a Device Properties Dialog Box from a Command-line Prompt](#)

[DeviceProperties\\_RunDLL Function Prototype](#)

For more information about invoking **DeviceProperties\_RunDLL**, see the [Knowledge Base article 815320](#) on the Microsoft Help and Support website.

# Invoking a Device Properties Dialog Box Programmatically in an Installation Application

12/5/2018 • 2 minutes to read • [Edit Online](#)

To invoke the device properties dialog box programmatically in an installation application, the application code should do the following:

- Include macro definitions and type definitions in the application code that ensure that the appropriate version of [DeviceProperties\\_RunDLL](#) is linked when the application is built. Windows supports a Unicode version and an ASCII version.
- Load *devmgr.dll*.
- Obtain a pointer to the [DeviceProperties\\_RunDLL](#) function.
- Call [DeviceProperties\\_RunDLL](#), supplying the appropriate parameters.

The following code example shows how to define a *pDeviceProperties* function pointer that references the [DeviceProperties\\_RunDLL](#) function. If the `_UNICODE` macro is defined when the code is complied, *pDeviceProperties* is a pointer to the Unicode version of the function; otherwise *pDeviceProperties* is a pointer to the ASCII version of the function.

```
#ifdef _UNICODE
#define DeviceProperties_RunDLL "DeviceProperties_RunDLLW"
typedef void (_stdcall *PDEVICEPROPERTIES)(
    HWND hwndStub,
    HINSTANCE hAppInstance,
    LPWSTR lpCmdLine,
    int nCmdShow
    );
#else
#define DeviceProperties_RunDLL "DeviceProperties_RunDLAA"
typedef void (_stdcall *PDEVICEPROPERTIES)(
    HWND hwndStub,
    HHINSTANCE hAppInstance,
    LPSTR lpCmdLine,
    int nCmdShow
    );
#endif

PDEVICEPROPERTIES pDeviceProperties;
```

The following code example uses the definition of *pDeviceProperties* that was provided in the preceding code example and shows how to load *devmgr.dll* programmatically and how to obtain the function pointer to the appropriate version of [DeviceProperties\\_RunDLL](#).

```
HINSTANCE hDevMgr = LoadLibrary(_TEXT("devmgr.dll"));
if (hDevMgr) {
    pDeviceProperties = (PDEVICEPROPERTIES)GetProcAddress((HMODULE)hDevMgr, DeviceProperties_RunDLL);
}
```

After obtaining a *pDeviceProperties* function pointer, you must supply a computer name and [device instance identifier](#) in a call to [DeviceProperties\\_RunDLL](#). The following code example illustrates the appropriate format and requirements for these items:

- (Windows XP and later) An optional *machine-name-parameter* field is not supplied, which indicates, by default, that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the device instance identifier "root\system\0000".

```
if (pDeviceProperties){
    pDeviceProperties(m_hWnd, NULL, _TEXT("/DeviceID root\\system\\0000"), NULL);
};
```

- (Windows XP and later) An optional *machine-name-parameter* field is not supplied, which indicates, by default, that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the device instance identifier "PCI\VEN\_8086&DEV\_2445&SUBSYS\_010E1028&REV\_12\3&172E68DD&0&FD".

```
if (pDeviceProperties){
    pDeviceProperties(m_hWnd, NULL, _TEXT("/DeviceID
PCI\\VEN_8086&DEV_2445&SUBSYS_010E1028&REV_12\\3&172E68DD&0&FD"), NULL);
};
```

- (Windows 2000 and later) A required *machine-name-parameter* field supplies a pair of double quotation marks ("") for *machine-name*, which indicates that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the device instance identifier "root\system\0000".

```
if (pDeviceProperties){
    pDeviceProperties(m_hWnd, NULL, _TEXT("/MachineName \"\" /DeviceID root\\system\\0000"), NULL);
};
```

- (Windows 2000 and later) A required *machine-name-parameter* field supplies the remote machine name "\\RemoteMachineAbc" and a required *device-instance-ID-parameter* field supplies the device instance identifier "root\system\0000".

```
if (pDeviceProperties){
    pDeviceProperties(m_hWnd, NULL, _TEXT("/MachineName \\\\RemoteMachineAbc /DeviceID
root\\system\\0000"), NULL);
};
```

# Invoking a Device Properties Dialog Box from a Command-line Prompt

12/5/2018 • 2 minutes to read • [Edit Online](#)

The [DeviceProperties\\_RunDLL](#) function in Device Manager can be invoked from a command-line prompt using `rundll32.exe`. The following code example demonstrates the format for invoking `DeviceProperties_RunDLL` from a command prompt.

```
rundll32.exe devmgr.dll, DeviceProperties_RunDLL machine-name-parameter device-instance-ID-parameter
```

The format and requirements for the *machine-name-parameter* field is the same as that described for the command-line string supplied by the */lpCmdLine* parameter of `DeviceProperties_RunDLL`. The format and requirements for the *device-instance-ID-parameter* field is also the same as that described for the */lpCmdLine* command-line string, subject to the following additional requirement: if the *device-instance-ID* subfield includes an ampersand (&), the *device-instance-ID* subfield must be enclosed in quotation marks ("").

The following code examples illustrate the format and requirements for supplying a *machine-name-parameter* and *device-instance-ID-parameter* to invoke `DeviceProperties_RunDLL` from a command-line prompt. These examples correspond to the examples provided in [Invoking a Device Properties Dialog Box Programmatically in an Installation Application](#).

- (Windows XP and later) An optional *machine-name-parameter* field is not supplied, which indicates, by default, that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the **device instance identifier** "root\system\0000".

```
rundll32.exe devmgr.dll,DeviceProperties_RunDLL /DeviceID root\system\0000
```

- (Windows XP and later) An optional *machine-name-parameter* field is not supplied, which indicates, by default, that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the device instance identifier "PCI\VEN\_8086&DEV\_2445&SUBSYS\_010E1028&REV\_12\3&172E68DD&0&FD". Because the device instance identifier includes an ampersand (&), the *device-instance-ID* subfield is enclosed in quotation marks ("").

```
rundll32.exe devmgr.dll,DeviceProperties_RunDLL /DeviceID  
"PCI\VEN_8086&DEV_2445&SUBSYS_010E1028&REV_12\3&172E68DD&0&FD"
```

- (Windows 2000 and later) A required *machine-name-parameter* field supplies a pair of quotation marks ("") as *machine-name*, which indicates that the computer is the local computer. A required *device-instance-ID-parameter* field supplies the device instance identifier "root\system\0000".

```
rundll32.exe devmgr.dll,DeviceProperties_RunDLL /MachineName "" /DeviceID root\system\0000
```

- (Windows 2000 and later) A required *machine-name-parameter* field supplies the remote machine name "\\\RemoteMachineAbc". A required *device-instance-ID-parameter* field supplies the device instance identifier "root\system\0000".

```
rundll32.exe devmgr.dll,DeviceProperties_RunDLL /MachineName \\RemoteMachineAbc /DeviceID  
root\SYSTEM\0000
```

# DeviceProperties\_RunDLL Function Prototype

12/5/2018 • 2 minutes to read • [Edit Online](#)

The DeviceProperties\_RunDLL function opens the device properties dialog box for a specified device that is installed on a local or remote computer.

```
void DeviceProperties_RunDLL(
    HWND      hwndStub,
    HINSTANCE hAppInstance,
    LPCTSTR   lpCmdLine,
    int       nCmdShow
);
```

## Parameters

### *hwndStub*

A handle to the window in which to display the user interface items that DeviceProperties\_RunDLL creates.

### *hAppInstance*

This parameter is not used to invoke a device properties dialog box and should be set to **NULL**.

### *lpCmdLine*

A pointer to a constant NULL-terminated command-line string that contains a *machine-name-parameter* field followed by a *device-instance-ID-parameter* field in the following format:

*machine-name-parameter device-instance-ID-parameter*

### *machine-name-parameter*

The *machine-name-parameter* field supplies the name of the machine that is associated with the device that is specified by the *device-instance-ID-parameter* field. The format of the *machine-name-parameter* field is

**/MachineName \*\*\*\* machine-name**, where **/MachineName** indicates that *machine-name* supplies a computer name. For more information about the *machine-name-parameter* field, see the Remarks section.

### *device-instance-ID-parameter*

The *device-instance-ID-parameter* field supplies a **device instance identifier** of the device for which to display a device properties dialog box. The format of *device-instance-ID-parameter* field is **/DeviceId \*\*\*\* device-instance-ID**, where **/DeviceId** indicates that *device-instance-ID* supplies a device instance identifier. The *device-instance-ID-parameter* field is required.

### *nCmdShow*

This parameter is not used to invoke a device properties dialog box and should be set to **NULL**.

## Return Value

None

## Headers

DeviceProperties\_RunDLL is not declared in a public header and can only be invoked indirectly by programmatically obtaining a pointer to the function or by using rundll32.

## Remarks

On Windows XP, the *machine-name-parameter* field is required only for a remote computer, and, if the *machine-name-parameter* field is not supplied, the local computer is used by default. On Windows 2000, the *machine-name-parameter* field is required for a local computer or a remote computer. To specify a local computer, set

*machine-name* in the *machine-name-parameter* field to a pair of quotation marks (""). If the machine is a remote computer, set *machine-name* to a valid computer name. A valid computer name must include a prefix that consists of a pair of backslashes(\) followed by the machine name.

The following are examples of command-line strings:

- (Windows XP and later) Specifying the local computer is optional, in which case the command-line string is required to include only the device instance identifier. For example, the following command-line specifies the local computer by default and the device instance identifier "root\system\0000".

```
/DeviceId root\system\0000
```

- (Windows 2000 and later) The following command-line string supplies the remote computer name "\\\RemoteMachineAbc" and the device instance identifier "root\system\0000".

```
/MachineName \\\RemoteMachineAbc /DeviceId root\system\0000
```

- (Windows 2000 and later) The following command-line string specifies a local computer, which is specified by a pair of quotation marks (""), and supplies the device instance identifier "root\system\0000".

```
/MachineName "" /DeviceId root\system\0000
```

# Writing a Device Installation Application

12/1/2020 • 2 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

If your driver package includes drivers and INF files that replace "inbox" 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 and distribution medium should be compatible with AutoRun, so that AutoRun starts the application automatically when a user inserts your distribution medium. For more information about AutoRun, see [Creating an AutoRun-Enabled Application](#).

For guidelines about how to write a device installation application, see [Guidelines for Writing Device Installation Applications](#).

Your [driver package](#) must handle two situations:

1. The user plugs in your hardware before inserting your distribution medium. This is commonly referred to as a [hardware-first installation](#).
2. The user inserts your distribution medium before plugging in your hardware. This is commonly referred to as a [software-first installation](#).

# Guidelines for Writing Device Installation Applications

12/1/2020 • 2 minutes to read • [Edit Online](#)

*Device installation applications* must do the following:

- Support removal of all device-specific applications that they install. As part of that uninstall process, the device installation application should check whether 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](#).
- Starting with Windows Vista, list all the applications that were installed by using Microsoft Windows Installer (MSI), and that are available in **Programs and Features** in Control Panel. You can then uninstall these items if necessary.
- In versions of Windows earlier than Windows Vista, list all the applications that were installed by using Microsoft Windows Installer (MSI), and that are available in **Add Or Remove Programs** in Control Panel. You can then uninstall these items if necessary.
- Follow the guidelines for Microsoft Windows applications. See the [Microsoft Developer Network](#) website for more information.

*Device installation applications* can do the following:

- [Install device-specific applications](#)

**Note** We highly recommend that you submit device-specific applications to the appropriate [Hardware Certification Kit \(HCK\)](#) for software. See the [Microsoft Developer Network](#) website for more information.

- [Preload driver packages](#)
- [Preinstall driver packages](#)

*Device installation applications* must not do the following:

- Instruct the user to copy or overwrite any files, especially `.inf` and `.sys` files.
- Delete the installed driver files from the system during the uninstall operation, even if the hardware is removed.
- Force any unnecessary system restarts. Restarts are generally not required for installing PnP devices or software applications. The `bRebootRequired` parameter of the [UpdateDriverForPlugAndPlayDevices](#) function indicates the need for a restart.
- Use RunOnce registry keys to start *device installation applications*, because this requires a system restart.
- Use a device or class co-installer, or a class installer, to start 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 during a server-side installation, the system will stop responding.

After the device is installed, device installation applications can be started safely by using [finish-install actions](#) that are supplied by class installers or co-installers.

For more information about co-installers, see [Writing a Co-installer](#).

- Use the Startup Group to start *device installation applications*.

- Use *win.ini* entries to start device installation applications.
- 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 inbox application does not support such a capability.

# Installing Device-Specific Applications

11/2/2020 • 2 minutes to read • [Edit Online](#)

If your distribution medium includes device-specific applications, you can use the following methods to install those applications:

- Use a device co-installer to install the applications by using [finish-install actions](#).

If the user plugs in the device before inserting the distribution medium, this is referred to as a [hardware-first installation](#). If the device is not supported by inbox drivers, Windows calls the co-installer that is supplied by the medium during the installation process.

The co-installer should determine whether the device-specific applications have already been installed. If they have not, the co-installer should do one of the following

1. Start a *device installation application* on the distribution medium to install the device-specific applications.
2. Prompt the user to download a newer version of the device installation application from the Internet.

Independent hardware vendors (IHVs) can use various methods to provide [hardware-first installation](#) solutions for installing device-specific applications. For more information about these methods, see [Hardware-First Installation of Device-Specific Applications](#).

For more information about co-installers, see [Writing a Co-installer](#).

- Use 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, this is referred to as a [software-first installation](#). The medium's AutoRun-invoked device installation application should determine whether the device-specific applications have already been installed and if they have not, it should install them by using Windows Installer. For more information, see the Windows SDK documentation.

# Device Installation Application Not Included in the Driver Package

12/1/2020 • 2 minutes to read • [Edit Online](#)

This method describes a way through which a co-installer, by using [finish-install actions](#), can start a *device installation application* to install device-specific applications.

In this method, the device installation application is not part of the [driver package](#), and the driver package's INF file is not used to copy this file to the user's hard drive. Instead, the co-installer starts the device installation application directly from the distribution medium, or prompts the user to download the device installation application from the Internet.

This method has the following advantages:

- If the co-installer supplies [finish-install actions](#), this method can be used to install [driver packages](#) and device-specific applications on Windows Vista and later versions of Windows.
- Only the driver package must be digitally signed. The device installation application and associated installation files do not have to be digitally signed. For more information about digital signatures, see [Driver Signing](#).
- Only the driver package is copied to the driver store. When the device installation application is launched, the device-specific applications are installed elsewhere on the user's hard drive.
- The user can be prompted during [driver package](#) updates to update the device-specific applications. This occurs through the [finish-install actions](#) supplied by the package's co-installer. In this way, synchronizing versions of the driver package and device-specific applications is optional. Also, additional device-specific applications, which are not on the distribution medium, can be downloaded from the Internet.
- The independent hardware vendor (IHV) has greater flexibility with the distribution medium. The device installation application and device-specific applications can be on the medium or can be downloaded from the Internet.

If you use this method, the following will occur whenever the user installs the device before inserting the distribution medium, or Windows Updates detects a new driver for the device:

1. The [driver package](#) for the device is installed as described in [hardware-first installation](#).
2. If the driver package's co-installer supplies [finish-install actions](#), one of the following occurs at the end of the driver package installation:
  - The device installation application on the distribution medium is launched to install the device-specific applications.
  - The user is prompted to download a newer version of the device installation application from the Internet. Through this, the IHV can provide additional device-specific applications which are not on the distribution medium.

The co-installer then starts the device installation application as soon as it is downloaded from the Internet.

**Note** Since the [driver package](#) has already been installed before the device installation application is launched, the application must detect that the drivers are already installed and only install the device-specific applications.

For more information about co-installers, see [Writing a Co-installer](#).

For more information about starting device installation application through co-installers, see [Guidelines for Starting Device Installation Applications through Co-installers](#).

# Device Installation Application that is Included in the Driver Package

12/1/2020 • 2 minutes to read • [Edit Online](#)

This method describes a way through which a co-installer, using [finish-install actions](#), can start a *device installation application* to install device-specific applications.

In this method, the device installation application and associated installation files are part of the [driver package](#), and the driver package's INF file is used to copy the application and installation files to the driver store. As a result, the application and driver are both installed in all scenarios.

This method has the following advantages:

- If the co-installer supplies [finish-install actions](#), this method can be used to install driver packages, including device-specific applications, on Windows Vista and later versions of Windows.
- Since the device installation application, associated installation files, and driver are part of the same driver package, they will be installed or updated at the same time. This synchronizes the versions of all components in the driver package.

However, this method also has the following disadvantages:

- The device installation application and associated installation files must be digitally signed if it is part of the [driver package](#). For more information about digital signatures, see [Driver Signing](#).

In addition, the driver, device installation application, and associated installation files must be submitted to Windows Hardware Quality Labs (WHQL) as a single driver package whenever these components are modified. For more information about this process, see [WHQL release signatures](#).

- The device installation application and associated installation files are copied to two places on the user's hard drive: the driver store as well as the directory that is specified in the INF file's [INF DestinationDirs Section](#).
- Optional device-specific applications, which are not installed through the device installation application, cannot be installed from the distribution medium or downloaded from the Internet.

Since the device installation application and associated installation files are signed as part of the [driver package](#), only applications installed through the device installation application are allowed in order to ensure the integrity of the driver package.

If you use this method, the following will occur whenever the user installs the device before inserting the distribution medium, or Windows Updates detects a new driver for the device:

1. The driver package for the device is installed as described in [hardware-first installation](#).
2. The [driver package's](#) INF file copies the device installation application and associated installation files to a directory on the user's hard drive, which is typically the Program Files directory. This is done as part of the processing of the [INF CopyFiles directive](#) in the INF file.
3. If the driver package's co-installer supplies [finish-install actions](#), the co-installer starts the device installation application on the distribution medium to install the device-specific applications.

**Note** Since the [driver package](#) has already been installed before the device installation application is launched, the application must detect that the drivers are already installed and only install the device-specific applications.

For more information about co-installers, see [Writing a Co-installer](#).

For more information about starting device installation applications through co-installers, see [Guidelines for Starting Device Installation Applications through Co-installers](#).

# Device Installation Application Started through AutoRun

1/11/2019 • 2 minutes to read • [Edit Online](#)

This method builds upon existing [software-first installation](#) methods to create a [hardware-first installation](#) scenario. This method relies on the [INF HardwareId directive](#), which is supported in Windows Vista and later versions of Windows.

In this method, the *device installation application* on the distribution medium is launched as part of the processing of the Autorun.inf file. When the user plugs in the device, the Found New Hardware Wizard parses this file, and looks for an [INF HardwareId directive](#) that matches the device that is being installed. If the wizard finds a match, it invokes the AutoRun-enabled device installation application to install the [driver package](#) and device-specific applications.

This method has the following advantages:

- Independent hardware vendors (IHVs) can easily use this method to an existing AutoRun-compatible distribution medium by adding one or more [INF HardwareId directives](#) to the Autorun.inf file. For more information about and AutoRun and AutoRun-enabled applications, see [Creating an AutoRun-Enabled Application](#).
- Only the [driver package](#) must be digitally signed. The device installation application and associated installation files do not have to be digitally signed. For more information about digital signatures, see [Driver Signing](#).
- Only the driver package is copied to the [driver store](#). The device-specific applications are installed elsewhere on the user's hard drive.
- The user can be prompted during driver package updates to update the device-specific applications. This occurs through the [finish-install actions](#) supplied by the package's co-installer. In this way, updates to the driver package and device-specific applications can be synchronized. Also, additional or optional applications, which are not on the distribution medium, can be downloaded from the Internet.

However, this method also has the following disadvantages:

- This method can be used for installing [driver package](#) and device-specific application installation only on Windows Vista and later versions of Windows.
- The distribution media must be AutoRun-compatible, such as a CD or DVD. The [INF HardwareId directive](#) does not provide any capability for downloading an application installer from the Internet.

# Guidelines for Starting Device Installation Applications through Co-installers

12/1/2020 • 2 minutes to read • [Edit Online](#)

The following guidelines must be followed for co-installers which supply finish-install actions (Windows Vista and later versions of Windows) to start *device installation applications*:

- A co-installer must not exit from its finish-install pages or finish-install action until the device installation application has finished. If a co-installer exits early and another driver requires a restart, Windows might restart the computer before the application has completed.
- A co-installer should first look for the device installation application on the distribution medium and, if the application is present, silently begin the installation process.

If the distribution medium is not present, the co-installer should offer the user the choice of either inserting a medium or downloading the device installation application from the Internet. If the user chooses to insert the medium, the co-installer should detect the new media notification, silently exit, and allow the medium's AutoRun process to start the device installation application.

A co-installer can detect the new media by listening for a WM\_DEVICECHANGED/DBT\_DEVICEARRIVAL message with dbch\_devicetype set to DBT\_DEVTYPE\_VOLUME and dbcv\_flags set to DBTF\_MEDIA.

For more information, see [Detecting Media Insertion or Removal](#).

- A co-installer should never assume that the distribution media is available during installation. For example, a co-installer should never use the %1% DirId to find the media from within the co-installer.

In Windows Vista and later versions of Windows, the co-installers are invoked after they are copied to the driver store. By this time, the distribution media might be removed, which is why it is important to avoid prompting the user for the medium. For example, suppose that a user performs a [software-first installation](#), removes the distribution medium, and then plugs in the device. The co-installer is invoked only when the user plugs in the device for the first time.

- If a co-installer starts Windows Internet Explorer to download the device installation application, it must start it in Protected Mode, which has features that protect users against malicious code.

For more information about starting Internet Explorer in Protected Mode, see [Understanding and Working in Protected Mode Internet Explorer](#).

For more information about co-installers, see [Writing a Co-installer](#).

# Preloading Driver Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

Plug and Play (PnP) [driver packages](#) can be *preloaded* on a computer as part of a Windows installation or after Windows is installed on a computer. A network administrator can also preload driver packages on a network server that provides the source for driver packages that are installed on network computers. When Windows searches for drivers that match a device, Windows will check whether there are preloaded driver packages that match the device.

How to configure a Windows installation to preload driver packages is outside the scope of this documentation. For information about how to configure a Windows installation to preload driver packages, see [How to Add OEM Plug and Play Drivers to Windows XP](#) and [How to Add OEM Plug and Play Drivers to Windows Installations](#).

After Windows is installed, a [driver package](#) can be preloaded in one of the following ways:

1. To preload a driver package on a local computer, copy the driver package to a package-specific directory on a local computer and concatenate the local directory path of the driver package to the **DevicePath** value entry under the **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion** subkey of the registry.
2. To preload a driver package for a computer network, a network administrator can copy the driver package to a shared directory on a network server and concatenate the path of the shared directory to the **DevicePath** value entry in the registry of the network computers that have access to the shared directory.

The **DevicePath** value entry is a [REG\\_EXPAND\\_SZ](#)-typed entry that contains the `%SystemRoot%\inf` directory path and zero or more directory path entries. The format of the **DevicePath** value entry is the following, where each directory path is either a local directory path or a path of a shared directory on a network server where the preloaded [driver packages](#) are located:

```
%SystemRoot%\inf;DirectoryPath1;DirectoryPath2;...
```

For example, to preload a driver package for a network adapter in the `%SystemRoot%\Drivers\NIC` directory on a local computer, an administrator copies the driver package to that directory and concatenates the path of the **DevicePath** value entry, as follows:

```
%SystemRoot%\inf;...;%SystemRoot%\inf\Drivers\NIC
```

For example, to preload a driver package for a network adapter in the shared directory `\DriverPackageServer\ShareName\Drivers\NIC` on a network, a network administrator copies the driver package to the shared directory and concatenates the shared directory path of the **DevicePath** value entry in the registry of the network computers, as follows:

```
%SystemRoot%\inf;...;\DriverPackageServer\ShareName\Drivers\NIC
```

## NOTE

Specifying network share in **DevicePath** in a machine with point and print client connection can result in excessive network share access and printing delays. This is because each time printerdata is changed in the server, the client will iterate through **DevicePath** directories checking for availability of newer print drivers.

# Preinstalling Driver Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

To preinstall 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 that is created in step (1).  
The driver package includes the driver or drivers, the INF file, the catalog file, and other installation files.
3. Call [SetupCopyOEMInf](#) specifying the INF file in the directory that was 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 %SystemRoot%\Inf directory on the target system and directs Windows 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 that it recognizes a device that is 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](#).)

# Hardware-First Installation

12/5/2018 • 2 minutes to read • [Edit Online](#)

A hardware-first installation involves the installation of device drivers that is triggered by plugging in a new hardware device to a system. In this case, your [driver package](#) for the device has not been preinstalled in the system.

If the user plugs in new hardware before inserting your distribution medium into a drive, the way that Windows installs a driver depends on the type of drivers that are available for the device. The different installation types are described in the following sections:

- [Installing an In-box Driver or a Preinstalled Driver](#)

# Installing an Inbox Driver or a Preinstalled Driver

11/2/2020 • 2 minutes to read • [Edit Online](#)

If there is an inbox driver or a [preinstalled driver](#) that matches a device, Windows automatically installs the driver that has the best match to the device. Windows will not prompt the user to insert a distribution medium. In addition, Windows does not attempt to access Windows Update to determine whether there is a Windows Update driver that is a better match than the drivers that are already installed on the computer.

For more information about how driver packages are located during a [hardware-first installation](#), see [Where Windows Searches for Drivers](#).

For information about how Windows selects the driver that has the best match to a device, see [How Windows Selects Drivers](#).

# Software-First Installation

11/2/2020 • 2 minutes to read • [Edit Online](#)

A software-first installation involves the staging and preinstallation of your [driver package](#) on the system before the hardware device is plugged in. After the device is plugged in, the driver from the driver package is installed.

If the user inserts your distribution medium before plugging in the device, an AutoRun-enabled installation application can:

- [Check for in-progress installations](#), and stop executing if other installation activities are in progress.
- [Determine whether a device is plugged in](#).
- [Preinstall driver packages](#)
- Use Microsoft Installer to [install device-specific applications](#).
- If the device is "hot-pluggable," tell the user to plug it in.

If the bus does not provide hot-plug notification, initiate reenumeration 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.

# Checking for In-Progress Installations

11/2/2020 • 2 minutes to read • [Edit Online](#)

Your *device installation application* should determine whether other installation activities are in progress before performing its installations. To make this determination, the device installation 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 example, the Found New Hardware Wizard might be active), the device installation application should exit.

To make your *device installation application* compatible with platforms that do not support **CMP\_WaitNoPendingInstallEvents**, the application should include the following code:

```
BOOL
IsDeviceInstallInProgress (VOID)
{
    HMODULE hModule;
    CMP_WAITNOOPENINGINSTALLEVENTS_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_WAITNOOPENINGINSTALLEVENTS_PROC)GetProcAddress(hModule,
                                                          "CMP_WaitNoPendingInstallEvents");
    if(!pCMP_WaitNoPendingInstallEvents)
    {
        // We're running on a release of the operating system that doesn't supply this function.
        // Trust the operating system to suppress AutoRun when appropriate.
        return FALSE;
    }
    return (pCMP_WaitNoPendingInstallEvents(0) == WAIT_TIMEOUT);
}

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 does not support **CMP\_WaitNoPendingInstallEvents**, the platform does not start AutoRun if installation activities are in progress.

For a sample usage of this code, see the toaster installation package under the *src\general\toaster* subdirectory of the Windows Driver Kit (WDK).

# Determining Whether a Device Is Plugged In

11/2/2020 • 3 minutes to read • [Edit Online](#)

Be aware 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 independent hardware vendors (IHVs) typically provide one distribution disk, and a disk can only have one AutoRun-invoked application, your AutoRun-invoked device installation application must determine whether your device is plugged in.

To determine whether a device is plugged in, the application can call the [UpdateDriverForPlugAndPlayDevices](#) function, passing the hardware ID of the device. The device is plugged in if one of the following is true:

- The function returns **TRUE**. (This also installs the driver for the device.)
- The function returns **FALSE** and the Win32 [GetLastError](#) function returns **ERROR\_NO\_MORE\_ITEMS**. (No installation occurs.)

The device is not plugged in if the function returns **FALSE** and [GetLastError](#) returns **NO\_SUCH\_DEVINST**. (No installation occurs.)

## Reinstalling an Unplugged Device

When a device that formerly was attached is now unplugged, the device's *devnode* remains in the system, although it is both inactive and hidden. Before you can reinstall such a device, you must first find this "phantom" devnode, and mark it as needing reinstallation. Then, when the device is plugged back in, Plug and Play will reenumerate the device, find the new driver for it, and install the driver for the device.

### To reinstall an unplugged device:

1. Call the [SetupCopyOEMInf](#) function.

The [SetupCopyOEMInf](#) function ensures that the correct INF file is present in the `%SystemRoot%\inf` directory.

2. Find the unplugged devices.

Call the [SetupDiGetClassDevs](#) function. In the call to this function, clear the **DIGCF\_PRESENT** flag in the *Flags* parameter. You have to find *all* devices, not just those that are present. You can narrow the results of your search by specifying the particular device class in the *ClassGuid* parameter.

3. Find the hardware IDs and compatible IDs of unplugged devices.

[SetupDiGetClassDevs](#) returns a handle to the *device information set* that contains all installed devices, whether plugged in or not, in the device class (assuming that you specified a device class in the first step). By making successive calls to the [SetupDiEnumDeviceInfo](#) function, you can use this handle to enumerate all the devices in the device information set. Each call gives you an [SP\\_DEVINFO\\_DATA](#) structure for the device. To obtain the list of hardware IDs, call the [SetupDiGetDeviceRegistryProperty](#) function with the *Property* parameter set to **SPDRP\_HARDWAREID**. To obtain the list of the compatible IDs, call the same function, but with the *Property* parameter set to **SPDRP\_COMPATIBLEIDS**. Both lists are MULTI-SZ strings.

4. Look for a match between the ID of your device and the hardware IDs (or compatible IDs) of the previous step.

Make sure that you perform full string comparisons between the hardware ID/compatible ID and the ID for your device. A partial comparison could lead to incorrect matches.

When you find a match, call the [CM\\_Get\\_DevNode\\_Status](#) function, passing **SP\_DRVINFO\_DATA.DevInst**

in the *dnDevInst* parameter. If this function returns CR\_NO SUCH\_DEVINST, that confirms that the device is unattached (that is, has a phantom devnode).

5. Mark the device.

Call the [SetupDiGetDeviceRegistryProperty](#) function with the *Property* parameter set to SPDRP\_CONFIGFLAGS. When this function returns, the *PropertyBuffer* parameter points to the device's **ConfigFlags** value from the registry. Perform a bitwise OR of this value with CONFIGFLAG\_REINSTALL (defined in *Regstr.h*). After doing this, call the [SetupDiSetDeviceRegistryProperty](#) function, with the *Property* parameter set to SPDRP\_CONFIGFLAGS, and the *PropertyBuffer* parameter set to the address of the device's modified **ConfigFlags** value. This action modifies the registry's **ConfigFlags** value to incorporate the CONFIGFLAG\_REINSTALL flag. This causes the device to be reinstalled the next time that the device is reenumerated.

6. Plug in the device.

Plug and Play will reenumerate the device, find the new driver for it, and install that driver.

# Uninstalling Devices and Driver Packages

1/11/2019 • 2 minutes to read • [Edit Online](#)

After a device is installed, it might be necessary to uninstall a device or a [driver package](#). For example, an end-user might decide to replace the associated device, or the driver package might have to be uninstalled when a driver is updated.

When you uninstall a device, you must remove the device node (*devnode*) that represents the physical instance of the device in the system.

When you uninstall a [driver package](#), you must complete the following actions:

- Remove the files that are associated with the [driver package](#) from the [driver store](#).
- Delete the binary files of the driver package.

This section describes how to uninstall devices and driver packages. It is intended for driver developers who want to provide instructions or tools to their customers.

This section includes the following topics:

[How Devices and Driver Packages are Uninstalled](#)

[Using Device Manager to Uninstall Devices and Driver Packages](#)

[Using SetupAPI to Uninstall Devices and Driver Packages](#)

# How Devices and Driver Packages are Uninstalled

11/2/2020 • 2 minutes to read • [Edit Online](#)

This page describes how software uninstalls a device and removes a driver package from the [driver store](#).

## Uninstalling the Device

To remove the device node (*devnode*) that represents a physical device, use one of the following:

- To uninstall only the specified device, use a device installation application that calls the [SetupAPI](#) function [SetupDiCallClassInstaller](#) with a request of [DIF\\_REMOVE](#).
- To uninstall the specified device and any devices below it in the device tree, use a device installation application that calls the [DiUninstallDevice](#) function.

When a device is uninstalled using one of these methods, the Plug and Play (PnP) manager removes the association between the driver binary files and the device.

The device remains in the kernel PnP tree and the [driver package](#) remains in the [driver store](#). If the PnP manager re-enumerates the device (for example if the device is unplugged and then plugged in again), the PnP manager treats it as a new device instance and installs the driver package from the driver store.

For info on how an end user can uninstall a device, see [Using Device Manager to Uninstall Devices and Driver Packages](#).

## Deleting a Driver Package from the Driver Store

To delete a [driver package](#) from the [driver store](#), do one of the following:

- From the command prompt, use `pnputil /delete-driver <example.inf> /uninstall`. For info on PnPUtil commands, see [PnPUtil Command Syntax](#).
- On Windows 10, version 1703 or later, a device installation application can call [DiUninstallDriverW](#).
- On earlier versions of Windows, a device installation application should first issue a [DIF\\_REMOVE](#) request or call the [DiUninstallDevice](#) function to uninstall all devices and then call [SetupUninstallOEMInf](#) to remove the driver.

Deleting a driver package from the driver store removes associated metadata from the PnP manager's internal database and deletes related INF files from the system INF directory.

After the driver package has been removed, it is no longer available to be installed on a device. To reinstall, download the driver again from the original source, such as Windows Update.

Manually deleting the [driver package](#) from the [driver store](#) may result in unpredictable behavior.

# Using Device Manager to Uninstall Devices and Driver Packages

11/2/2020 • 2 minutes to read • [Edit Online](#)

This page describes how to uninstall a device or driver on Windows 10.

First, open Settings (you can do this using the `Windows+I` keyboard shortcut) and type Remove. Select **Add or remove programs**. If your device appears in the list of programs, select uninstall.

If your device does not appear in the list, click the Start button, type Device Manager, and press Enter.

Then follow these steps:

1. Expand the node that represents the type of device that you want to uninstall, right-click the device entry, and select **Uninstall**.
2. On the **Confirm Device Removal** dialog box, select the **Delete the driver software for this device** option, and select **OK**.
3. When the uninstall process is complete, remove the physical device.

With some devices, the device might continue to function until the system has been restarted.

For more information about uninstalling driver and driver packages, see [How Devices and Driver Packages are Uninstalled](#).

# Using SetupAPI to Uninstall Devices and Driver Packages

12/1/2020 • 2 minutes to read • [Edit Online](#)

SetupAPI is a system component that provides two sets of functions:

- [General Setup functions](#)
- [Device installation functions](#)

*Device installation applications, co-installers, and class installers* can use these functions to perform custom operations for device installation. SetupAPI also supports uninstalling the devices and [driver packages](#) that it installs.

This topic describes the procedures that you can follow to uninstall devices and driver packages by using the SetupAPI functions.

For more information about uninstalling driver and driver packages, see [How Devices and Driver Packages are Uninstalled](#).

## Uninstalling the Device

SetupAPI allows you to uninstall a device and remove the device node (*devnode*) from the system by using the following methods:

- A device installation application can request that a device be uninstalled by calling the [SetupDiCallClassInstaller](#) function. When the application calls this function to uninstall a device, it must set the *InstallFunction* parameter to the [DIF\\_REMOVE](#) code. For a list of all DIF codes, see [Device Installation Functions](#).

If [SetupDiRemoveDevice](#) is called during the processing of the DIF\_REMOVE request, the function removes the device's devnode from the system. It also deletes the device's hardware and software registry keys, together with any hardware-profile-specific registry keys (configuration-specific registry keys).

**Note** [SetupDiRemoveDevice](#) must only be called by a class installer and not by a device installation application.

For more information about DIF codes, see [Handling DIF Codes](#).

- Starting with Windows 7, a device installation application can uninstall a device by calling the [DiUninstallDevice](#) function. This function is similar to calling [SetupDiCallClassInstaller](#) with the *InstallFunction* parameter set to [DIF\\_REMOVE](#). However, in addition to removing the devnode of the specified device, this function attempts to remove all child devnodes of the device that are present on the system at the time of the call.

## Deleting a Driver Package from the Driver Store

Starting with Windows XP, a device installation application can call the [SetupUninstallOEMInf](#) function to remove a specified [INF file](#) from the system INF file directory.

Starting with Windows Vista, this function also removes the [driver package](#), which contains the specified INF file, from the [driver store](#).

## Deleting the Binary Files of the Installed Driver

SetupAPI cannot be used to perform this action.



# Updating Driver Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

Drivers are updated whenever one of the following occurs:

- The **Hardware Update Wizard** is run from Device Manager.

**Note** Starting with Windows Vista, this wizard is now named the **Update Driver Software Wizard**.

- Windows Update is run.
- Installation software for a device is run.
- Starting with Windows Vista, you can run the [PnPUtility](#) tool from an elevated command prompt to install or update the [driver package](#) for the device.

Use the following guidelines when you write 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.

Starting with Windows Vista, installation software can also call one of the following to update drivers:

- [DiInstallDriver](#), which pre-installs a driver and then installs the driver on devices present in the system that the driver supports.
- [DiInstallDevice](#), which installs a specified driver from the driver store on a specified device that is present in the system.

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\\_NEWDVICEWIZARD\\_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.
- Starting with Windows XP, the registry values [CoInstallers32](#) and [EnumPropPages32](#) are deleted before the delivery of [DIF\\_REGISTER\\_COINSTALLERS](#). INF files for earlier operating system versions must explicitly either delete these values or perform a nonappending modify operation on them.
- Starting with Windows XP, the registry values [UpperFilters](#) and [LowerFilters](#) are deleted before the delivery of [DIF\\_INSTALLDEVICE](#). INF files for earlier operating system versions must explicitly either delete these values or perform a nonappending modify operation on them.
- Do *not* use [INF DelFiles directives](#) or [INF RenFiles directives](#) when updating drivers. Windows 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 computer. Windows 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.

For more information about INF files, see [Creating an INF File](#) and [INF File Sections and Directives](#).

# Creating Secure Device Installations

12/1/2020 • 5 minutes to read • [Edit Online](#)

When you create a [driver package](#), you must make sure that the installation of your device will always be performed in a secure manner. A secure device installation is one that does the following:

- limits access to the device and its device interface classes
- limits access to the driver services that were created for the device
- protects driver files from modification or deletion
- limits access to the device's registry entries
- limits access to the device's WMI classes
- uses SetupAPI functions correctly

Device installation security is controlled by *security descriptors*. The primary medium for specifying security descriptors is the INF file. The system provides default security descriptors, and under most circumstances you do not have to override these descriptors.

## Security Settings for Devices and Interfaces

The system supplies default security descriptors for all [system-supplied device setup classes](#). Generally, these descriptors allow full access for system administrators and read/write/execute access for users. (The security descriptors that control access to a device also control access to the device's [device interface classes](#), if any.)

INF files for WDM drivers can specify security settings, either per class or per device, that override the system's default settings. Vendors who create a new device setup class should specify a security descriptor for the class. Generally, specifying a device-specific security descriptor is not necessary. It might be useful to supply a device-specific security descriptor if different types of devices that belong to the same class have significantly different types of users.

To specify a security descriptor for all devices that belong to a WDM device setup class, use an [INF AddReg directive](#) within an [INF ClassInstall32 section](#) of the class installer's INF file. The **AddReg** directive must point to an *add-registry-section* that sets values for **DeviceType** and **Security** registry entries. These registry values specify a security descriptor for all devices of the specified device type.

To specify a security descriptor for a single device that belongs to a WDM device setup class, use an [INF AddReg directive](#) within an [INF DDInstall.HW section](#) of the device's INF file. The **AddReg** directive must point to an *add-registry-section* that sets values for **DeviceType** and **Security** registry entries. These registry values specify a security descriptor for all devices that match the [hardware ID](#) or [compatible IDs](#) specified by an associated [INF Models section](#).

By default, the system applies the security descriptor set for a device to a request to open the device object that represents the device (for example, a request to open the device whose NT device name is `|Device|DeviceName`).

However, the system does not by default apply the security descriptor set for a device to a request to open an object in the namespace of the device, where the device namespace includes all objects whose names have the form `|Device|DeviceName|ObjectName`. To ensure that the same security settings are applied to open requests for objects in the namespace of a device, set the `FILE_DEVICE_SECURE_OPEN` device characteristics flag for a device. For more information about secure device access, see [Controlling Device Namespace Access \(Windows Drivers\)](#). For information about how to set the `FILE_DEVICE_SECURE_OPEN` device characteristics flag, See [Specifying Device Characteristics \(Windows Drivers\)](#).

The PnP manager sets security values on device objects after it calls a driver's **AddDevice** routine. Some WDM drivers can specify a device-specific security descriptor when creating a physical device object (PDO) by calling **IoCreateDeviceSecure**. For more information, see [Securing Device Objects](#).

## Security Settings for Driver Files

When copying files by using the **INF CopyFiles directive**, it is possible to specify a *file-list-section*.**Security** section. This section specifies a security descriptor for all files that are copied by the **CopyFiles** directive. However, vendors never have to specify a security descriptor for driver files, if the installation destination is one of the system subdirectories of %SystemRoot%. (For more information about these subdirectories, see [Using Dirids](#).) The system provides default security descriptors for these subdirectories, and the default descriptors should not be overridden.

## Security Settings for Driver Services

Within a driver INF file's *service-install-section* (see [INF AddService Directive](#)), you can include a **Security** entry. This entry specifies the permissions that are required to perform such operations as starting, stopping, and configuring the driver services that are associated with your device. However, the system provides a default security descriptor for driver services, and this default descriptor generally does not have to be overridden.

## Security Settings for Device and Driver Registry Entries

When specifying registry entries in INF files by using **INF AddReg directives**, you can include an *add-registry-section*.**Security** section for each *add-registry-section*. The *add-registry-section*.**Security** section specifies access permissions to the created registry entries that are created by the associated *add-registry-section* section. The system provides a default security descriptor for all registry entries created under the HKR relative root. Therefore, you do not have to specify a security descriptor when creating registry entries under the relative root.

## Security Settings for WMI Classes

The system assigns default security descriptors to the GUIDs that identify WMI classes. For Windows XP and earlier operating system versions, the default security descriptor for WMI GUIDs allows full access to all users. Starting with Windows Server 2003, the default security descriptor allows access only to administrators.

If your driver defines WMI classes and you do not want to use the system's default security descriptors for these classes, you can supply security descriptors by using an **INF DDInstall.WMI section** within the device's INF file.

## Using SetupAPI Functions Correctly

If your [driver package](#) includes installers, co-installers, or other installation applications that call SetupAPI functions, you must follow the [guidelines for using SetupAPI](#).

## Testing Installation Security Settings

Use [SetupAPI logging](#) to verify that security settings that are associated with installing your device have been specified correctly. Set the logging level to verbose (0x0000FFFF), then attempt various installation scenarios.

Such scenarios should include both initial installations and reinstallations, from both user accounts and system administrator accounts. Try plugging in your device before you install software, and vice versa.

If an installation succeeds, view the log to confirm that no errors occurred. If an installation fails, view the log to determine the cause of the failure.

Additionally, after an installation completes you can do the following:

- Use Registry Editor to view the security settings that are assigned to a registry entry.
- Use **My Computer** to view the security settings that are assigned to a file.

# Using a Component INF File

11/2/2020 • 4 minutes to read • [Edit Online](#)

If you want to include user-mode software for use with a device on Windows 10, you have the following options to create a [DCH-compliant driver](#):

METHOD	SCENARIO
<a href="#">Hardware support apps (HSA)</a>	Device add-on software packaged as a UWP app that is delivered and serviced from the Microsoft Store. Recommended approach.
Software components	Device add-on software is an MSI or EXE binary, a Win32 service, or software installed using AddReg and CopyFiles. Referenced binary only runs on desktop editions (Home, Pro, and Enterprise). The referenced binary will not run on Windows 10S.

A software component is a separate, standalone driver package that can install one or more software modules. The installed software enhances the value of the device, but is not necessary for basic device functionality and does not require an associated function driver service.

This page provides guidelines for the use of software components.

## Getting started

To create components, an [extension INF file](#) specifies the [INF AddComponent directive](#) one or more times in the [INF DDInstall.Components](#) section. For each software component referenced in an extension INF file, the system creates a virtual software-enumerated child device. More than one driver package can reference the same software component.

Virtual device children can be updated independently just like any other device, as long as the parent device is started. We recommend separating functionality into as many different groupings as makes sense from a servicing perspective, and then creating one software component for each grouping.

You'll provide an INF file for each software component.

If your software component INF specifies the [AddSoftware directive](#), the component INF:

- Must be a [universal INF file](#).
- Must specify the [SoftwareComponent](#) setup class.

You can specify the [AddSoftware directive](#) one or more times.

### NOTE

When using Type 2 of the AddSoftware directive, it is not required to utilize a Component INF. The directive can be used in any INF successfully. An AddSoftware directive of Type 1, however, must be used from a Component INF.

Additionally, any INF (component or not) matching on a software component device:

- Can specify Win32 user services using the [AddService directive](#).
- Can install software using the [INF AddReg directive](#) and the [INF CopyFiles directive](#).

- Does not require a function driver service.
- Can be uninstalled by the user independently from the parent device.

You can find an example of a component INF in the [Driver package installation toolkit for universal drivers](#).

**Note:** In order for a software-enumerated component device to function, its parent must be started. If there is no driver available for the parent device, driver developers can create their own and optionally leverage the pass-through driver "umpass.sys". This driver is included in Windows and, effectively, does nothing other than start the device. In order to use umpass.sys, developers should use the Include/Needs INF directives in the [DDInstall section](#) for each possible [DDInstall.\*] section to the corresponding [UmPass.\*] sections as shown below, regardless of whether the INF specifies any directives for that section or not:

```
[DDInstall]
Include=umpass.inf
Needs=UmPass
; also include any existing DDInstall directives

[DDInstall.HW]
Include=umpass.inf
Needs=UmPass.HW
; also include any existing DDInstall.HW directives

[DDInstall.Interfaces]
Include=umpass.inf
Needs=UmPass.Interfaces
; also include any existing DDInstall.Interfaces directives

[DDInstall.Services]
Include=umpass.inf
Needs=UmPass.Services
; also include any existing DDInstall.Services directives
```

## Accessing a device from a software component

To retrieve the device instance ID of a device that is associated with a software component, use the **SoftwareArguments** value in the [INF AddSoftware directive](#) section with the `<>DeviceInstanceID>>` runtime context variable.

The executable can then retrieve the device instance ID of the software component from its incoming argument list.

Next, if the software component is targeting the Universal [target platform](#), use the following procedure:

1. Call [CM\\_Locate\\_DevNode](#) with the device instance ID of the software component to retrieve a device handle.
2. Call [CM\\_Get\\_Parent](#) to retrieve a handle to that device's parent. This parent is the device that added the software component using the [INF AddComponent Directive](#).
3. Then, to retrieve the device instance ID of the parent, call [CM\\_Get\\_Device\\_ID](#) on the handle from [CM\\_Get\\_Parent](#).

If the software component is targeting the Desktop [target platform](#) only, use the following procedure:

1. Call [SetupDiCreateDeviceInfoList](#) to create an empty device information set.
2. Call [SetupDiOpenDeviceInfo](#) with the software component device's device instance ID.
3. Call [SetupDiGetDeviceProperty](#) with `DEVPKEY_Device_Parent` to retrieve the device instance ID of the parent.

## Example

The following example shows how you might use a software component to install a control panel using an

executable for a graphics card.

### Driver package INF file

```
[Version]
Signature      = "$WINDOWS NT$"
Class          = Extension
ClassGuid      = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}
ExtensionId   = {zzzzzzz-zzzz-zzzz-zzzz-zzzzzzzzzz} ; replace with your own GUID
Provider       = %CONTOSO%
DriverVer     = 06/21/2006,1.0.0.0
CatalogFile   = ContosoGrfx.cat

[Manufacturer]
%CONTOSO%=Contoso,NTx86

[Contoso.NTx86]
%ContosoGrfx.DeviceDesc%=ContosoGrfx, PCI\VEN0001&DEV0001

[ContosoGrfx.NT]
;empty

[ContosoGrfx.NT.Components]
AddComponent = ContosoControlPanel,, Component_Inst

[Component_Inst]
ComponentIDs = VID0001&PID0001&SID0001

[Strings]
CONTOSO = "Contoso Inc."
ContosoGrfx.DeviceDesc = "Contoso Graphics Card Extension"
```

### Software component INF file

```

[Version]
Signature = "$WINDOWS NT$"
Class = SoftwareComponent
ClassGuid = {5c4c3332-344d-483c-8739-259e934c9cc8}
Provider = %CONTOSO%
DriverVer = 06/21/2006,1.0.0.0
CatalogFile = ContosoCtrlPnl.cat

[SourceDisksNames]
1 = %Disk%,,""

[SourceDisksFiles]
ContosoCtrlPnl.exe = 1

[DestinationDirs]
DefaultDestDir = 13

[Manufacturer]
%CONTOSO%=Contoso,NTx86

[Contoso.NTx86]
%ContosoCtrlPnl.DeviceDesc%=ContosoCtrlPnl, SWC\VID0001&PID0001&SID0001

[ContosoCtrlPnl.NT]
CopyFiles=ContosoCtrlPnl.NT.Copy

[ContosoCtrlPnl.NT.Copy]
ContosoCtrlPnl.exe

[ContosoCtrlPnl.NT.Services]
AddService = , %SPSVCINST_ASSOCSERVICE%

[ContosoCtrlPnl.NT.Software]
AddSoftware = ContosoGrfx1CtrlPnl,, Software_Inst

[Software_Inst]
SoftwareType = 1
SoftwareBinary = %13%\ContosoCtrlPnl.exe
SoftwareArguments = <<DeviceInstanceId>>
SoftwareVersion = 1.0.0.0

[Strings]
SPSVCINST_ASSOCSERVICE = 0x00000002
CONTOSO = "Contoso"
ContosoCtrlPnl.DeviceDesc = "Contoso Control Panel"

```

The driver validation and submission process is the same for component INFs as for regular INFs. For more info, see [Windows HLK Getting Started](#).

For more info on setup classes, see [System-Defined Device Setup Classes Available to Vendors](#).

## See Also

[INF AddComponent Directive](#)

[INF AddSoftware directive](#)

[INF DDInstall.Components Section](#)

[INF DDInstall.Software Section](#)

# Pairing a driver with a Universal Windows Platform (UWP) app

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting in Windows 10 version 1709, you can specify that a Universal Windows Platform (UWP) app should only load if a specific driver is present. When you use this option, the Microsoft Store offers each user the most recent version of the app that works with the installed version of the driver on that user's computer.

The app can further constrain loading to a particular driver version or date. This topic describes the steps required in **both the app and driver** to create such a requirement.

## NOTE

Both the application *and* the driver *must* declare the dependency on the application (HSA).

## Steps in the app

To cause a UWP app to load only when a specific driver is present, add two XML elements to the manifest XML (.appx) file for the app:

- [uap5:DriverDependency](#)
- [uap5:DriverConstraint](#)

In particular, use these elements to specify at least one driver dependency containing at least one driver constraint. See further details on use of these elements on the reference pages linked to above. The latter page contains an example.

## Steps in the driver

Next, do the following in the driver's INF file:

1. Specify the [INF AddSoftware Directive](#).
2. Set the **SoftwareType** entry to 2.
3. Provide a Package Family Name (PFN) in the **SoftwareID** entry.

In addition to matching the most recent app and driver versions, the system also tries to match previous app and driver versions. For example, if app version 2 specifies minimum driver version 2, and app version 1 specifies minimum driver version 1, a system that has driver version 1 will successfully load app version 1.

## See Also

- [uap5:DriverDependency](#)
- [uap5:DriverConstraint](#)
- [INF AddSoftware Directive](#)
- [Hardware Support App \(HSA\): Steps for Driver Developers](#)
- [Hardware Support App \(HSA\): Steps for App Developers](#)

# Device Identification Strings

11/2/2020 • 2 minutes to read • [Edit Online](#)

The Plug and Play (PnP) manager and other [device installation components](#) use device identification strings to identify devices that are installed in a computer.

Windows uses the following device identification strings to locate the information (INF) file that best matches the device. These strings are reported by a device's enumerator, a system component that discovers PnP devices based on a PnP hardware standard. These tasks are carried out by PnP Bus Drivers in partnership with the PnP manager. A device is typically enumerated by its parent bus driver, such as the PCI or PCMCIA bus driver. Some devices are enumerated by a bus filter driver, such as the ACPI Driver.

- [Hardware IDs](#)
- [Compatible IDs](#)

Windows tries to find a match for one of the hardware IDs or compatible IDs. For more information about how Windows uses these IDs to match a device to an INF file, and how to specify IDs in an INF file, see [How Windows Selects Drivers](#).

In addition to using the preceding IDs to identify devices, the PnP manager uses the following IDs to uniquely identify instances of each device that are installed in a computer:

- [Instance IDs](#)
- [Device instance IDs](#)

Starting with Windows 7, the PnP manager uses the [Container ID](#) device identification string to group one or more device nodes (devnodes) that were enumerated from each instance of a physical device installed in a computer.

Each enumerator customizes its device IDs, hardware IDs, and compatible IDs to uniquely identify the device that it enumerates. In addition, each enumerator has its own policy to identify hardware IDs and compatible IDs. For more information about hardware ID and compatible ID formats for most of the system buses, see [Device Identifier Formats](#).

## NOTE

Device identification strings should not be parsed. They are meant only for string comparisons and should be treated as opaque strings.

# Plug and Play ID - PNPID Request

6/25/2019 • 2 minutes to read • [Edit Online](#)

Plug and Play ID are now requested through the UEFI.org website. To request an ID, see [PNP ID and ACPI ID Registry](#).

# Device ID

11/2/2020 • 2 minutes to read • [Edit Online](#)

A device ID is a string reported by a device's *enumerator*. A device has only one device ID. A device ID has the same format as a [hardware ID](#).

The Plug and Play (PnP) manager uses the device ID to create a subkey for a device under the registry key for the device's enumerator.

To obtain a device ID, use an [IRP\\_MN\\_QUERY\\_ID](#) request and set the **Parameters.QueryId.IdType** field to **BusQueryDeviceID**.

# Hardware ID

12/1/2020 • 2 minutes to read • [Edit Online](#)

A hardware ID is a vendor-defined identification string that Windows uses to match a device to an INF file. In most cases, a device has associated with it a list of hardware IDs. (However, there are exceptions – see Identifiers for 1394 Devices). When an *enumerator* reports a list of hardware IDs for a device, the hardware IDs should be listed in order of decreasing suitability.

A hardware ID has one of the following generic formats:

```
<enumerator>\<enumerator-specific-device-ID>
```

This is the most common format for individual PnP devices reported to the Plug and Play (PnP) manager by a single enumerator. New enumerators should use this format or the following format. For more information about enumerator-specific device IDs, see [Device Identifier Formats](#).

```
\*<generic-device-ID>
```

The asterisk indicates that the device is supported by more than one enumerator, such as ISAPNP and the BIOS. For more information about this type of ID, see [Generic Identifiers](#).

```
<device-class-specific-ID>
```

## Selecting a hardware ID

Root enumerated devices sharing generic namespace such as `ROOT\SYSTEM` may cause conflicts and yellow-bang in device manager on OS upgrade.

To prevent this, use a unique namespace for each driver that includes a root enumerated device. For a USB or system device, instead of using `ROOT\USB` or `ROOT\SYSTEM`, use `ROOT\[COMPANYNAME]\[DEVICENAME]`. In addition, the driver installer code should check to see if the devnode is already present and take any necessary corrective action before installing.

An existing device class that has established its own naming convention might use a custom format. For information about their hardware ID formats, see the hardware specification for such buses. New enumerators should not use this format.

The number of characters of a hardware ID, excluding a NULL terminator, must be less than `MAX_DEVICE_ID_LEN`. This constraint applies to the sum of the lengths of all the fields and any "\" field separators in a hardware ID. For more information about constraints on device IDs, see the Operations section of [IRP\\_MN\\_QUERY\\_ID](#).

## Obtaining the list of hardware IDs for a device

To obtain the list of hardware IDs for a device, call [IoGetDeviceProperty](#) with the *DeviceProperty* parameter set to `DevicePropertyHardwareID`. The list of hardware IDs that this routine retrieves is a `REG_MULTI_SZ` value. The maximum number of characters in a hardware list, including a NULL terminator after each hardware ID and a final NULL terminator, is `REGSTR_VAL_MAX_HCID_LEN`. The maximum possible number of IDs in a list of hardware IDs is 64.

## Examples of Hardware IDs

In the following, the first example is a [generic identifier](#) for a PnP device, and the second example is an

identifier for a PCI device:

```
root\*PNP0F08
```

```
PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02
```

## See Also

[INF HardwareId Directive](#)

# Compatible ID

11/2/2020 • 2 minutes to read • [Edit Online](#)

A compatible ID is a vendor-defined identification string that Windows uses to match a device to an INF file. A device can have associated with it a list of compatible IDs. The compatible IDs should be listed in order of decreasing suitability. If Windows cannot locate an INF file that matches one of a device's hardware IDs, it uses compatible IDs to locate an INF file. Compatible IDs have the same format as [hardware IDs](#). However, compatible IDs are typically more generic than hardware IDs.

If a vendor ships an INF file that specifies a compatible ID for a driver node, the vendor should make sure that their INF file can support all the hardware that matches the compatible ID.

To obtain a list of compatible IDs for a device, call [IoGetDeviceProperty](#) with the *DeviceProperty* parameter set to **DevicePropertyCompatibleID**. The list of compatible IDs that this routine retrieves is a [REG\\_MULTI\\_SZ](#) value. The maximum number of characters in a compatible ID list, including a NULL terminator after each compatible ID and a final NULL terminator, is [REGSTR\\_VAL\\_MAX\\_HCID\\_LEN](#). The maximum possible number of IDs in a list of compatible IDs is 64.

# Instance ID

11/2/2020 • 2 minutes to read • [Edit Online](#)

An instance ID is a device identification string that distinguishes a device from other devices of the same type on a computer. An instance ID contains serial number information, if supported by the underlying bus, or some kind of location information. The string cannot contain any "\" characters; otherwise, the generic format of the string is bus-specific.

The number of characters of an instance ID, excluding a NULL-terminator, must be less than MAX\_DEVICE\_ID\_LEN. In addition, when an instance ID is concatenated to a [device ID](#) to create a device instance ID, the lengths of the device ID and the instance ID are further constrained by the maximum possible length of a device instance ID.

The **UniqueId** member of the [DEVICE\\_CAPABILITIES](#) structure for a device indicates if a bus-supplied instance ID is unique across the system, as follows:

- If **UniqueId** is **FALSE**, the bus-supplied instance ID for a device is unique only to the device's bus. The Plug and Play (PnP) manager modifies the bus-supplied instance ID, and combines it with the corresponding device ID, to create a device instance ID that is unique in the system.
- If **UniqueId** is **TRUE**, the device instance ID, formed from the bus-supplied device ID and instance ID, uniquely identifies a device in the system.

An instance ID is persistent across system restarts.

To obtain the bus-supplied instance ID for a device, use an [IRP\\_MN\\_QUERY\\_ID](#) request and set the **Parameters.QueryId.IdType** member to **BusQueryInstanceId**.

# Device Instance ID

11/2/2020 • 2 minutes to read • [Edit Online](#)

A device instance ID is a system-supplied device identification string that uniquely identifies a device in the system. The Plug and Play (PnP) manager assigns a device instance ID to each device node (*devnode*) in a system's [device tree](#).

The format of this string consists of an [instance ID](#) concatenated to a [device ID](#), as follows:

```
<device-ID>\<instance-specific-ID>
```

The number of characters of a device instance ID, excluding a NULL-terminator, must be less than MAX\_DEVICE\_ID\_LEN. This constraint applies to the sum of the lengths of all the fields and "\" field separator between the *device ID* and *instance-specific-ID* fields.

A device instance ID is persistent across system restarts.

The following is an example of an instance ID ("1&08") concatenated to a device ID for a PCI device:

```
PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02\1&08
```

# Container ID

1/11/2019 • 2 minutes to read • [Edit Online](#)

A container ID is a system-supplied device identification string that uniquely groups the functional devices associated with a single-function or multifunction device installed in the computer.

Starting with Windows 7, the Plug and Play (PnP) manager uses the container ID to group one or more device nodes (*devnodes*) that originated from and belong to each instance of a particular physical device. This instance is referred to as the *device container*.

Grouping all the devnodes that originated from an instance of a single device achieves the following:

- The operating system can determine how functionality is related among child devnodes and their container devnode.
- The user or applications are presented with a device-centric view of devices instead of the traditional function-centric view.

This section contains topics that discuss the container ID in more detail:

[Overview of Container IDs](#)

[How Container IDs are Generated](#)

[Verifying the Implementation of Container IDs](#)

[Troubleshooting the Implementation of Container IDs](#)

# Overview of Container IDs

11/2/2020 • 2 minutes to read • [Edit Online](#)

In the Windows family of operating systems, devices are basically a collection of functional device instances, each of which represents a functional endpoint that enables some form of communication to the device.

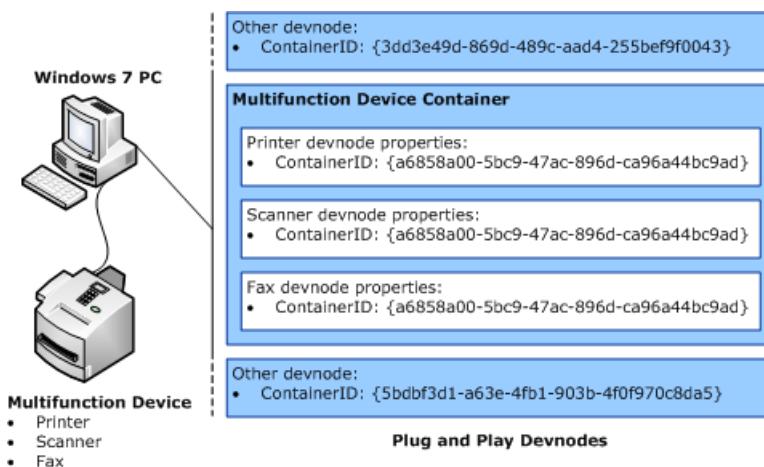
The term *device node (devnode)* refers to the *driver stack* for such a functional endpoint, in addition to the properties that describe the endpoint and its associated state. For example, a multifunction device that supports printer, scanner, and fax functionality can have multiple devnodes, one for each functional endpoint in the device.

Before Windows 7, each functional endpoint had a devnode that is associated with it. Windows platform components and third-party applications can query devnodes for device status and information, and can communicate with device hardware through interfaces that functional endpoints expose.

For a single-function device, a single devnode contains all the information that relates to device's functional endpoint. Similarly, a multifunction device has multiple devnodes that are associated with each of the device's functional endpoints. However, Windows cannot recognize that a group of devnodes originated from the same physical device. Each devnode that belongs to the same multifunction device does not include any identification information that enables the Plug and Play (PnP) manager to group several devnodes as a single device. Therefore, it is not possible to have a holistic view of the device and the functions that a single physical device provides.

Starting with Windows 7, the operating system uses a new ID (*container ID*) to group one or more devnodes that originated from and belong to each instance of a particular physical device. The container ID is a property of every devnode, and is specified through a globally unique identifier (*GUID*) value.

Each instance of a physical device that is installed in the computer has a unique container ID. All devnodes that represent a function on that instance of the physical device share the same container ID. The following figure shows an example of that relationship.



There is one container ID with a special meaning for bus drivers: `NUL_GUID` which is defined as: `{00000000-0000-0000-000000000000}`.

In general, do not return `NUL_GUID` as the default case when reporting a container ID. Instead, do not handle `IRP_MN_QUERY_ID` for the `BusQueryContainerIDs` case and let PnP apply its default logic.

When returning `NUL_GUID` as a container ID, a bus driver declares to PnP that the device must not be part of any container, thus returning `NUL_GUID` is appropriate only in very special cases. For example, a *devnode* such as a volume device may span multiple disks in multiple containers but do not belong to any container. Such a device will have a `DEVPKEY_Device_BaseContainerId` equal to `NUL_GUID`, and it will not have a `DEVPKEY_Device_ContainerId` at all.

Aside from very special cases, a bus driver should never return NULL\_GUID when reporting a hardware device and bus drivers should guard against faulty hardware that reports a NULL\_GUID value from their bus. In these cases the bus driver should either treat this as a device error, or treat it as if the device did not report a value.

# How Container IDs are Generated

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, the Plug and Play (PnP) manager generates a container ID for a device node (*devnode*) through one of three mechanisms:

- A bus driver provides a container ID.

When assigning a container ID to a devnode, the PnP manager first checks whether the bus driver of the devnode can provide a container ID. Bus drivers provide a container ID through an [IRP\\_MN\\_QUERY\\_ID](#) request with the **Parameters.QueryId.IdType** field set to **BusQueryContainerID**.

A bus driver can either obtain a genuine container ID that was embedded in the physical device hardware, or use a bus-specific unique ID from the device hardware to generate a container ID. Some examples of bus-specific unique IDs are a device's serial number or a media access control (MAC) address in the device's firmware.

**Note** The independent hardware vendor (IHV) is responsible for the uniqueness of the container ID reported by the bus driver.

For more information, see [Container IDs Generated from a Bus-Specific Unique ID](#).

- The PnP manager generates a container ID through the removable device capability.

If a bus driver cannot provide a container ID for a devnode that it is enumerating, the PnP manager uses the removable device capability to generate a container ID for all devnodes enumerated for the device. The bus driver reports this device capability in response to an [IRP\\_MN\\_QUERY\\_CAPABILITIES](#) request.

For more information, see [Container IDs Generated from the Removable Device Capability](#).

- The PnP manager generates a container ID through an override of the removable device capability.

**Note** In Windows 10, DPWS devices will always generate a container ID for the device using this method.

Although the override mechanism does not change the value of the removable device capability, it forces the PnP manager to use the override setting and not the value of the removable device capability when generating container IDs for devices.

For example, if an override of the removable device capability specifies the device is removable, the PnP manager generates a container ID for all devnodes enumerated for the device. This action is performed regardless of whether the device reported itself as removable or not.

An IHV can populate the registry with keys that override the removable device capability reported by the device. This override mechanism is useful for legacy devices that either do not support the removable device capability or report it incorrectly.

For more information, see [Container IDs Generated from a Removable Device Capability Override](#).

In addition to these methods, the system uses ACPI BIOS object settings to specify device container groupings. For more information, see [Using ACPI for Device Container Grouping](#).

# Container IDs Generated from a Bus-Specific Unique ID

11/2/2020 • 2 minutes to read • [Edit Online](#)

The preferred way to generate a container ID for a device is based on a bus-specific unique ID. This is the most precise and reliable method for generating container IDs.

The Plug and Play (PnP) manager uses this method if the following are true:

- The device contains a bus-specific unique ID.
- The bus driver for the device recognizes this unique ID as present and well formatted.
- The bus driver can reliably hash the unique ID into a globally unique identifier (*GUID*), and returns this GUID in response to the [IRP\\_MN\\_QUERY\\_ID](#) function code when the **Parameters.QueryId.IdType** member of the [IO\\_STACK\\_LOCATION](#) structure is set to **BusQueryContainerID**.

Windows 7 and later versions of Windows provide inbox drivers for several of the most common bus types. This includes USB, Bluetooth, and PnP-X. For these bus types, the device is only required to include a bus-specific unique ID. The supplied Windows bus driver will then read the unique ID from the device and create a container ID.

The following topics describe how the inbox bus drivers generate container IDs for certain bus types:

[Container IDs for USB Devices](#)

[Container IDs for Bluetooth Devices](#)

[Container IDs for PnP-X Devices](#)

[Container IDs for 1394 Devices](#)

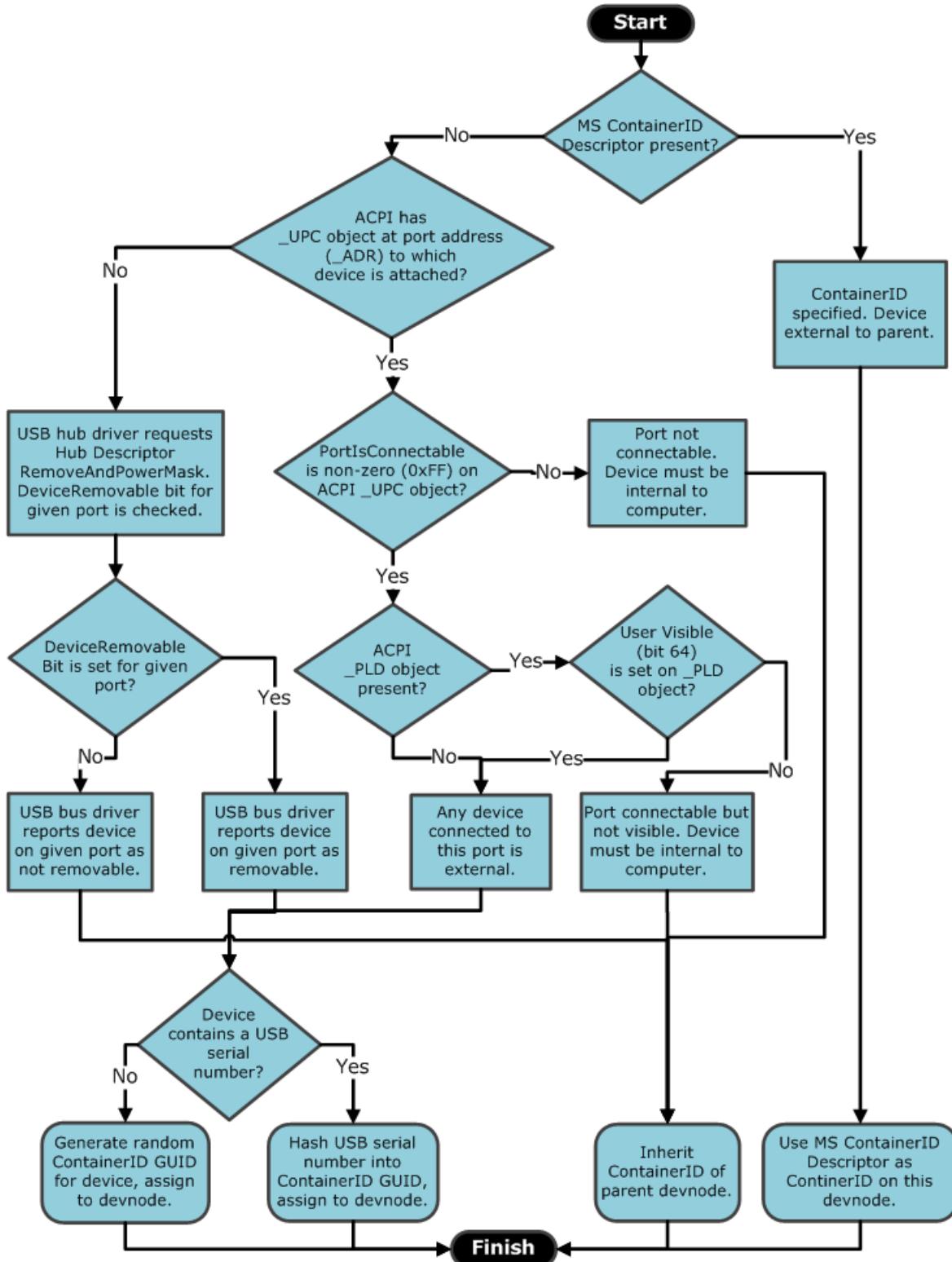
[Container IDs for eSATA Devices](#)

[Container IDs for PCI Express Devices](#)

# How USB Devices are Assigned Container IDs

11/2/2020 • 4 minutes to read • [Edit Online](#)

For a device that is connected to the computer through the Universal Serial Bus (USB), the following flowchart shows the heuristic used to assign a container ID to a USB device node (*devnode*).



This heuristic uses information from several sources to determine whether one of the following is true about a USB devnode:

- Does the devnode represent a new device on the USB bus? If this is true, the devnode would receive a new

container ID.

- Is the devnode a child devnode of an existing device? If this is true, the devnode would inherit the container ID of the parent devnode.

A container ID for a USB device is generated in several ways. This decision is based on information that is contained in the device. This information is retrieved from ACPI settings, the USB bus driver and the USB hub.

This heuristic follows these steps for each devnode that the Plug and Play (PnP) manager enumerates on the USB bus.

1. When queried by the USB bus driver, the USB device can report a container ID through the Microsoft operating system (OS) **ContainerID** descriptor.

Starting with Windows 7, the operating system supports the Microsoft OS **ContainerID** descriptor. Through this descriptor, the independent hardware vendor (IHV) can precisely specify the container ID for a device. Therefore, the device's container ID is unique and does not change on every computer the device is installed in. Also, if it reports a Microsoft OS **ContainerID** descriptor, the device indicates to the operating system that all enumerated devnodes are part of the same physical device.

The Microsoft OS **ContainerID** descriptor is intended to be used in devices that support simultaneous connection of the device through multiple system buses. For example, a printer may support simultaneous USB and IP network connections by using Plug and Play Extensions (PnP-X). By using a single Microsoft OS **ContainerID** descriptor, the same container ID is reported on both transports. Therefore, the PnP manager determines that the devnodes enumerated by each bus are part of the same physical device.

For more information about the Microsoft OS **ContainerID** descriptor, see the [Microsoft OS Descriptors](#).

2. If the USB device does not report a Microsoft OS **ContainerID** descriptor, the USB hub driver queries ACPI to determine whether the device is attached to an external-facing port.

The operating system tries to locate an ACPI address (**\_ADR**) object that matches the address of the USB port to which the device is connected. If a matching address object is found, the operating system performs the following steps:

- The USB port capabilities (**\_UPC**) object is queried and the **PortIsConnectable** value is checked. If **PortIsConnectable** has a nonzero value of 0xFF, the port can be used to connect external devices. Therefore, any device connected to this port must be external to the computer.
- If the computer implements ACPI 3.0 and the **PortIsConnectable** byte is nonzero, the operating system additionally queries the physical location description (**\_PLD**) object. The operating system checks if the **UserVisible** bit (bit 64) is set on the **\_PLD** object. It does this as an additional check to ensure that the port is both connectable and externally visible to the user.

If the information that is collected from ACPI indicates that the device is external, the PnP manager generates a container ID for the device. The **ContainedID** value is either a hash of the device's USB serial number or a randomly generated value. The devnode is assigned this container ID.

**Note** If the operating system determines that the device is internal to the computer, the devnode will inherit the container ID of the parent devnode, which (in this case) is the container ID of the computer itself.

3. If ACPI does not return an **\_ADR** object that matches the USB port address to which the device is connected, the PnP manager generates a container ID based on the removable status of the devnode.

The USB hub driver queries the USB **RemoveAndPowerMask** descriptor from the hub, and checks whether the **DeviceRemovable** bit is set for the port to which the device is connected. If the **DeviceRemovable** bit is set, devices attached to the port are removable from the hub. If the **DeviceRemovable** bit is not set, devices attached to the port are not-removable from the hub.

The USB bus driver reports the port removable/not-removable status to the PnP manager, which generates a **ContainerId** for the devnode through the following steps:

- If the hub indicates that devices attached to the given port are removable from the hub, the PnP manager determines that devices attached to this port are external to the computer. The container ID it generates for the devnode is either a hash of the USB serial number of the device, or is a randomly generated value.
- If the hub indicates that devices attached to the given port are not removable from the hub, the PnP manager determines that devices attached to this port are sub-functions of a multifunction device. In this case, the devnode inherits the container ID of the parent devnode.

For more information about the ACPI 3.0 interface, see [Advanced Configuration and Power Interface Specification Revision 3.0b](#).

# Using ACPI to Configure USB Ports on a Computer

11/2/2020 • 3 minutes to read • [Edit Online](#)

If the system requires ACPI BIOS changes to accurately reflect the USB port configuration, you should consider the user's ability to connect a device to the port when you configure the port.

If you use ACPI to specify the configuration of a USB port, you must define the USB port capabilities (`_UPC`) and physical location description (`_PLD`) objects. Although the ACPI 6.0 specification does not specifically prohibit the use of only the `_UPC` object, the use of both objects more precisely indicates the user's ability to connect devices to the port. Using only the `_UPC` object might not set the device container grouping correctly or as expected.

Devices that are attached to the port are removable from the hub if the `DeviceRemovable` bit is set. The following table shows how the values of the ACPI objects for a given port affect the value of the USB hub descriptor `DeviceRemovable` bit that Windows reports for the device.

USB PORT STATUS	EXAMPLE	<code>_UPC.PORTISCONNEXTABLE BYTE</code>	<code>_PLD.USERVISIBLE BIT (BIT 64)</code>	RESULTING DEVICEREMOVABLE BIT VALUE
Port is visible and the user can freely connect and disconnect devices.	Port is exposed on the face of a panel on the computer that is visible to the user.	Set (0xFF)	Set (1)	Set
Port is hidden or internal and user cannot freely connect and disconnect devices.	Port is directly hard-wired to an integrated device, such as a laptop webcam or an internal USB hub.	Set (0xFF)	Cleared	Cleared
Port is physically implemented by the USB host controller, but is not used.	Port is an excess port that is not connected to a port plug terminal or an integrated device.	Cleared (0x00)	Clear	Cleared

**Note** It is an invalid configuration to define a port as not connectable but visible to the user.

The following examples show correctly formed ACPI Source Language (ASL) that demonstrates the use of the `_UPC` and `_PLD` objects to describe a USB port:

- To specify a port that is internal (not user visible) and can be connected to an integrated device, the `_UPC.PortIsConnectable` byte must be set to 0xFF and the `_PLD.UserVisible` bit must be set to 0.

In the following example the device is grouped with the computer's device container.

```

Name(_UPC, Package(){
    0xFF,           // Port is connectable
    0xFF,           // Connector type (N/A for non-visible ports)
    0x00000000,    // Reserved 0, must be zero
    0x00000000})  // Reserved 1, must be zero

Name(_PLD, Buffer(0x10){
    0x81, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x30, 0x1C, 0x00, 0x00, 0x00, 0x00, 0x00})

```

- To specify a port that is external (user visible) and can be connected to an external device, the **\_UPC.PortIsConnectable** byte must be set to 0xFF and the **\_PLD.UserVisible** bit must be set to 1. The **\_UPC.PortConnectorType** byte must be set to the appropriate USB connector type as specified in Section 9.13 of the ACPI 3.0 specification.

In the following example the device is assigned a new device container and is displayed as a separate physical device.

```

Name(_UPC, Package(){
    0xFF,           // Port is connectable
    0x00,           // Connector type, Type 'A' in this case
    0x00000000,    // Reserved 0, must be zero
    0x00000000})  // Reserved 1, must be zero

Name(_PLD, Buffer(0x10){
    0x81, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x31, 0x1C, 0x00, 0x00, 0x00, 0x00, 0x00})

```

A USB Type-C connector must be correctly described in ACPI in order to pass the [USB Type-C ACPI Validation Hardware Lab Kit](#) test.

Example \_UPC for a USB Type-C connector:

```

Name(_UPC, Package(4){
    0x01,           // Port is connectable
    0x09,           // Connector type: Type C connector - USB2 and SS with Switch
    0x00000000,    // Reserved0 - must be zero
    0x00000000})  // Reserved1 - must be zero

```

For more information about the ACPI 6.0 interface, see [Advanced Configuration and Power Interface Specification Revision 6.0](#).

# Best Practices for Supporting Container IDs on USB Devices

1/11/2019 • 2 minutes to read • [Edit Online](#)

The heuristic used to generate the container ID for a USB device node (*devnode*), which was discussed in [How USB Devices are Assigned Container IDs](#), is complex, and relies on information taken from several sources.

Starting with Windows 7, the recommendations that are discussed in this section should be followed to allow the operating system to correctly determine the device container and provide the best user experience for your device.

This section includes the following topics that discuss these recommendations:

[Using Microsoft OS ContainerID Descriptors](#)

[Using the USB Removable Capability for Device Container Grouping](#)

[Avoiding Device Container Conflicts](#)

# Using Microsoft OS ContainerID Descriptors

1/11/2019 • 2 minutes to read • [Edit Online](#)

The Microsoft operating system (OS) **ContainerID** descriptor can be used in devices that support simultaneous connections of the device through multiple system buses. An explicitly defined Microsoft OS **ContainerID** descriptor ensures that all the device nodes (*devnodes*) enumerated for the device on the USB bus are grouped into the same device container.

**Note** If you decide to implement an Microsoft OS **ContainerID** descriptor, the descriptor value must be unique on every device to avoid container ID conflicts.

The Microsoft OS **ContainerID** descriptor is useful when a device supports simultaneous connections to the device through more than one bus. In this way, the same container ID is used on each bus supported by the device. This allows the operating system to determine whether functions on each bus are part of the same device container.

If you decide to use a Microsoft OS **ContainerID** within your USB device, you should be aware of the following points:

- For devices that are not integrated into the computer (that is, all external devices), it is a best practice to always provide a Microsoft OS **ContainerID** descriptor and a serial number in the USB device hardware. This will ensure that the Windows Plug and Play (PnP) infrastructure is able to correctly group all the device functions exposed by the device. Starting with Windows 7, components of the operating system rely on the proper grouping of device functions. Following this practice will provide the best user experience for devices on the Windows platform.
- USB devices integrated with a computer should never provide a Microsoft OS **ContainerID** descriptor. To ensure that integrated devices are correctly grouped with the computer's device container, integrated devices should rely on ACPI BIOS settings or on the USB hub descriptor **DeviceRemovable** bit for the port.

# Using the USB Removable Capability for Device Container Grouping

12/5/2018 • 2 minutes to read • [Edit Online](#)

The **USB Removable** capability allows the operating system to create a device container for legacy devices. This mechanism exists to provide backward compatibility for devices that cannot provide a Microsoft OS **ContainerID** descriptor or for devices integrated with a computer which does not have a USB port capabilities (**\_UPC**) value for the corresponding USB port.

It is important to recognize that the USB hub driver uses available removability information from the physical USB hardware in order to report a more accurate **Removable** capability for devices connected to each of its internal or external-facing ports. For more information, see [Container IDs Generated from the Removable Device Capability](#).

# Avoiding Device Container Conflicts

12/5/2018 • 2 minutes to read • [Edit Online](#)

Attaching two or more devices to a computer which share the same container ID (explicitly defined by using Microsoft operating system (OS) **ContainerID** descriptor or the same USB serial number) will result in a device container conflict. The operating system will interpret the device functions as originating from a single device and will create a single device container. This could cause unexpected behavior with the devices and within Windows.

To avoid this problem, ensure that the Microsoft OS **ContainerID** descriptor value and USB serial number are unique to a single physical device. Do not share these values among devices in a product line.

If your USB device relies on the operating system to generate a container ID based on the USB serial number, the container ID is generated as follows:

1. The operating system concatenates the USB device serial number, vendor ID, product ID, and revision number to generate a string.
2. The string that results is hashed into a GUID by using the UUID Version 5 (SHA-1) hash algorithm under a USB-specific namespace. The generated container ID will be unique, provided the independent hardware vendor (IHV) provides a unique serial number on each device.

# Container IDs for Bluetooth Devices

1/11/2019 • 2 minutes to read • [Edit Online](#)

For a Bluetooth device that is connected to the computer, the device's media access control (MAC) address is used to generate a container ID for the device.

The Bluetooth bus driver uses the MAC address as a seed value to generate a unique container ID for the device. This container ID is supplied by the Bluetooth bus driver for each Bluetooth device node (*devnode*) that is enumerated for a physical device.

Bluetooth devices frequently implement Bluetooth-specific services. These services are not installed as Windows PnP devices and therefore do not have associated devnodes. However, these services are effectively functional device instances, because they provide specific functionality and enable communication with the Bluetooth device.

Starting with Windows 7, the operating system considers Bluetooth services to be functional device interfaces, and groups these services together with the Bluetooth devnodes for a device.

All Bluetooth devices must include a MAC address. Therefore, a container ID for Bluetooth devnodes and services is always based on the MAC address value. Unlike USB devices, the removable device capability is never used to generate container IDs for Bluetooth devices.

To ensure that a unique container ID is generated for every device, developers of Bluetooth devices must configure each device with a unique MAC address.

# Container IDs for PnP-X Devices

12/21/2018 • 2 minutes to read • [Edit Online](#)

PnP extensions (PnP-X) extends Windows Plug and Play (PnP) to support devices that are connected to the computer through an IP-based network. For more information about PnP-X, refer to the [PnP-X: Plug and Play Extensions for Windows specification](#).

PnP-X devices can specify a container ID as an XML element in their device metadata. Two protocols are supported:

- Device Profile for Web Services (DPWS).

For more information about DPWS, refer to the [DPWS specification](#).

For more information about supporting container IDs through DPWS, see [Container IDs for DPWS Devices](#).

- Universal PnP (UPnP).

For more information, refer to the [UPnP Device Architecture specification](#).

For more information about supporting container IDs through UPnP, see [Container IDs for UPnP Devices](#).

If a PnP-X device does not specify a container ID in the DPWS device metadata or the UPnP device description document, the PnP manager generates a container ID for the device through an algorithm specific to the protocol the device supports:

- For DPWS devices, the generated container ID is either created from the GUID in the device's endpoint reference address (EPR) or is a SHA-1 hash of the device's EPR (if not a GUID).
- For UPnP devices, the generated container ID is the device's Unique Device Name (UDN).

**Note** In Windows 10, the PnP manager will always generate a container ID for DPWS devices by using the above algorithms, even if a container ID has been specified in the device metadata.

For devices which operate on a single bus or PnP-X protocol, the PnP-X generated container ID is sufficient.

Multiprotocol devices may want to specify a container ID. In a multiprotocol device, the same container ID would be shared on each protocol to allow Windows to group all instances of the device into one device container. In this manner, a container ID for the device can be specified through both DPWS and UPnP.

# Container IDs for DPWS Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, a device that supports PnP extensions (PnP-X) and Device Profile for Web Services (DPWS) can specify a container ID by including the **ContainerId** XML element in the device metadata document. For more information about DPWS and the DPWS device metadata document, refer to the [DPWS specification](#).

## NOTE

Starting with Windows 10, the system ignores the container ID provided by a device and instead generates one on its own. It does this either by using the GUID from the device's endpoint reference address (EPR) or a SHA-1 hash of the device's EPR (if not a GUID).

The **ContainerId** XML element is declared as follows:

```
<df:ContainerId xmlns:df="">  
  xs:string  
</df:ContainerId>
```

The **ContainerId** XML element type is a string, for which the value is a globally unique identifier (*GUID*) formatted. This string is formatted as {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}.

The following is an example of a **ContainerId** XML element.

```
<df:ContainerId xmlns:df="">  
  {101392d0-5e91-11dd-ad8b-0800200c9a66}  
</df:ContainerId>
```

The <ContainerId> XML element is required to be in the <ThisDevice> section of the device metadata exchange Simple Object Access Protocol (SOAP) message. The following example shows the correct placement of the <ContainerId> element in a metadata exchange message.

## NOTE

This is not a complete DPWS metadata exchange document. For more information about DPWS, refer to the [DPWS specification](#).

```

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:wsdisco="http://schemas.xmlsoap.org/ws/2005/04/discovery"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:wsd="http://schemas.xmlsoap.org/ws/2006/02/devprof"
    xmlns:df="http://schemas.microsoft.com/windows/2008/09/devicefoundation">

    <soap:Header>
        <!-- Place SOAP header information here.-->
    </soap:Header>

    <soap:Body>
        <wsx:Metadata>

            <wsx:MetadataSection
                Dialect="http://schemas.xmlsoap.org/ws/2005/05/devprof/ThisModel">
                <wsd:ThisDevice>
                    <!-- Place ThisDevice metadata here.-->
                    <df:ContainerId>
                        <!-- Place the ContainerID GUID here.-->
                        {101392d0-5e91-11dd-ad8b-0800200c9a66}
                    </df:ContainerId>
                </wsd:ThisDevice>
            </wsx:MetadataSection>

        </wsx:Metadata>
    </soap:Body>
</soap:Envelope>

```

If the DPWS device metadata document does not include the **ContainerId** XML element, the Plug and Play (PnP) manager uses the value of the device's endpoint reference address as the container ID.

# Container IDs for UPnP Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, a device that supports PnP extensions (PnP-X) and Universal PnP (UPnP) can specify a container ID by including the **X\_containerId** XML element in the device description document. For more information about UPnP and the UPnP device description document, refer to the [UPnP Device Architecture specification](#).

The **X\_containerId** XML element is declared as follows:

```
<df:X_containerId xmlns:df="">  
  xs:string  
</df:X_containerId>
```

The **X\_containerId** XML element type is a string, for which the value is a globally unique identifier (*GUID*). This string is formatted as `{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}`.

The following is an example of an **X\_containerId** XML element.

```
<df:X_containerId xmlns:df="">  
  {101392d0-5e91-11dd-ad8b-0800200c9a66}  
</df:X_containerId>
```

The **X\_containerId** XML element is required to be in the `<device>` section of the UPnP device description document. The following example shows the correct placement of the **X\_containerId** element in a device description document.

## NOTE

This is not a complete UPnP device description document. For more information about UPnP, refer to the [UPnP Device Architecture specification](#).

```
<?xml version="1.0" ?>
<root
  xmlns="urn:schemas-upnp-org:device-1-0"
  xmlns:df=
  "http://schemas.microsoft.com/windows/2008/09/devicefoundation">

  <specVersion>
    <major>major version number</major>
    <minor>minor version number</minor>
  </specVersion>

  <URLBase>device URL</URLBase>

  <device>
    <!-- Place device metadata here. See UPnP spec for details.-->
    <df:X_containerID>
      <!-- Place the ContainerID GUID here.--->
      {101392d0-5e91-11dd-ad8b-0800200c9a66}
    </ df:X_containerID >

  </device>
</root>
```

If the UPnP device description document does not include the **X\_containerId** XML element, the Plug and Play (PnP) manager generates a container ID through the device's Unique Device Name (UDN).

# Container IDs for 1394 Devices

1/11/2019 • 2 minutes to read • [Edit Online](#)

The 1394 bus specification does not specify an internal hardware mechanism to indicate whether a device function is or is not removable from the 1394 host controller. The 1394 bus driver that is included with Windows marks every device node (*devnode*) as removable from the parent host controller.

If a single 1394 device exposes multiple device functions, each devnode that the bus driver enumerates is marked as removable. However, the 1394 bus driver that is included with Windows recognizes that each devnode originated from a single device and assigns the same container ID to each devnode. Therefore, each 1394 device receives a single device container object and is displayed as a single device in the Devices and Printers user interface (UI).

# Container IDs for eSATA Devices

12/21/2018 • 2 minutes to read • [Edit Online](#)

The External Serial Advanced Technology Attachment (eSATA) bus cannot report a container ID. When the Windows operating system determines the device container grouping for an eSATA device, it relies on the removable capability that the ATA bus driver returns.

The ATA bus driver determines that the eSATA device is removable by reading the following Advanced Host Controller Interface (AHCI) register bits.

AHCI REGISTER	BYTE OFFSET	BIT LOCATION	DESCRIPTION
HBA Capabilities (CAP))	0x000	5 - Supports External SATA (SXS)	<p>When set to 1, this bit value indicates that the host bus adapter (HBA) has one or more SATA ports with a signal-only connector that is externally available (such as an eSATA connector).</p> <p>If this bit is set to 1, software can reference the PxCMD.ESP bit to determine whether a specific port has its signal connector externally available as a signal-only connector (that is, power is not part of that connector).</p>
Port x Command and Status (PxCMD)	0x18	18 - Hot-Plug Capable Port (HPCP)	<p>When set to 1, this bit value indicates that the signal and power connectors for the port are externally available through a joint signal and power connector.</p> <div style="border: 1px solid black; padding: 5px;"><p><b>Note</b> This only applies to blindmate connectors that support hot-plug capabilities.</p></div>

AHCI REGISTER	BYTE OFFSET	BIT LOCATION	DESCRIPTION
Port x Command and Status (PxCMD)	0x18	21 - External SATA Port (ESP)	<p>When set to 1, this bit value indicates that the signal connector for the port is externally available on a signal-only connector (such as an eSATA connector). Because of this, the port may experience hot-plug events.</p> <p>If ESP is set to 1, the PxCMD.HPCP bit must be cleared to 0 and the CAPSXS bit must be set to 1.</p>

The ATA bus driver marks any device that is attached to the eSATA port as removable if one of the following is true:

- The HPCP bit is set to 1, which indicates that the eSATA port is an external port that supports hot-plug operations.
- The SXS and ESP bits are both set to 1, which indicates that the SATA port is an external signal-only port.

**Note** These conditions are mutually exclusive. An eSATA port may declare itself to be either an external hot-plug-capable port or an external signal-only port, but not both.

For more information about the SATA and eSATA interface, refer to the [Serial ATA Advanced Host Controller Interface \(AHCI\) 1.3 specification](#).

# Container IDs for PCI Express Devices

12/21/2018 • 2 minutes to read • [Edit Online](#)

The PCI Express (PCIe) bus cannot express a container ID. The Windows operating system relies on the removable capability that the PCI bus driver returns when it determines the device container grouping for a PCIe device.

The PCI bus driver determines that a PCIe device is removable by reading the following PCIe register bits.

PCIE REGISTER	BYTE OFFSET	BIT LOCATION	DESCRIPTION
PCI Express Capabilities	0x02	8 - Slot Implemented	When set to 1, this bit value indicates that the PCIe link that is associated with this port is connected to a physical slot, instead of being connected to an integrated component.
Slot Capabilities	0x14	6 - Hot-Plug Capable	When set to 1, this bit value indicates that this slot can support hot-plug operations.

The PCI bus driver marks a PCIe device as removable if both of the following conditions are satisfied:

- The Slot Implemented bit is set to 1.
- The Hot-Plug-Capable bit is set to 1:

The mechanism that is used to set these register bits varies by PCIe chipset version and manufacturer. For example, some chipsets let the firmware program these bits, whereas other chipsets require physical pins to be strapped to the voltage charge connection (Vcc) or ground (GND).

Be aware that if the device implements an \_EJ0 method in the ACPI namespace, the ACPI driver marks the device as removable. This occurs regardless of the setting of the Slot Implemented or Hot-Plug Capable bits. For more information, see the [Hot-Plug PCI and Windows](#) white paper.

For more information about the PCIe interface, see the [PCIe Base](#) specification.

# Container IDs Generated from the Removable Device Capability

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, if a bus driver cannot provide a container ID for a device node (*devnode*) that it is enumerating, the Plug and Play (PnP) manager uses the removable device capability to generate a container ID for all devnodes enumerated for the physical device. The bus driver reports the removable device capability in response to an [IRP\\_MN\\_QUERY\\_CAPABILITIES](#) request.

This section contains the following topics:

[Overview of the Removable Device Capability](#)

[How Container IDs are Generated from the Removable Device Capability](#)

# Overview of the Removable Device Capability

12/1/2020 • 2 minutes to read • [Edit Online](#)

The removable device capability is a bit (**Removable**) that bus drivers set in the **DEVICE\_CAPABILITIES** structure in response to the **IRP\_MN\_QUERY\_CAPABILITIES** function code for a specified device node (*devnode*).

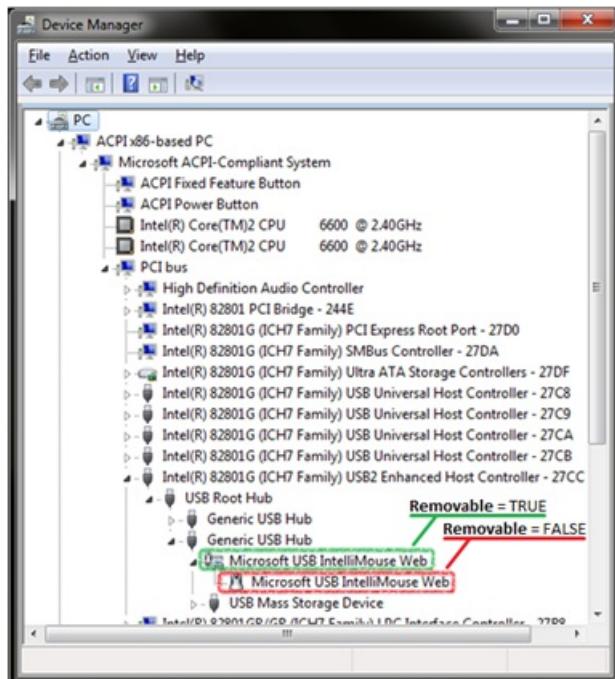
Bus drivers set the removable device capability for a devnode when the devnode and all its child devnodes make up a device that can be physically removed, disconnected, or unplugged from its parent devnode while the computer is running. Typically, a devnode should be marked as removable if it is the topmost devnode in a devnode topology.

Setting the removable device capability correctly on a devnode is important. If a bus driver cannot provide a container ID for a devnode that it is enumerating, the Plug and Play (PnP) manager uses the removable device capability to generate a container ID for all devnodes enumerated for the device.

For example, suppose that a single-function device, such as a mouse, is connected to the computer through USB. In this case, the USB bus driver detects the new device, detects that it is a USB human interface device (HID), and creates a USB HID devnode for the device. The HID devnode also detects that the HID device is a mouse and creates a child devnode for a HID-compliant mouse. At this point, the mouse is installed and is functional on the computer. Both of the new devnodes use independent *driver stacks*.

As a general rule, the topmost (parent) devnode of the device should be set as removable, while each of its child devnodes should not be set as removable. In the previous example, the USB bus driver sets the **Removable** bit to **TRUE** for the USB HID devnode, and sets the **Removable** bit to **FALSE** for the child HID-compliant mouse devnode.

The following Device Manager screen shot shows the devnode topology for a generic USB mouse, and shows which devnodes of the mouse are marked as removable.



# How Container IDs are Generated from the Removable Device Capability

1/11/2019 • 2 minutes to read • [Edit Online](#)

If a bus driver cannot provide a container ID for a device node (*devnode*) that it is enumerating, the Plug and Play (PnP) manager uses the removable device capability to generate a container ID for all devnodes enumerated for the device. For more information about the removable device capability, see [Overview of the Removable Device Capability](#).

The following heuristic describes how Container IDs are generated from the removable device capability:

1. If the devnode has the removable device capability set to **TRUE**, generate a new container ID for the devnode.
2. If the devnode has the removable device capability set to **FALSE**, inherit the container ID from its parent devnode.

A devnode cannot enumerate child devnodes until it is initialized and its *driver stack* is started. As soon as its container ID is assigned during initialization, the devnode is ready to propagate its container ID down to any of its nonremovable children as they are enumerated.

A devnode with the removable device capability set to **TRUE** is considered the topmost (parent) devnode for the device, and a container ID is generated for this devnode.

All the children of this parent devnode inherit the same container ID unless they themselves have their removable device capability set to **TRUE**. In this case, a removable child devnode is assigned a different container ID and becomes the parent devnode of this removable device. All the children of that devnode inherit the same container ID.

For example, suppose that a single-function mouse is connected to the computer through USB. In this case, the USB bus driver detects a new device and detects that it is a USB human interface device (HID). The USB bus driver then creates a USB HID devnode for the device. The HID devnode also detects that the HID device is a mouse and creates a child devnode for a HID-compliant mouse.

Applying this heuristic to this example results in the following actions:

1. The USB HID devnode is created. The removable device capability is set to **TRUE** on this devnode because its parent USB hub devnode recognized that it was plugged into an external-facing USB port.
2. A container ID is created for this devnode because it is the topmost devnode of a removable device. As a result, this devnode is considered the parent devnode for the removable device.
3. The HID-compliant mouse devnode is created. The removable device capability is set to **FALSE** on this devnode because its parent USB HID devnode reports all its children as nonremovable. In this case, the HID-compliant mouse devnode inherits the container ID of the parent devnode.

Through this heuristic, the same container ID is assigned to each devnode that belongs to the mouse. The PnP manager successfully grouped the devnodes into a logical device, even when there is no unique identifier for the device.

**Note** The success of this heuristic relies on a specific bus driver that correctly reports the removable device capability for each devnode that it enumerates. The bus driver must ensure that the parent devnode of the device should be set as removable, and its child devnodes should not be set as removable.

# Container IDs Generated from a Removable Device Capability Override

11/2/2020 • 5 minutes to read • [Edit Online](#)

Starting with Windows 7, new devices should provide a bus-specific unique ID (as described in [Container IDs Generated from a Bus-Specific Unique ID](#)).

Alternatively, devices and bus drivers must set the removable device capability correctly (as described in [Container IDs Generated from the Removable Device Capability](#)). For more information about the removable device capability, see [Overview of the Removable Device Capability](#).

Windows 7 and later versions of Windows also support a mechanism to override the reported removable device capability. This mechanism is useful for legacy devices that report the removable device capability incorrectly.

Although the override mechanism does not change the value of the removable device capability, it forces the PnP manager to use the override setting and not the value of the removable device capability when generating container IDs for devices.

Through this override mechanism, a container ID can be generated through a registry-based method. As soon as the container ID is generated for the topmost (parent) device node (*devnode*) of a device, the same container ID is inherited by each child devnode of the device through the heuristic described in [Container IDs Generated from the Removable Device Capability](#).

The override mechanism is a registry-based lookup table that consists of registry keys that map to specific devices. This override table is maintained under the [DeviceOverrides registry key](#), and consists of the following registry keys and subkeys.

TABLE LEVEL	REGISTRY KEY/SUBKEY NAME	DESCRIPTION
1	<a href="#">DeviceOverrides</a>	Parent key for all removable device capability overrides.
2	<a href="#">HardwareID</a>	Specifies the <a href="#">hardware ID</a> of a device to which the removable device capability override applies.  The name of this subkey is the actual hardware ID, with all backslash ("") characters replaced by number ("#") characters.
2	<a href="#">CompatibleID</a>	Specifies the <a href="#">compatible ID</a> of a device to which the removable device capability override applies.  The name of this subkey is the actual hardware ID, with all backslash ("") characters replaced by number ("#") characters.

TABLE LEVEL	REGISTRY KEY/SUBKEY NAME	DESCRIPTION
3	<a href="#">LocationPaths</a>	Specifies that only the location path of the device's parent device node ( <i>devnode</i> ) will have the removable device capability override applied.
3	<a href="#">ChildLocationPaths</a>	Specifies that the location path of the device's child devnodes will have the removable device capability override applied.
		<p><b>Note</b> The parent devnode of the specified device are not affected by the removable device capability override, unless a <a href="#">LocationPaths</a> registry subkey is also specified or a <a href="#">ChildLocationPaths</a> registry subkey is specified for the parent devnode.</p>
4	<a href="#">LocationPath</a>	<p>Specifies the discrete location path of the devnode to which the removable device capability override applies.</p> <p>The name of this subkey is the actual location path for a single devnode instance of a device installed in the computer.</p>
4	*	Specifies that the removable device capability override applies to all devnodes for the specified device.

Within the [LocationPath](#) and \* registry subkeys, a DWORD value (**Removable**) specifies whether the applicable devnodes are considered removable (1) or not removable (0).

### Example 1

The following shows a device override for a devnode that matches a [HardwareID](#) registry subkey in addition to a location path that is specified through the [LocationPaths](#) registry subkey.

In this example, the override will disable the removable device capability, and is applied to all devnodes that have a [hardware ID](#) of USB\VID\_1234&PID\_5678 at the location path PCIROOT(0)\PCI(102)\USBROOT(0)\USB(1).

The following is an example of the registry table format for this override.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceOverrides
  USB#VID_1234&PID_5678
    LocationPaths
      PCIROOT(0)\PCI(102)\USBROOT(0)\USB(1)
        Removable=0
```

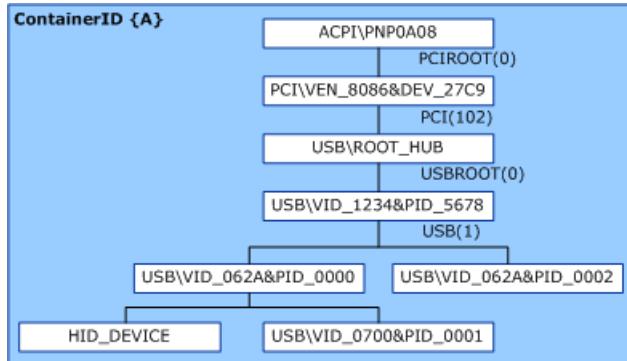
In this example, `USB#VID_1234&PID_5678` is the name of the [HardwareID](#) registry subkey, and `PCIROOT(0)\PCI(102)\USBROOT(0)\USB(1)` is the name of the [LocationPath](#) registry subkey.

This override changes the Plug and Play (PnP) manager's interpretation of the device topology. Notice that the

devnode with a [hardware ID](#) value of USB\VID\_1234&PID\_5678 was marked as not removable in the registry. A new container ID is not generated for this devnode, because the PnP manager interprets the devnode as not being removable from its parent. Instead, USB\VID\_1234&PID\_5678 (and all its children) inherit the container ID (ContainerID {A}) of its parent.

The result of this override is a single device grouping because all the devnodes in the tree have the same container ID. The device USB\VID\_1234&PID\_5678 is interpreted as being integrated with the computer.

The following diagram shows the resulting device topology and associated container ID assignment.



The previous example shows a frequently encountered devnode topology: portable computers with devices hardwired to specific bus locations that incorrectly report themselves as removable. Devices that are physically integrated with a computer, such as a Webcam or a biometric (fingerprint) sensor, should not be reported as removable because a user cannot physically separate them from the computer. The removable override lets an independent hardware vendor (IHV) or original equipment manufacturer (OEM) change how the PnP manager interprets the removable device capability, and thereby affects the container ID assignment for the device.

## Example 2

The following shows a removable device capability override for all devnodes that matches a specific [hardware ID](#) value.

In this example, the override will enable the removable device capability, and the override is applied to devnodes that have a hardware ID value of USB\VID\_062A&PID\_0000.

The following is a high-level description of the registry table format for this override.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceOverrides
USB#VID_062A&PID_0000
    LocationPaths
        *
        Removable=1
```

1 The name of the [HardwareID](#) registry subkey.

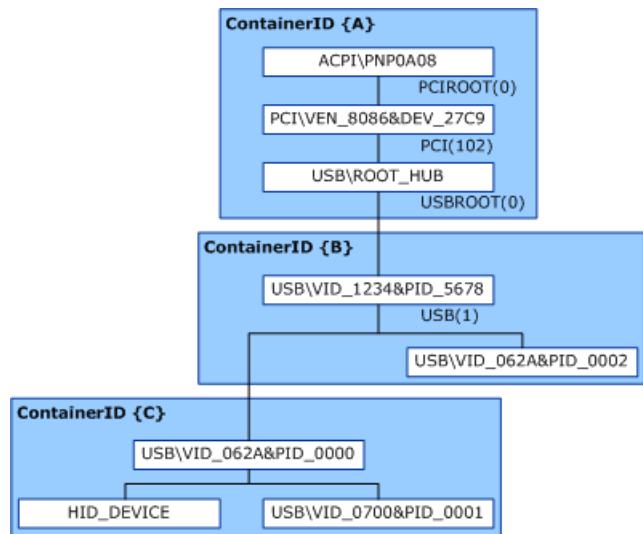
In this example, the devnode with a [hardware ID](#) of USB\VID\_1234&PID\_5678 reports the device removable capability correctly. The PnP manager generates a container ID (ContainerID {B}) for it and all its child devnodes.

However, the child devnode with a [hardware ID](#) of USB\VID\_062A&PID\_0000 matches the override. As a result, the PnP manager generates another contained ID (ContainerID {C}) for this devnode and all its child devnodes.

As before, this override changes the PnP manager's interpretation of the device topology. The physical device is assigned two container IDs, and is seen by Windows as two devices. Notice that the devnode with the [hardware ID](#) of USB\VID\_062A&PID\_0000 is interpreted as removable in grouping the devnodes into devices. This does not change the value reported by the devnode for the device removable capability.

Additionally, the \* registry subkey was specified to indicate that this override should be applied to all devnodes on the computer that have the [hardware ID](#) of USB\VID\_062A&PID\_0000.

The following diagram shows the resulting device topology and associated container ID assignment.



# Using ACPI for Device Container Grouping

12/21/2018 • 2 minutes to read • [Edit Online](#)

Using ACPI settings to specify device container groupings is intended for use by computer original equipment manufacturers (OEMs). By configuring ACPI objects in the computer BIOS, it is possible to indicate the precise configuration of the computer. This capability is in addition to other device container grouping mechanisms that are described in this paper.

The use of ACPI to indicate the computer configuration has several advantages:

- The settings persist in the computer BIOS and are preserved across operating system upgrades.
- Windows evaluates ACPI settings after it evaluates bus driver-supplied Removable capabilities. Therefore, a manufacturer can use the ACPI settings to fix devices that are incorrectly reported as Removable and Windows can group the functionality into the computer's device container.
- Using ACPI settings is especially useful for computer OEMs to indicate which USB ports are internal to the computer and which USB ports the user is capable of attaching external devices. For more information, see [Using ACPI to Configure USB Ports on a Computer](#).

The computer OEM is strongly encouraged to configure the ACPI BIOS settings to accurately reflect the USB port topology of the computer. This ensures that USB devices physically integrated with the computer (for example, an internal Bluetooth radio or an integrated webcam) are grouped into the computer's device container. It also allows the operating system to better determine the boundary between the computer and externally attached devices, because devices attached to connectable/user-visible ports are assumed to be external devices.

For more information about how to use ACPI object settings for device container grouping, see the [Container Groupings in Windows 7](#) white paper

# Verifying the Implementation of Container IDs

12/5/2018 • 2 minutes to read • [Edit Online](#)

Starting with Windows 7, you can see all the devices that are connected to the computer in the Devices and Printers user interface (UI). Devices appear as they physically appear to users, which is as a single "piece of plastic" that supports one or more functions. The icon that is displayed in the UI represents the primary function for the device.

To take advantage of the new capabilities that the Devices and Printers UI provides, devices must correctly implement container IDs.

The simplest way to verify that a device complies with the container ID requirements is to open the Devices and Printers UI to see how the device appears. If the device complies with the container ID requirements, only one icon should appear in the Devices and Printers UI for that device.

The following screen shot shows the Devices and Printers UI for a computer that has an attached USB keyboard and mouse. Notice that only one icon appears for each device.



In this example, the mouse is attached to a USB port on a desktop computer. However, only one instance of the mouse appears for the physical device. As a result, this device correctly implements the container ID requirements.

The physical device, which has one primary function, or "container," is represented by one object in the Devices and Printers UI. In this example, the mouse does not contain a Microsoft ContainerID operating system descriptor or serial number. Therefore, the Plug and Play (PnP) manager generates a container ID value by using the removable capability for the mouse.

For more information about the removable device capability, see [Container IDs Generated from the Removable Device Capability](#).

# Troubleshooting the Implementation of Container IDs

11/2/2020 • 2 minutes to read • [Edit Online](#)

If more than one instance of a device in the Devices and Printers user interface (UI) appears when you expect only one, the device does not correctly implement the container ID requirements. This incorrect implementation causes the Plug and Play (PnP) manager to group one or more device nodes (*devnodes*) into additional device containers for the device.

In such a case, you should examine the following:

- Is the removable device capability set correctly for each devnode that is enumerated for the device?

This is the most common cause of multiple device instances in the Devices and Printers UI. Make sure that each devnode for the device has the removable device capability set appropriately. The top-most, or *parent*, devnode of the device should be reported as removable, and all its children should be reported as not removable. Custom bus driver implementations must correctly assign the removable relationship for devnodes that they enumerate.

Device Manager is a valuable tool to diagnose these issues. You can examine the complete devnode hierarchy by following these steps:

1. Right-click the **My Computer** icon, and then click **Manage** . and select **Device Manager** from the System Tools listed in the resulting display.
2. Click **View by connection** from the drop-down menu.
3. Locate the devnodes that make up your device. For each devnode, right-click the node, and then click **Properties**.
4. On the **Details** tab, in the **Properties** drop-down list, click **Capabilities**.

If the list of capability values for the devnode contains the CM\_DEVCAP\_REMOVABLE flag, the devnode is marked as removable. The Plug and Play (PnP) manager then creates a new device container for the devnode and its children that cannot be removed.

For more information about the removable device capability, see [Container IDs Generated from the Removable Device Capability](#).

For more information about Device Manager, see [Using Device Manager](#).

- Does the device contain a container ID or other unique identifier in the hardware?

Make sure that the format of the container ID or unique identifier in the hardware complies with the format requirements for the given bus. For more information, see [Container IDs Generated from a Bus-Specific Unique ID](#).

If devnodes for the device are enumerated by a custom bus driver, check that the bus driver correctly responds to the **IRP\_MN\_QUERY\_ID** request for **BusQueryContainerID**.

- Is the device concurrently connected to the computer by more than one bus?

If the device is concurrently connected to the computer by two or more buses, two or more instances of the device can appear in the Devices and Printers UI. These instances can have one or more device instances for each bus to which the device is attached. To resolve this problem, make sure that the device reports a container ID or a device-specific unique identifier, and reports the same value on each bus.

# Generic Identifiers

11/2/2020 • 2 minutes to read • [Edit Online](#)

Most, but not all, identifier strings are bus-specific. The Plug and Play (PnP) manager also supports a set of generic device identifiers for devices that can appear on many different buses. These identifiers are of the form:

\*PNPd(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). You can find the official list of these identifiers in [Plug and Play Vendor IDs and Device IDs](#) on the Microsoft Download Center. see [Plug and Play ID - PNPID Request](#) for more information.

# Identifiers for PCI Devices

12/1/2020 • 2 minutes to read • [Edit Online](#)

## IMPORTANT

You can find a list of known IDs used in PCI devices at [The PCI ID Repository](#). To list IDs on Windows, use `devcon hwids *`.

The following is a list of the [device identification string](#) formats that the PCI bus driver uses to report hardware IDs. When the Plug and Play (PnP) manager queries the driver for the hardware IDs of a device, the PCI bus driver returns a list of hardware IDs in order of increasing generality.

```
PCI\\VEN_v(4)&DEV_d(4)&SUBSYS_s(4)n(4)&REV_r(2)  
PCI\\VEN_v(4)&DEV_d(4)&SUBSYS_s(4)n(4)  
PCI\\VEN_v(4)&DEV_d(4)&REV_r(2)  
PCI\\VEN_v(4)&DEV_d(4)  
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)
```

Where:

- v(4) is the four-character PCI SIG-assigned identifier for the vendor of the device, where the term *device*, following PCI SIG usage, refers to a specific PCI chip. As specified in [Publishing restrictions](#), `0000` and `FFFF` are invalid codes for this identifier.
- d(4) is the four-character vendor-defined identifier for the device.
- s(4) is the four-character vendor-defined subsystem identifier.
- n(4) is the four-character PCI SIG-assigned identifier for the vendor of the subsystem. As specified in [Publishing restrictions](#), `0000` and `FFFF` are invalid codes for this identifier.
- r(2) is the two-character revision number.
- c(2) is the two-character base class code from the configuration space.
- s(2) is the two-character subclass code.
- p(2) is the Programming Interface code.

## Examples

### NOTE

In these examples, you'll need to replace the placeholder SUBSYS values of `00000000`. As mentioned earlier, `0000` is invalid for the v(4) and n(4) identifiers.

The following is an example of a hardware ID for a display adapter on a portable computer. The format of this hardware ID is `PCI\VEN_v(4)&DEV_d(4)&SUBSYS_s(4)n(4)&REV_r(2)`:

```
PCI\\VEN_1414&DEV_00E0&SUBSYS_00000000&REV_04
```

The following is the hardware ID for the display adapter in the previous example with the revision information removed. The format of this hardware ID is PCI\VEN\_v(4)&DEV\_d(4)&SUBSYS\_s(4)n(4).

```
PCI\\VEN_1414&DEV_00E0&SUBSYS_00000000
```

**NOTE**

In Windows 10, some IDs that previously appeared in the Hardware IDs list now appear in the list of Compatible IDs.

## Reporting compatible IDs

The following is a list of the device identification string formats that the PCI bus driver uses to report compatible IDs. The variety of these formats provides substantial flexibility to specify compatible IDs. The PCI bus driver constructs a list of compatible IDs based on the information that the driver can obtain from the device. When the PnP manager queries the driver for the compatible IDs of a device, the PCI bus driver returns a list of compatible IDs in order of decreasing compatibility.

```
PCI\\VEN_v(4)&DEV_d(4)&REV_r(2)  
PCI\\VEN_v(4)&DEV_d(4)  
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)&DT_d(4) (applies only to a PCI Express device)  
PCI\\CC_c(2)s(2)p(2)  
PCI\\CC_c(2)s(2)&DT_d(4) (applies only to a PCI Express device)  
PCI\\CC_c(2)s(2)\`
```

Where:

- The definitions of the following fields in a compatible ID are identical to the definitions of the corresponding fields that used in a hardware ID: *v*(4), *r*(2), *c*(2), *s*(2), and *p*(2).
- *d*(4) in the DEV\_d(4) field is the four-character vendor-defined identifier for the device.
- *d*(4) in the DT\_d(4) field is the four-character device type, as specified in the PCI Express Base specification.

For the example of a display adapter on a portable computer, any of the following compatible IDs would match the information in an INF file for that adapter:

PCI\\VEN\_1414&DEV\_00E0&REV\_04  
PCI\\VEN\_1414&DEV\_00E0  
PCI\\VEN\_1414&DEV\_00E0&REV\_04&CC\_0300  
PCI\\VEN\_1414&DEV\_00E0&CC\_030000  
PCI\\VEN\_1414&DEV\_00E0&CC\_0300  
PCI\\VEN\_1414&CC\_030000  
PCI\\VEN\_1414  
PCI\\CC\_030000  
PCI\\CC\_0300

# Identifiers for SCSI Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 10, Version 2004 (OS build 19041.488 or higher), two additional identifiers are available for NVMe storage disk drives which support the [STOR\\_RICH\\_DEVICE\\_DESCRIPTION](#) structure:

`SCSI\t*v(8)p(40)`

Where:

- $t^*$  is a device type code of variable length
- $v(8)$  is an 8-character vendor identifier
- $p(40)$  is a 40-character product identifier

`SCSI\t*v(8)p(40)r(8)`

Where:

- $t^*$  is a device type code of variable length
- $v(8)$  is an 8-character vendor identifier
- $p(40)$  is a 40-character product identifier
- $r(8)$  is an 8-character revision level value

In versions of Windows prior to Windows 10, Version 2004 (OS build 19041.488 or higher), the device ID format for a small computer system interface (SCSI) device is as follows:

`SCSI\\t\*v(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
- $r(4)$  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 SCSI Port 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

SCSI TYPE CODE	DEVICE TYPE	GENERIC TYPE	PERIPHERAL ID
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 as follows:

`SCSI\\DiskSEAGATE_ST39102LW_____0004`

There are four hardware IDs in addition to the device ID:

`SCSI\\t\\*v(8)p(16)`

`SCSI\\t\\*v(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 SCSI Port driver supplies only one compatible ID, one of the variable-sized generic type codes from the previous table.

For example, the compatible ID for a disk drive is as follows:

`GenDisk`

The generic identifier is used in INF files for SCSI devices more than any other, because SCSI drivers are typically generic.

Be aware that the SCSI Port driver returns no generic name for sequential access and "processor" devices.

# Identifiers for IDE Devices

12/5/2018 • 2 minutes to read • [Edit Online](#)

Identifiers for integrated device electronics (IDE) devices resemble SCSI identifiers. The device ID format is as follows:

IDE\t\*v(40)r(8)

Where:

- *t\** is a device-type code of variable length.
- *v(40)* is a string that contains 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\t\*v(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 the SCSI Port 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

IDE TYPE CODE	DEVICE TYPE	GENERIC TYPE	PERIPHERAL ID
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**. The SCSI Port driver 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 the SCSI Port driver currently returns eighteen. In other respects, the generic names returned by the IDE bus driver are the same as those returned by the SCSI Port driver.

The compatible ID for an IDE tape drive is as follows:

GenSequential

In the special case of an LS-120 device, the IDE bus driver returns the following compatible ID:

GenSFloppy

The following shows the kind of identifiers that can be generated for an IDE hard disk drive:

IDE\DiskMaxtor\_91000D8\_\_\_\_\_SASX1B18

IDE\Maxtor\_91000D8\_\_\_\_\_SASX1B18

IDE\DiskMaxtor\_91000D8\_\_\_\_\_

Maxtor\_91000D8\_\_\_\_\_SASX1B18

GenDisk

# Identifiers for PCMCIA Devices

12/5/2018 • 2 minutes to read • [Edit Online](#)

For Personal Computer Memory Card International Association (PCMCIA) devices, the device ID can take several 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.
- *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 adapter might be this:

PCMCIA\MEGAHERTZ-CC10BT/2-BF05

For a multifunction card, every function has an identifier of the form:

PCMCIA\Manufacturer-Product-DEVd(4)-Crc(4)

The child function number (*d*(4) in this example) is a decimal number without leading zeros.

If the card does not have a name of the manufacturer, the identifier has one of these three forms:

PCMCIA\UNKNOWN\_MANUFACTURER-Crc(4)

PCMCIA\UNKNOWN\_MANUFACTURER-DEVd(4)-Crc(4)

PCMCIA\MTD-MemoryType(4)

The last of these three alternatives is for a flash memory card without a 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 **Models section** can also contain a hardware ID composed by replacing the 4-digit hexadecimal cyclic redundancy check (CRC) with a string that contains the 4-digit hexadecimal manufacturer code, a hyphen, and the 4-digit hexadecimal manufacturer information code (both from on-board 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 as described in the following table.

PCMCIA-COMPATIBLE ID	DEVICE TYPE
PNP0600	An AT Attachment (ATA) disk driver
PNP0D00	A multifunction 3.0 PC Card

PCMCIA-COMPATIBLE ID	DEVICE TYPE
*PNPC200	A modem card

# Identifiers for ISAPNP Devices

12/5/2018 • 2 minutes to read • [Edit Online](#)

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)\_DEVn(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 be this:

PNP0501

# Identifiers for 1394 Devices

12/5/2018 • 2 minutes to read • [Edit Online](#)

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.
- *UnitSwVersion* identifies the software specification.

The information that is 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. Therefore, 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 be:

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.
- *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&CmdSetIdn\*

SBP2\Gen

Gen

SBP2\<n\*>&d\*

Where:

- *n\** is the Command Set ID number.
- *Gen* is one of the generic names that are listed in the Generic Type column of the following table.
- *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 that corresponds to the *Gen* string identifier.

The fourth ID, listed in the previous example (SBP2\<n\*>&*d\**), is unique among all 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 the SCSI Port 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

# Identifiers for Secure Digital (SD) Devices

12/5/2018 • 2 minutes to read • [Edit Online](#)

When the SD bus driver detects an SD device in the host controller socket, it examines the device configuration of the card to construct a device and hardware IDs for the device and its functions. For SD combination cards and multifunction SDIO devices, the bus driver creates a PDO and a hardware ID for each respective function.

Because the internal configuration of an SD memory device is significantly different from that of an SDIO device, the SD bus driver uses two different hardware ID formats, one for SD memory devices and another for SDIO devices.

## SD device ids

The device ID of an SD memory device uses the following format:

SD\VID\_v(2)&OID\_o(4)&PID\_p(0-5)&REV\_n(1).m(1)

Where:

- $v(2)$  is a two-digit hexadecimal ID assigned by the SD Card Association (SDA) that identifies the card's manufacturer.
- $o(4)$  is a four-digit hexadecimal ID, also assigned by the SDA, that identifies the card's original equipment manufacturer (OEM) and/or the card contents.
- $p(0-5)$  is a vendor-supplied ASCII string, of 0 to 5 five characters, that indicates the product name, and  $n(1).m(1)$  is a two digit, vendor-supplied, revision number, with a decimal between the two digits (for example, 6.2).

The device ID of an SDIO device uses the following format:

SD\VID\_v(4)&PID\_p(4)

Where:

- $v(4)$  is a four-digit hexadecimal vendor code assigned by PCMCIA and JEIDA.
- $p(4)$  is the four-digit hexadecimal product and/or revision number that the vendor assigns to the device.

The SD bus driver extracts the vendor and product codes from the CISTPL\_MANFID tuple in the device's Card Information Structure (CIS) area.

## SD hardware IDs

For SD memory devices, the bus driver supplies two hardware IDs: one that is identical to the device ID, and another that is the same as the device ID, but without the revision information. The ID with revision information uses the following format:

SD\VID\_v(2)&OID\_o(4)&PID\_p(0-5)

Where, as with the device ID:

- $v(2)$  is a two-digit hexadecimal ID assigned by the SD Card Association (SDA) that identifies the card's manufacturer.
- $o(4)$  is a four-digit hexadecimal ID, also assigned by the SDA, that identifies the card's original equipment manufacturer (OEM) and/or the card contents.

- $p(0-5)$  is a vendor-supplied ASCII string, of 0 to 5 five characters, that indicates the product name.

For SDIO devices, the SD bus driver supplies a single hardware ID that is identical to the device ID.

### **SD compatible IDs**

In addition to device and hardware IDs, the SD bus driver generates a compatible ID under certain circumstances.

For SD memory devices, the bus driver always generates the following compatible ID:

SD\CLASS\_STORAGE

For SDIO devices, the SD bus driver generates the following compatible ID, provided the value in the function basic register (FBR) is not zero:

SD\CLASS\_c(2)

where  $c(2)$  is the two-digit hexadecimal device interface code.

# Identifiers for USB Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section describes the device, hardware and compatible identifiers generated by USB devices.

Since the Windows operating system generates special USB identifiers for printer and mass storage devices, the following documentation divides USB identifiers into two groups:

[Standard USB Identifiers](#)

[Special USB Identifiers](#)

For all USB devices, the USB bus driver generates a standard set of identifiers composed of values retrieved from the USB device and interface descriptors. Standard USB Identifiers are discussed in the first of the two sections indicated above. In addition to the standard USB identifiers, native Windows drivers for mass storage and printer devices generate a separate set of USB identifiers composed of information about special relevance to printers and storage devices. These special USB identifiers are discussed in the second section.

**Note** Starting with Microsoft Windows 2000, the numbers that are embedded in USB identifiers is in hexadecimal format.

# Standard USB Identifiers

11/2/2020 • 2 minutes to read • [Edit Online](#)

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, the system-supplied USB hub driver composes the following device ID by 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 vendor code that the USB committee assigns to the vendor.
- $d(4)$  is the 4-digit product code that the vendor assigns to the device.
- $r(4)$  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)

Where:

- $c(2)$  is the device class code taken from the device descriptor.
- $s(2)$  is the device subclass code.
- $p(2)$  is the protocol code.

The device class code, subclass code, and protocol code are determined by the *bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* fields of the device descriptor, respectively. These are 2-digit numbers.

## Multiple-Interface USB Devices

Devices with multiple interfaces are called *composite* devices. Starting with Windows 2000, when a new [USB composite device](#) is plugged into a computer, the USB hub driver creates a physical device object (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 operating 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. However, if no other match is found, the operating system uses 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 vendor code that the USB committee assigns to the vendor.
- $d(4)$  is the 4-digit product code that the vendor assigns to the device.
- $z(2)$  is the interface number that is extracted from the  $bInterfaceNumber$  field of the interface descriptor.

An INF model section can also specify the following compatible IDs:

USB\CLASS\_d(2)&SUBCLASS\_s(2)&PROT\_p(2)

USB\CLASS\_d(2)&SUBCLASS\_s(2)

USB\CLASS\_d(2)

USB\COMPOSITE

Where:

- $d(2)$  is the device class code taken from the device descriptor.
- $s(2)$  is the subclass code.
- $p(2)$  is the protocol code.

The device class code, subclass code, and protocol code are determined by the  $bInterfaceClass$ ,  $bInterfaceSubClass$ , and  $bInterfaceProtocol$  fields of the interface descriptor, respectively. These are 2-digit numbers.

# Identifiers Generated by USBSTOR.SYS

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows 2000, the operating system provides native support for many USB mass storage devices. The *Usbstor.inf* installation file contains device IDs for those devices that are explicitly supported. If the USB hub driver enumerates one of these devices, the operating system will automatically load the USB storage port driver, *Usbstor.sys*.

The device IDs for USB mass storage devices in *Usbstor.inf* take the usual form for USB device IDs composed by using information in the USB device's device descriptor:

USB\VID\_v(4)&PID\_d(4)&REV\_r(4)

Where:

- *v(4)* is the 4-digit vendor code that the USB committee assigns to the vendor.
- *d(4)* is the 4-digit product code that the vendor assigns to the device.
- *r(4)* is the revision code.

In addition to these device IDs, *Usbstor.inf* contains compatible IDs for class 8 ATAPI CD-ROM and removable media devices that support bulk-only transport:

USB\CLASS\_08&SUBCLASS\_02&PROT\_50

USB\CLASS\_08&SUBCLASS\_05&PROT\_50

USB\CLASS\_08&SUBCLASS\_06&PROT\_50

Where:

- class 08h = mass storage devices.
- subclass 02h = SFF-8020i ATAPI CD-ROM devices.
- subclass 05h = SFF-8070i ATAPI removable media.
- subclass 06h = generic SCSI media.
- protocol 50h = the bulk-only transport protocol.

If the data retrieved from the device's device descriptor matches any of these compatible IDs, the operating system will load *Usbstor.sys*.

As soon as it is loaded, the USB storage port driver creates a new PDO for each of the device's logical units. For more information, see the example device stack created by *Usbstor.sys* illustrated in [Device Object Example for a USB Mass Storage Device](#).

When the PnP manager queries for the device identification strings of the newly created PDOs, the USB storage port driver creates a new set of device, hardware and compatible IDs derived from the device's SCSI inquiry data. The device ID format is as follows:

USBSTOR\v(8)p(16)r(4)

Where:

- *v(8)* is an 8-character vendor identifier.

- $p(16)$  is a 16-character product identifier.

- $r(4)$  is a 4-character revision level value.

An example of a device ID for a disk drive would be as follows:

USBSTOR\SEAGATE\_ST39102LW\_\_\_\_\_0004

The hardware IDs that the USB storage port driver generates are as follows:

USBSTOR\t\*v(8)p(16)r(4)

USBSTOR\t\*v(8)p(16)

USBSTOR\t\*v(8)

USBSTOR\v(8)p(16)r(1)

v(8)p(16)r(1)

USBSTOR\GenericTypeString

GenericTypeString

Where:

- $t^*$  is a SCSI device type code of variable length.
- $v(8)$  is an 8-character vendor identifier.
- $p(16)$  is a 16-character product identifier.
- $r(4)$  is a 4-character revision level value. In these additional identifiers,  $r(1)$  represents just the first character of the revision identifier.

The following table contains the SCSI device type codes used by the USB storage port driver to generate identifier strings.

SCSI TYPE CODE	DEVICE TYPE	GENERIC TYPE
DIRECT_ACCESS_DEVICE (0)	Disk or SFloppy	GenDisk or GenSFloppy
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential	GenSequential
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom
OPTICAL_DEVICE (7)	Optical	GenOptical
MEDIUM_CHANGER (8)	Changer	GenChanger

SCSI TYPE CODE	DEVICE TYPE	GENERIC TYPE
Default Type (all values not listed previously)	Other	UsbstorOther

These examples show the hardware IDs that are generated by the USB storage port driver:

USBSTOR\DiskSEAGATE\_ST39102LW\_\_\_\_\_0004

USBSTOR\DiskSEAGATE\_ST39102LW\_\_\_\_\_

USBSTOR\DiskSEAGATE\_

USBSTOR\SEAGATE\_ST39102LW\_\_\_\_\_0

SEAGATE\_ST39102LW\_\_\_\_\_0

USBSTOR\GenDisk

GenDisk

The USB storage port driver generates two compatible IDs.

USBSTOR\t\*

USBSTOR\RAW

where  $t^*$  is a SCSI device type code of variable length.

The compatible IDs generated by the USB storage port driver are illustrated by the following examples:

USBSTOR\Disk

USBSTOR\RAW

# Identifiers Generated by USBPRINT.SYS

12/5/2018 • 2 minutes to read • [Edit Online](#)

Starting with Windows 2000, the operating system provides a kernel-mode USB print driver, *usbprint.sys* that connects the printer subsystem to the USB stack. The native USB printer driver frees vendors from the need for developing their own kernel-mode USB printer drivers. This allows vendors to develop high-level user-mode printer drivers that work with both USB and parallel printers.

The *usbprint.inf* installation file contains a compatible ID that matches all USB class 7 printer devices. If the USB hub driver enumerates one of these devices, the operating system will find a match for the ID that the hub driver generates in *usbprint.inf* and will load the USB printer driver, *usbprint.sys*. The compatible ID found in *usbprint.inf* has the following form:

USB\CLASS\_07

Where:

- class 07h = devices that belong to the USB printer class

As soon as it is loaded, the USB printer driver creates a new PDO for the printer device. When the Plug and Play (PnP) manager queries for the device identification strings of the newly created PDO, the USB printer driver creates a new hardware ID, derived from the device's IEEE 1284 string that is compatible with the string identifiers generated by the parallel bus enumerator. This hardware ID has the following format:

USBPRINT\NameModel(20)Checksum(4)

Where:

- *NameModel(20)* is the concatenation of the manufacturer name and the model of the device, truncated to a maximum of 20 characters.
- *Checksum(4)* is a 4-character cyclic redundancy check (CRC) code calculated from the manufacturer name and the model name.

Spaces in the string are replaced with underscores. For example, if the manufacturer's name is "Hewlett-Packard," the model name is "HP Color LaserJet 550," and the checksum is 3115, the hardware ID would be as follows:

USBPRINT\Hewlett-PackardHP\_Co3115

In the previous example, the space between "HP" and "Color" in the model name was replaced with an underscore to produce the truncated make/model string "Hewlett-PackardHP\_Co."

**Note** The CRC that is generated by the operating system may not match the CRC that is calculated as described in the preceding section, or by any other CRC algorithm. As a result of this, your printer driver may not be able to calculate the correct hardwareID to use with the INF file for the printer driver. To retrieve the hardwareID, it is better to search the setupapi.dev.log file that is associated with the USB printer that is being installed.

# Overview of Device Setup Classes

12/5/2018 • 2 minutes to read • [Edit Online](#)

To facilitate device installation, devices that are set up and configured in the same manner 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 does not have 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 symbolic 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.

# Creating a New Device Setup Class

12/1/2020 • 2 minutes to read • [Edit Online](#)

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-defined 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 that are 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 correctly 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 example, 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 by using an INF file. In addition to 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](#).

# Overview of Device Interface Classes

11/2/2020 • 2 minutes to read • [Edit Online](#)

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.

Starting with Windows 2000, 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 mouse devices 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 starts. 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.

Do not confuse device interfaces with the interfaces that drivers can export in response to an `IRP_MN_QUERY_INTERFACE` request. That IRP is used to pass routine entry points between kernel-mode drivers.

# Registering a Device Interface Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 *device installation applications* call `SetupDi Xxx` functions. For more information about these functions, see [SetupDi Device Interface 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. For more information, see [Creating a Device Object](#).

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 that is 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.

# Enabling and Disabling a Device Interface Instance

11/2/2020 • 2 minutes to read • [Edit Online](#)

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](#) together 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](#) request for the device. If a driver does not disable a device's interfaces when it handles these removal IRPs, it must not subsequently attempt to do this because the PnP manager will disable the interfaces when it removes the device.

A driver should not disable the interfaces when the device is stopped ([IRP\\_MN\\_STOP\\_DEVICE](#)); instead, it should leave any device interfaces enabled and queue I/O requests until it receives another [IRP\\_MN\\_START\\_DEVICE](#) request. Similarly, a driver should not disable its interfaces when the device is put in a sleep state. It should queue I/O requests until the device wakes up. For more information, see [Supporting Devices that Have Wake-Up Capabilities](#).

# Using a Device Interface

11/2/2020 • 2 minutes to read • [Edit Online](#)

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use `SetupDiXxx` functions to find out about registered, enabled device interfaces. See [SetupDi Device Interface Functions](#) for more information.

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 receive 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 that corresponds 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](#).

# Registering for Notification of Device Interface Arrival and Device Removal

11/2/2020 • 2 minutes to read • [Edit Online](#)

This topic describes how a user-mode application or driver registers for notification of device interface arrival and device removal.

Typically, a user-mode component calls [CM\\_Register\\_Notification](#) to find a device interface, and then sends I/O requests to the interface. To do so, the component registers for both **CM\_NOTIFY\_FILTER\_TYPE\_DEVICEINTERFACE** and **CM\_NOTIFY\_FILTER\_TYPE\_DEVICEHANDLE**, for notification of device interface arrivals and device removals respectively. The calling sequence might look like the following.

## Registering for Notification of Device Interface Arrival and Device Removal

1. Call [CM\\_Register\\_Notification](#) with **CM\_NOTIFY\_FILTER\_TYPE\_DEVICEINTERFACE** to register for device interface arrival notifications. When future interfaces in the specified class arrive, the system notifies your component.
2. Because the interface you want to send I/O to might already be present on the system, call [CM\\_Get\\_Device\\_Interface\\_List](#) or [SetupDiGetClassDevs](#) to retrieve a list of existing interfaces. **Note** If an interface arrives between step 1 and step 2, the interface is listed twice, from the registration in step 1 and the list of interfaces in step 2.
3. Once you find your desired interface, call [CreateFile](#) to open a handle for the device.
4. After successfully creating a device handle in step 3, call [CM\\_Register\\_Notification](#) a second time. This time, register for notifications of type **CM\_NOTIFY\_FILTER\_TYPE\_DEVICEHANDLE**, and provide the new device handle as the handle for which to receive notifications. When the device represented by the interface receives a query remove request, the system notifies your component.
5. Use this table as you implement your device handle notification callback.

ACTION VALUE THE CALLBACK RECEIVES	WHAT YOUR COMPONENT SHOULD DO
<b>CM_NOTIFY_ACTION_DEVICEQUERYREMOVE</b>	Call <a href="#">CloseHandle</a> to close the device handle. If you do not do this, your open handle prevents the query remove of this device from succeeding.

ACTION VALUE THE CALLBACK RECEIVES	WHAT YOUR COMPONENT SHOULD DO
CM_NOTIFY_ACTION_DEVICEQUERYREMOVEFAILED	<p>The query remove failed, so the device and its interface are still valid. To continue sending I/O to the interface, open a new handle to it.</p> <p>First, unregister the notifications for your old handle by calling <a href="#">CM_Unregister_Notification</a>. You must do this from a deferred routine because you cannot call <a href="#">CM_Unregister_Notification</a> from a notification callback for the notification handle you are unregistering. See the <b>Remarks</b> section of <a href="#">CM_Unregister_Notification</a> for more information.</p> <p>Then, either continuing in the deferred routine, or back in your notification callback, call <a href="#">CreateFile</a> to create a new handle. Then call <a href="#">CM_Register_Notification</a> with the new handle and <a href="#">CM_NOTIFY_FILTER_TYPE_DEVICEHANDLE</a>.</p> <p>Note that if you register for notifications on a device that is in the process of being query removed after the <a href="#">CM_NOTIFY_ACTION_DEVICEQUERYREMOVE</a> notifications have been sent, you may receive a <a href="#">CM_NOTIFY_ACTION_DEVICEQUERYREMOVEFAILED</a> notification without first receiving a <a href="#">CM_NOTIFY_ACTION_DEVICEQUERYREMOVE</a> notification.</p>
CM_NOTIFY_ACTION_DEVICEREMOVEPENDING	<p>Call <a href="#">CM_Unregister_Notification</a> to unregister the notifications for your handle. You must do this from a deferred routine. See the <b>Remarks</b> section of <a href="#">CM_Unregister_Notification</a> for more information. If you still have an open handle to the device, call <a href="#">CloseHandle</a> to close the device handle.</p>
CM_NOTIFY_ACTION_DEVICEREMOVECOMPLETE	<p>Call <a href="#">CM_Unregister_Notification</a> to unregister the notifications for your handle. You must do this from a deferred routine. See the <b>Remarks</b> section of <a href="#">CM_Unregister_Notification</a> for more information. If you still have an open handle to the device, call <a href="#">CloseHandle</a> to close the device handle.</p>

- Once you are finished with the device, call [CM\\_Unregister\\_Notification](#) to unregister the interface notification callback that you registered in step 1.

If you are following this procedure in a UMDF 2 driver, see [Using Device Interfaces](#) for a code example. A UMDF 2 driver might perform steps 1-4 in the driver's [EvtDevicePrepareHardware](#) callback routine, and step 6 in one of the driver's device removal callback routines.

## Related topics

[CM\\_Register\\_Notification](#)

[CM\\_Unregister\\_Notification](#)

[Using Device Interfaces](#)

# Windows Classes vs. Interface Classes

11/2/2020 • 2 minutes to read • [Edit Online](#)

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. Instead of 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.

Windows 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 example, *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 more information about GUIDs, see [Using GUIDs in Drivers](#).

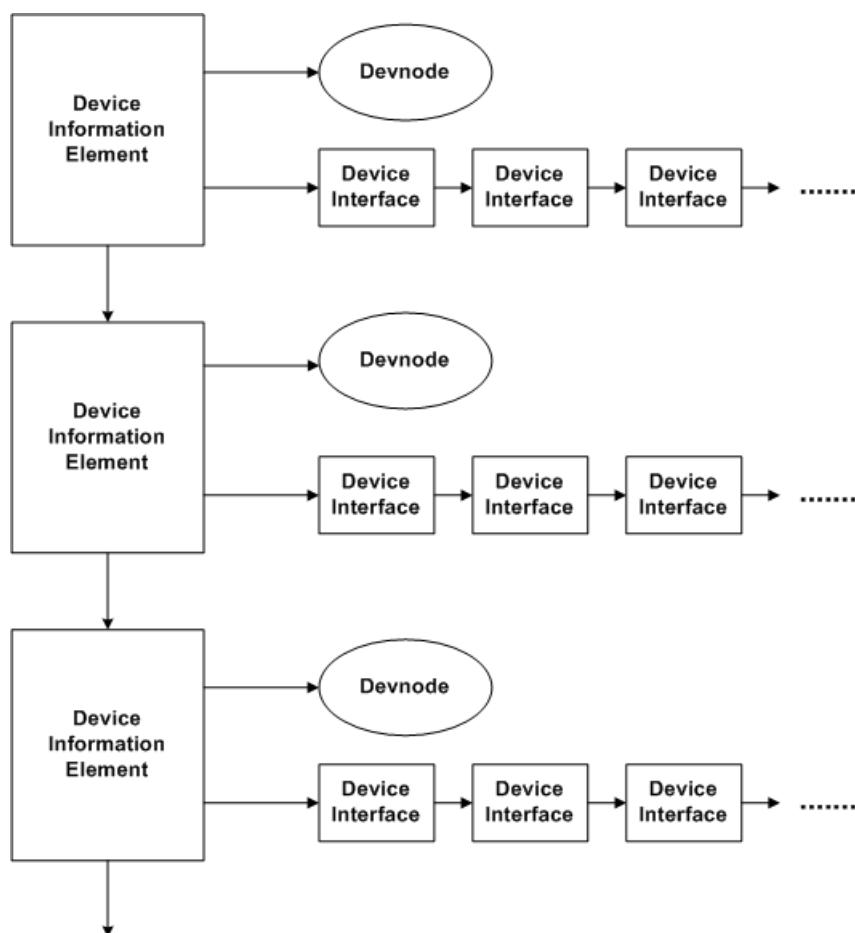
# Device Information Sets

11/2/2020 • 2 minutes to read • [Edit Online](#)

In user mode, devices that belong to either [device setup classes](#) or [device interface classes](#) are managed by using *device information elements* and *device information sets*. A device information set consists of device information elements for all the devices that belong 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 shows the internal structure of a 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 is 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** cannot 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 that has 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.

# Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

Device properties codify the attributes of device instances, [device setup classes](#), [device interface classes](#), and device interfaces. These attributes describe the function of the component and its configuration in the Windows operating system.

Windows Vista and later versions of Windows support a [unified device property model](#) that defines how these device properties are represented.

Microsoft Windows Server 2003, Windows XP, and Windows 2000 do not support this unified device property model. However these earlier Windows versions do support corresponding [device property representations](#) that depend on the component type and property type. To maintain compatibility with these earlier Windows versions, Windows Vista and later versions of Windows also support these earlier representations. However, you should use the unified device property model of Windows Vista and later to access device properties.

For reference information about the components of the unified device property model—including device property functions, system-defined device properties, data structures, and INF file directives—see [Device Property Reference](#).

# Unified Device Property Model

12/1/2020 • 2 minutes to read • [Edit Online](#)

Windows Vista and later versions of Windows support a unified device property model that characterizes the system configuration of *device instances*, [device setup classes](#), [device interface classes](#), and *device interfaces*. For information about the unified device property model, see the following topics:

- [System-Defined Device Properties](#)
- [Creating Custom Device Properties](#)
- [Property Keys](#)
- [Property-Data-Type Identifiers](#)
- [Property Value Requirements](#)
- [Properties and Related System-Defined Items](#)
- [INF File Entry Values That Modify Device Properties](#)
- [Using SetupAPI to Access Device Properties](#)
- [Using the INF AddProperty Directive and the INF DelProperty Directive](#)

Many of the system-defined device properties in the unified device property model have corresponding representations that can be used to access the same information on Microsoft Windows Server 2003, Windows XP, and Windows 2000. To maintain compatibility with these earlier Windows versions, Windows Vista and later versions of Windows also support these representations. However, you should use the unified device property model of Windows Vista and later to access device properties.

# System-Defined Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports system-defined properties that characterize the configuration or operation of device instances, [device setup classes](#), [device interface classes](#), and device interfaces. Each property is represented by a [property key](#), which is a GUID value that identifies a property category and a property identifier. The system-defined property key categories are reserved for system use only.

The following system-defined device property keys are defined in *Devpkey.h*:

- The DEVPKEY\_NAME property key that represents the name of a component. Use the value of the DEVPKEY\_NAME property to identify the component to an end-user. Windows supports the DEVPKEY\_NAME property for [device instances](#), [device setup classes](#), and [device interfaces](#).
- Property keys that represent the [device instance properties that correspond to the SPDRP\\_Xxx identifiers](#). (The SPDRP\_Xxx identifiers are defined in *Setupapi.h*.)
- Property keys that represent the device instance properties that do not have corresponding SPDRP\_Xxx identifiers. This includes the following:

[Device status and problem properties](#)

[Device relations properties](#), including the parent device, child devices, and sibling devices

[Device driver properties](#)

[Device driver package properties](#)

[Miscellaneous other device properties](#)

- Property keys that represent [device setup class properties](#) that correspond to the SPCRP\_Xxx identifiers. (The SPCRP\_Xxx identifiers are defined in *Setupapi.h*.)
- Property keys that represent device setup class properties that do not have corresponding SPCRP\_Xxx identifiers.
- Property keys that represent [device interface class properties](#).
- Property keys that represent [device interface properties](#).

For information about how to create custom device properties, see [Creating Custom Device Properties](#).

# Creating Custom Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports creation of custom device property categories for device instances, [device setup classes](#), device interface classes, and device interfaces. A custom property can be accessed by calling the appropriate [SetupAPI property function](#). A custom device property can also be modified by using an [INF AddProperty directive](#) or an [INF DelProperty directive](#).

For more information about custom device properties, see the following topics:

[Creating Custom Device Property Categories](#)

[Using the SetupAPI Property Functions to Access Custom Device Properties](#)

[Using the INF AddProperty Directive or the INF DelProperty Directive to Modify a Custom Device Property](#)

## Creating Custom Device Property Categories

A custom device property category is a logically-related collection of custom device properties. To programmatically create a custom device property category, use the [DEFINE\\_DEVPROPKY](#) macro to create the property keys that represent the properties in the property category, as follows:

- Create a unique GUID value that represents the property category and set the GUID value of each property key to this unique GUID value. For information about how to create a new GUID value, see [Defining and Exporting New GUIDs](#).

**Note** The system-defined property categories are reserved for operating system use only.

- Set the property identifier of each property key to an integer value that is unique within the property category and that is greater than or equal to two.

You can also create a custom device property category for a device instance by using an [INF AddProperty directive](#).

## Using the SetupAPI Property Functions to Access Custom Device Properties

Access custom device properties in the same manner as described in [Using SetupAPI to Access Device Properties \(Windows Vista and Later\)](#). The following additional considerations apply when you access custom device properties:

- The [SetupDiGetXxxPropertyKeys](#) and [SetupDiGetXxxPropertyKeysEx](#) functions retrieve the system-defined device property keys and custom device property keys that represent properties that are set for a component.
- The [SetupDiSetXxxProperty](#) functions set a custom device property for a component. Windows internally associates the custom device property key, the property data type, and the property value. If a custom device property with the same property key is already set, the [SetupDiSetXxxProperty](#) function overwrites the property value and property data type that is associated with the property.
- The [SetupDiGetXxxProperty](#) functions retrieve a custom device property that is set for a component. The [SetupDiGetXxxProperty](#) function retrieves the property value and the property data type that were set when the property was set.

## Using the INF AddProperty Directive or the INF DelProperty Directive to Modify a Custom Device Property

To modify a custom device property by using an [INF AddProperty directive](#), include an AddProperty directive

in the section that installs the component and supply the following entries for the property:

- The *property-category-guid* entry that represents the custom device property category
- A property identifier entry that identifies the property within the custom device property category
- The *value* entry of a new device property or the *value* entry that modifies an existing device property value

Use the **INF DelProperty directive** to delete a custom device property.

For more information about how to use these directives, see the [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

# Property Keys

11/2/2020 • 2 minutes to read • [Edit Online](#)

Programmatically, all device properties in the [unified device property model](#) are represented by property keys. The property keys are coded as **DEVPROPKEY** structures. The property keys are defined in *Devpkey.h*.

A DEVPROPKEY structure has the following members:

## fmtid

A DEVPROPGUID-typed variable that identifies the property category.

## pid

A DEVPROPID-typed variable that is the property identifier. For internal system reasons, a property identifier must be greater than or equal to two.

To create a custom device property key, use the **DEFINE\_DEVPROPKEY** macro.

The following is an example of how to use the **DEFINE\_DEVPROPKEY** macro to create a DEVPROPKEY structure. The name of the structure is "DEVPROPKEYStructureName", the sequence of values 0xde5c254e through 0xe0 supply the GUID value, and the value "2" is the property identifier.

```
DEFINE_DEVPROPKEY(DEVPROPKEYStructureName, 0xde5c254e, 0xab1c, 0xffff, 0x80, 0x20, 0x67, 0xd1, 0x46, 0xa8,  
0x50, 0xe0, 2)
```

**Note** The system-defined property key categories are reserved for system use only.

# Property-Data-Type Identifiers

11/2/2020 • 2 minutes to read • [Edit Online](#)

A property-data-type identifier is a **DEVPROPTYPE**-typed value that represents the data format of a property. In general, a property-data-type identifier is a bitwise OR of a **base-data-type identifier** and a **property-data-type modifier**. A property-data-type identifier can represent a single fixed-length base-data-type value, a single variable-length base-data-type value, an array of fixed-length base-data-type values, or a list of variable-length base-data-type values.

The system-supported base-data-type identifiers and property-data-type modifiers are defined in *Devpropdef.h*.

Windows enforces the following requirements on property-data-type identifiers:

- The base-data-type identifier is one of the **DEVPARAM\_TYPE\_Xxx** identifiers.
- If the base-data-type identifier is **DEVPARAM\_TYPE\_EMPTY** or **DEVPARAM\_TYPE\_NULL**, the property data-type identifier cannot include a property-data-type modifier.
- If the property-data-type identifier includes a property-data-type modifier, the property-data-type modifier is one of the **DEVPARAM\_TYPEMOD\_Xxx** identifiers.
- The **DEVPARAM\_TYPEMOD\_ARRAY** property-data-type modifier can be combined only with the fixed-length base data types.
- The **DEVPARAM\_TYPEMOD\_LIST** property-data-type modifier can be combined only with the variable-length base data types.

In addition to enforcing requirements on property data type identifiers, Windows also enforces **property value requirements** that depend on the property data type.

The SetupAPI property functions that retrieve and set a property value take a *.PropertyType* parameter. For the functions that retrieve a property value, *.PropertyType* is an output parameter that receives the property-data-type identifier for a property. For the functions that set a property value, *.PropertyType* is an input parameter that supplies the property-data-type identifier for a device property.

For more information, see [Using SetupAPI to Access Device Properties \(Windows Vista and Later\)](#).

# Property Value Requirements

11/2/2020 • 2 minutes to read • [Edit Online](#)

Windows enforces the device property value size requirements that are listed in the following table. Windows only sets a device property value if the device property value complies with these value size requirements.

PROPERTY DATA TYPE	PROPERTY VALUE SIZE REQUIREMENT
A fixed-length <a href="#">base-data-type</a> value	The specified size of the supplied data must be the number of bytes in the base data type.
An array of a fixed-length base-data-type values	The specified size of the supplied data must be the number of bytes of an array of zero or more base-data-type values.
A <a href="#">DEVPROP_TYPE_SECURITY_DESCRIPTOR</a> data type value	The specified size of the supplied data must be the number of bytes of a variable-length, self-relative SECURITY_DESCRIPTOR structure.
A <a href="#">DEVPROP_TYPE_STRING</a> data type value, a <a href="#">DEVPROP_TYPE_SECURITY_DESCRIPTOR_STRING</a> data type value, or a <a href="#">DEVPROP_TYPE_STRING_INDIRECT</a> data type value	The specified size of the supplied data must be the number of bytes of a Unicode REG_SZ string, including the NULL-terminator.
A list of DEVPROP_TYPE_STRING-typed strings, a list of DEVPROP_TYPE_SECURITY_DESCRIPTOR_STRING-typed strings, or a DEVPROP_TYPE_STRING_LIST data type value	The specified size of the supplied data must be the number of bytes of a Unicode REG_MULTI_SZ list of strings, including the final NULL-terminator that terminates the list of strings.
All property values	In addition to the property value size requirements that are listed in the other rows of this table, the maximum size, in bytes, of a property value is UNICODE_STRING_MAX_BYTES.

# Properties and Related System-Defined Items

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the unified device property model manages the correspondence between the following system-defined items that pertain to the installation and management of devices:

- The [system-defined properties](#) and their corresponding [property keys](#), [property data types](#), and property values.
- The SPDRP\_Xxx device instance property identifiers and the SPCRP\_Xxx device setup class property identifiers that are defined in *Setupapi.h*. The CM\_DRP\_Xxx device instance property identifiers and the CM\_CRP\_Xxx [device setup class](#) identifiers that are defined in *Cfgmgr32.h*.
- The REGSTR\_VAL\_Xxx registry entry value identifiers that pertain to device installation and management. These identifiers are defined in *Regstr.h*.
- Registry entry values that correspond to device properties.
- INF file entry values that modify device properties.

For information about the correspondence between the system-defined items that are associated with the device properties, see the following topics:

[Device Instance Properties](#)

[Device Setup Class Properties](#)

[Device Interface Class Properties](#)

[Device Interface Properties](#)

# INF File Entry Values That Modify Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following are the INF file entry values that modify device properties on Windows Vista and later:

- INF file entry values that set corresponding [system-defined device properties](#).
- [INF AddReg directives](#) and [INF DelReg directives](#) that set or delete system-defined registry entry values that correspond to system-defined device properties.
- INF AddReg directives and INF DelReg directives that set or delete custom registry entry values
- [INF AddProperty directives](#) and [INF DelProperty directives](#) that set and delete device properties. For more information about how to use these directives, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

For general information about the INF file sections that install device instances, [device setup classes](#), [device interface classes](#), and device interfaces, see the following topics:

[INF DDInstall Section](#)

[INF ClassInstall32 Section](#)

[INF InterfaceInstall32 Section](#)

[INF DDInstall.Interfaces Section](#)

## INF File Entry Values That Set Corresponding System-Defined Device Properties

Some INF file entry values provide information that Windows uses to set corresponding system-defined device properties. The following are a few examples of device properties whose values are supplied by such INF file entry values:

- The [DEVPKEY\\_Device\\_DeviceDesc](#) property for a device instance is set by the *device-description* entry value in the [INF Models Section](#).
- The [DEVPKEY\\_DeviceClass\\_ClassName](#) property for a [device setup class](#) is set by the *class-name* entry value in the INF Class directive in the [INF Version Section](#).
- The [DEVPKEY\\_DeviceInterface\\_ClassGuid](#) property for a device interface is set by the *InterfaceClassGuid* entry value in the [INF InterfaceInstall32 section](#).

## INF AddReg Directives and INF DelReg Directives That Modify System-Defined Device Properties

Many system-defined device properties have corresponding system-defined registry entry values. For a device property that has a corresponding registry entry value, using an [INF AddReg directive](#) to add the corresponding registry entry value sets the corresponding device property. Similarly, using an [INF DelReg directive](#) to delete a registry entry value, deletes the corresponding device property.

For example, the following [AddReg](#) directive would set the [DeviceCharacteristics](#) registry entry value and the corresponding [DEVPKEY\\_Device\\_Characteristics](#) property for a device instance that is installed by the "Abc\_Device\_Install.HW" section.

```
[Abc_Device_Install.HW]
...
AddReg = Xxx_AddReg
...
[Xxx_AddReg]
...
[HKR,,DeviceCharacteristics,0x10001,0x00000001
]
```

## INF AddReg Directives and INF DelReg Directives That Modify Custom Registry Entry Values

Windows Vista and later versions support using the [INF AddReg directive](#) and the [INF DelReg directive](#) to modify custom registry entry values that represent custom device properties. However, creating custom registry entry values to represent device properties is not supported by the unified device property model. If you create custom registry entry values for a device, you must manage the registry entry values in the same manner as you manage them on Windows Server 2003, Windows XP, and Windows 2000. To simplify the management of custom device properties, you should create device property keys to represent custom device properties instead of creating custom registry entry values.

# Using SetupAPI to Access Device Properties

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the SetupAPI property functions can be used to access the properties of device instances, device classes, and device interfaces. For information about how to use these property functions to access the [system-defined device properties](#), see the following topics:

[Accessing Device Instance Properties](#)

[Accessing Device Class Properties](#)

[Accessing Device Interface Properties](#)

For information about how to access device properties on Windows Server 2003, Windows XP, and Windows 2000, see [Using SetupAPI and Configuration Manager to Access Device Properties](#).

# Accessing Device Instance Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, applications and installers can access [device instance properties](#) by calling the following SetupAPI functions:

- [SetupDiGetDevicePropertyKeys](#)

The [SetupDiGetDevicePropertyKeys](#) function retrieves an array of the device property keys that identify the device properties that are currently set for a device instance. For information about how to determine what properties are set for a device, see [Determining Which Properties Are Set for a Device Instance](#).

- [SetupDiGetDeviceProperty](#)

The [SetupDiGetDeviceProperty](#) function retrieves a device property that is set for a device instance.

- [SetupDiSetDeviceProperty](#)

The [SetupDiSetDeviceProperty](#) function sets a device property for a device instance.

For information about how to access device properties on Windows Server 2003, Windows XP, and Windows 2000, see [Using SetupAPI and Configuration Manager to Access Device Properties](#).

# Determining Which Properties Are Set for a Device Instance

11/2/2020 • 2 minutes to read • [Edit Online](#)

To determine which properties are set for a device instance on Windows Vista and later versions of Windows, follow these steps:

1. Call [SetupDiGetDevicePropertyKeys](#) to determine how many properties are set for a device instance.

Supply the following parameter values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device instance for which to retrieve a list of property keys.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device instance for which to retrieve the list of property keys.
- Set *PropertyKeyArray* to **NULL**.
- Set *PropertyKeyCount* to zero.
- Set *RequiredPropertyKeyCount* to a pointer to a DWORD-typed variable.
- Set *Flags* to zero.

In response to the call to [SetupDiGetDevicePropertyKeys](#), [SetupDiGetDevicePropertyKeys](#) sets *\*RequiredPropertyKeyCount* to the number of properties that are set for the device instance, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDevicePropertyKeys](#) again and supply the same parameter values that were supplied in the first call, except for the following changes:

- Set *PropertyKeyArray* as a [DEVPROPKEY](#)-typed pointer to the buffer that receives the requested property key array.
- Set *PropertyKeyCount* to the size, in [DEVPROPKEY](#)-typed values, of the *PropertyKeyArray* buffer. The first call to [SetupDiGetDevicePropertyKeys](#) retrieved the required size of the *PropertyKeyArray* buffer in *\*RequiredPropertyKeyCount*.

If the second call to [SetupDiGetDevicePropertyKeys](#) succeeds, [SetupDiGetDevicePropertyKeys](#) returns the requested property key array in the *PropertyKeyArray* buffer, sets *\*RequiredPropertyKeyCount* to the number of property keys in the buffer, and returns **TRUE**. If the function call fails, [SetupDiGetDevicePropertyKeys](#) returns **FALSE** and calling [GetLastError](#) will return the logged error code.

# Retrieving A Device Instance Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

To retrieve the value of a device instance property on Windows Vista and later versions of Windows, follow these steps:

1. Call [SetupDiGetDeviceProperty](#) to determine the data type and the size, in bytes, of the property value.

Supply the following parameter values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device instance for which to retrieve a property.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device instance for which to retrieve a property.
- Set *PropertyKey* to a pointer to a [DEVPROPKEY](#) structure that represents the property.
- Set *.PropertyType* to a pointer to a [DEVPROPTYPE](#)-typed variable.
- Set *PropertyBuffer* to **NULL**.
- Set *PropertyBufferSize* to zero.
- Set *RequiredSize* to a pointer to a **DWORD**-typed variable.
- Set *Flags* to zero.

In response to the first call to [SetupDiGetDeviceProperty](#), [SetupDiGetDeviceProperty](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDeviceProperty](#) again and supply the same parameter values that were supplied in the first call, except for the following changes:

- Set *PropertyBuffer* to a pointer to the buffer that receives the property value.
- Set *PropertyBufferSize* to the required size, in bytes, of the *PropertyBuffer* buffer. The first call to [SetupDiGetDeviceProperty](#) retrieved the required size of the *PropertyBuffer* buffer in *\*RequiredSize*.

If the second call to [SetupDiGetDeviceProperty](#) succeeds, [SetupDiGetDeviceProperty](#) sets *\*PropertyType* to the property-data-type identifier for the property, returns the property value in the *PropertyBuffer* buffer, sets *\*RequiredSize* to the size, in bytes, of the property value that was retrieved, and returns **TRUE**. If the function call fails, [SetupDiGetDeviceProperty](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Setting a Device Instance Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

To set the value of a device instance property on Windows Vista and later versions of Windows, call [SetupDiSetDeviceProperty](#) and supply the following parameters values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device instance for which to set the property.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device instance for which to set the property.
- Set *PropertyKey* to a pointer to the [DEVPROPKEY](#) structure that represents the property to set.
- Set *PropertyType* to a pointer to a [DEVPROPTYPE](#)-typed variable that supplies the property-data-type identifier for the property to set.
- Set *PropertyBuffer* to a pointer to the buffer that contains the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the property value.
- Set *RequiredSize* to a DWORD-typed variable.
- Set *Flags* to zero.

If this call to [SetupDiSetDeviceProperty](#) succeeds, [SetupDiSetDeviceProperty](#) sets the device instance property and returns TRUE. If the function call fails, [SetupDiGetDeviceProperty](#) returns FALSE and a call to [GetLastError](#) will return the logged error code.

# Accessing Device Class Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, applications and installers can access [device setup class properties](#) and [device interface class properties](#) by calling the following SetupAPI functions:

- [SetupDiGetClassPropertyKeys](#) and [SetupDiGetClassPropertyKeysEx](#)

The [SetupDiGetClassPropertyKeys](#) function retrieves an array of the class property keys that identify the class properties that are currently set for a device setup class or a device interface class on a local computer. The [SetupDiGetClassPropertyKeysEx](#) function performs the same operation on a local computer or a remote computer. For information about how to determine what properties are set for a device class, see [Determining Which Properties Are Set for a Device Class](#).

- [SetupDiGetClassProperty](#)

The [SetupDiGetClassProperty](#) function [retrieves a class property](#) for a device setup class or a device interface class. The [SetupDiGetClassPropertyEx](#) function performs the same operation on a local computer as it does on a remote computer.

- [SetupDiSetClassProperty](#)

The [SetupDiSetClassProperty](#) function [sets a class property for a device setup class or a device interface class](#). The [SetupDiSetClassPropertyEx](#) function performs the same operation on a local computer as it does on a remote computer.

For information about how to access device class properties on Windows Server 2003, Windows XP, and Windows 2000, see [Accessing Device Setup Class Properties](#) and [Accessing Device Interface Class Properties](#).

# Determining Which Properties Are Set for a Device Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following topics describe how to determine which class properties are set for a device class in Windows Vista and later versions of Windows:

[Determining Which Class Properties Are Set for a Device Class on a Local Computer](#)

[Determining Which Class Properties Are Set for a Device Class on a Remote Computer](#)

## Determining Which Class Properties Are Set for a Device Class on a Local Computer

To determine which properties are set for a device class on a local computer, follow these steps:

1. Call [SetupDiGetClassPropertyKeys](#) to determine how many properties are set for a device class. Supply the following parameter values:

- Set *ClassGuid* to a pointer to a GUID that identifies the [device setup class](#) or [device interface class](#) for which to retrieve a list of the class property keys.
- Set *PropertyKeyArray* to NULL.
- Set *PropertyKeyCount* to zero.
- Set *RequiredPropertyKeyCount* to a pointer to a DWORD-typed variable.
- If the device class is a device setup class, set *Flags* to [DCLASSPROP\\_INSTALLER](#); otherwise, if the device class is a device interface class, set *Flags* to [DCLASSPROP\\_INTERFACE](#).

In response to this first call to [SetupDiGetClassPropertyKeys](#), [SetupDiGetClassPropertyKeys](#) sets *\*RequiredPropertyKeyCount* to the number of properties that are set for the device setup class, logs the error code [ERROR\\_INSUFFICIENT\\_BUFFER](#), and returns FALSE. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDevicePropertyKeys](#) again and supply the same parameters that were supplied in the first call, except for the following changes:

- Set *PropertyKeyArray* to a [DEVPROPKEY](#)-typed pointer to the buffer that receives the requested property key array.
- Set *PropertyKeyCount* to the size, in [DEVPROPKEY](#)-typed values, of the *PropertyKeyArray* buffer. The first call to [SetupDiGetClassPropertyKeys](#) returned the required size of the *PropertyKeyArray* buffer in *\*RequiredPropertyKeyCount*.

If the second call to [SetupDiGetClassPropertyKeys](#) succeeds, the function returns the requested property key array in the *PropertyKeyArray* buffer, sets *\*RequiredPropertyKeyCount* to the number of property keys in the buffer, and returns TRUE. If the function call fails, [SetupDiGetClassPropertyKeys](#) returns FALSE and a call to [GetLastError](#) will return the logged error code.

## Determining Which Class Properties Are Set for a Device Class on a Remote Computer

To determine the class properties that are set for a device class on a remote computer, follow the procedure that is described in [Determining Which Class Properties Are Set for a Device Class on a Local Computer](#) with the following modifications:

- Call [SetupDiGetClassPropertyKeysEx](#) instead of [SetupDiGetClassPropertyKeys](#).
- In addition to supplying the parameter values that are required for both

[SetupDiGetClassPropertyKeysEx](#) and [SetupDiGetClassPropertyKeys](#), supply the *MachineName* parameter, which must be set to a pointer to a NULL-terminated string that contains the UNC name, including the \\ prefix, of a computer.

# Retrieving a Device Class Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following topics describe how to retrieve a device class property value in Windows Vista and later versions of Windows:

[Retrieving a Device Class Property Value on a Local Computer](#)

[Retrieving a Device Class Property Value on a Remote Computer](#)

## Retrieving a Device Class Property Value on a Local Computer

To retrieve the value of a device class property on a local computer, follow these steps:

1. Call [SetupDiGetClassProperty](#) to determine the data type and the size, in bytes, of the property value.

Supply the following parameter values:

- Set *ClassGuid* to a pointer to a GUID that identifies the [device setup class](#) or [device interface class](#) for which to retrieve a class property that is set for the device class.
- Set *PropertyKey* to a pointer to a [DEVPROPKEY](#) structure that represents the property.
- Set *.PropertyType* to a pointer to a [DEVPROPTYPE](#)-typed variable.
- Set *PropertyBuffer* to NULL.
- Set *PropertyBufferSize* to zero.
- Set *RequiredSize* to a DWORD-typed variable.
- If the device class is a device setup class, set *Flags* to [DCLASSPROP\\_INSTALLER](#). Otherwise, if the device class is a device interface class, set *Flags* to [DCLASSPROP\\_INTERFACE](#).

In response to this first call to [SetupDiGetClassProperty](#), [SetupDiGetClassProperty](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code [ERROR\\_INSUFFICIENT\\_BUFFER](#), and returns FALSE. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetClassProperty](#) again and supply the same parameters that were supplied in the first call, except for the following changes:

- Set *PropertyBuffer* to a pointer to the buffer that receives the property value.
- Set *PropertyBufferSize* to the required size, in bytes, of the *PropertyBuffer* buffer. The first call to [SetupDiGetClassProperty](#) retrieved the required size of the *PropertyBuffer* buffer in *\*RequiredSize*.

If the second call to [SetupDiGetClassProperty](#) succeeds, [SetupDiGetClassProperty](#) sets *\*PropertyType* to the property-data-type identifier for the property, sets the *PropertyBuffer* buffer to the property value, sets *\*RequiredSize* to the size, in bytes, of the property value that was retrieved, and returns TRUE. If the function call fails, [SetupDiGetDeviceProperty](#) returns FALSE and a call to [GetLastError](#) will return the logged error code.

## Retrieving a Device Class Property Value on a Remote Computer

To retrieve a device class property value on a remote computer, follow the same procedure as is described in [Retrieving a Device Class Property Value on a Local Computer](#) with the following modifications:

- Call [SetupDiGetClassPropertyEx](#) instead of [SetupDiGetClassProperty](#).
- In addition to supplying the parameter values that [SetupDiGetDevicePropertyEx](#) and [SetupDiGetClassProperty](#) both require, supply the *MachineName* parameter, which must be set to a pointer to a NULL-terminated string that contains the UNC name, including the \\ prefix, of a computer.

# Setting a Device Class Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following topics describe how to set a device class property value in Windows Vista and later versions of Windows:

[Setting a Device Class Property Value on a Local Computer](#)

[Setting a Device Class Property Value on a Remote Computer](#)

## Setting a Device Class Property Value on a Local Computer

To set the value of a device class property, call [SetupDiSetClassProperty](#) and supply the following parameters values:

- Set *ClassGuid* to a pointer to a GUID that identifies the [device setup class](#) or [device interface class](#) for which to set a class property.
- Set *PropertyKey* to a pointer to the [DEVPROPKET](#) structure that represents the property to set.
- Set *.PropertyType* to a pointer to a [DEVPROPTYPE](#)-typed variable that supplies the property-data-type identifier for the property to set.
- Set *PropertyBuffer* to a pointer to the buffer that contains the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the property value.
- Set *RequiredSize* to a DWORD-typed variable.
- If the device class is a device setup class, set *Flags* to [DICLASSPROP\\_INSTALLER](#). Otherwise, if the device class is a device interface class, set *Flags* to [DICLASSPROP\\_INTERFACE](#).

If a call to [SetupDiSetClassProperty](#) succeeds, [SetupDiSetClassProperty](#) sets the device class property and returns TRUE. If the function call fails, [SetupDiSetClassProperty](#) returns FALSE and a call to [GetLastError](#) will return the logged error code.

## Setting a Device Class Property Value on a Remote Computer

To set a device class property value on a remote computer, follow the procedure that is described in [Setting a Device Class Property Value on a Local Computer](#) with the following modifications:

- Call [SetupDiSetClassPropertyEx](#) instead of [SetupDiSetClassProperty](#).
- In addition to supplying the parameter values that both [SetupDiSetClassPropertyEx](#) and [SetupDiSetClassProperty](#) require, supply the *MachineName* parameter, which must be set to a pointer to a NULL-terminated string that contains the UNC name, including the \\ prefix, of a computer.

# Accessing Device Interface Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, applications and installers can access [device interface properties](#) by calling the following SetupAPI functions:

- [SetupDiGetDeviceInterfacePropertyKeys](#)

The [SetupDiGetDeviceInterfacePropertyKeys](#) function retrieves an array of the device interface property keys that identify the device interface properties that are currently set for a device interface instance. For information about how to determine what properties are set for a device interface, see [Determining Which Properties are Set for a Device Interface](#).

- [SetupDiGetDeviceInterfaceProperty](#)

The [SetupDiGetDeviceInterfaceProperty](#) function [retrieves a device interface property](#) that is set for a device interface.

- [SetupDiSetDeviceInterfaceProperty](#)

The [SetupDiSetDeviceInterfaceProperty](#) function [sets a device interface property](#) for a device interface.

For information about how to access device interface properties on Windows Server 2003, Windows XP, and Windows 2000, see Accessing Device Interface Properties.

# Determining Which Properties are Set for a Device Interface

11/2/2020 • 2 minutes to read • [Edit Online](#)

To determine which properties are set for a device interface in Windows Vista and later versions of Windows, follow these steps:

1. Call [SetupDiGetDeviceInterfacePropertyKeys](#) to determine how many properties are set for a device class. Supply the following parameter values:

- Set *DeviceInfoSet* to a handle to a device information set that contains a device interface instance for which to retrieve a list of device interface property keys.
- Set *DeviceInterfaceData* to a pointer to an [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure that represents the device interface instance for which to retrieve a list of device property keys.
- Set *PropertyKeyArray* to **NULL**.
- Set *PropertyKeyCount* to zero.
- Set *RequiredPropertyKeyCount* to a pointer to a DWORD-typed variable.
- Set *Flags* to zero.

In response to this call to [SetupDiGetDeviceInterfacePropertyKeys](#),

[SetupDiGetDeviceInterfacePropertyKeys](#) sets *\*RequiredPropertyKeyCount* to the number of properties that are set for the device interface, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDeviceInterfacePropertyKeys](#) again and supply the same parameter values that were supplied in the first call, except for the following changes:

- Set *PropertyKeyArray* to a [DEVPROPKEY](#)-typed pointer to the buffer that receives the requested property key array.
- Set *PropertyKeyCount* to the size, in [DEVPROPKEY](#)-typed values, of the *PropertyKeyArray* buffer. The first call to [SetupDiGetDeviceInterfacePropertyKeys](#) retrieved the required size of the *PropertyKeyArray* buffer in *\*RequiredPropertyKeyCount*.

If the second call to [SetupDiGetDeviceInterfacePropertyKeys](#) succeeds,

[SetupDiGetDeviceInterfacePropertyKeys](#) returns the requested property key array in the *PropertyKeyArray* buffer, sets *\*RequiredPropertyKeyCount* to the number of property keys in the buffer, and returns **TRUE**. If the function call fails, [SetupDiGetDeviceInterfacePropertyKeys](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Retrieving a Device Interface Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, follow these steps to retrieve the value of a [device interface property](#):

1. Call [SetupDiGetDeviceInterfaceProperty](#) to determine the data type and the size, in bytes, of the property value. Supply the following parameter values:
  - Set *DeviceInfoSet* to a handle to a device information set that contains a device interface for which to retrieve a list of device interface property keys.
  - Set *DeviceInterfaceData* to a pointer to an [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure that represents the device interface for which to retrieve a list of device property keys.
  - Set *PropertyKey* to a pointer to a [DEVPROPKEY](#) structure that represents the property.
  - Set *.PropertyType* to a pointer to a [DEVPROPKEY](#)-typed variable.
  - Set *PropertyBuffer* to **NULL**.
  - Set *PropertyBufferSize* to zero.
  - Set *RequiredSize* to a DWORD-typed variable.
  - Set *Flags* to zero.

In response to the first call to [SetupDiGetDeviceInterfaceProperty](#),

[SetupDiGetDeviceInterfaceProperty](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDeviceInterfaceProperty](#) again and supply the same parameter values that were supplied to the first call, except for the following changes:
  - Set *PropertyBuffer* to a pointer to the buffer that receives the property value.
  - Set *PropertyBufferSize* to the required size, in bytes, of the *PropertyBuffer* buffer. The first call to [SetupDiGetDeviceInterfaceProperty](#) retrieved the required size of the *PropertyBuffer* buffer in *\*RequiredSize*.

If the second call to [SetupDiGetDeviceInterfaceProperty](#) succeeds, [SetupDiGetDeviceInterfaceProperty](#) sets *\*PropertyType* to the property-data-type identifier for the property, sets the *PropertyBuffer* buffer to the property value, sets *\*RequiredSize* to the size, in bytes, of the property value that was retrieved, and returns **TRUE**. If the function call fails, [SetupDiGetDeviceInterfaceProperty](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Setting a Device Interface Property Value

11/2/2020 • 2 minutes to read • [Edit Online](#)

To set the value of a [device interface property](#) in Windows Vista and later versions of Windows, call [SetupDiSetDeviceInterfaceProperty](#) and supply the following parameters values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device interface for which to set a device interface property.
- Set *DeviceInterfaceData* to a pointer to an [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure that represents the device interface for which to set a device interface property.
- Set *PropertyKey* to a pointer to the [DEVPROPKEY](#) structure that represents the property to set.
- Set *PropertyType* to a pointer to a [DEVPROPKEY](#)-typed variable that supplies the property-data-type identifier for the property to set.
- Set *PropertyBuffer* to a pointer to the buffer that contains the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the property value.
- Set *RequiredSize* to a DWORD-typed variable.
- Set *Flags* to zero.

If the call to [SetupDiSetDeviceInterfaceProperty](#) succeeds, [SetupDiSetDeviceInterfaceProperty](#) sets the device class property and returns **TRUE**. If the function call fails, [SetupDiSetDeviceInterfaceProperty](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Using the INF AddProperty Directive and the INF DelProperty Directive

11/2/2020 • 5 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, you can use [INF AddProperty directives](#) and [INF DelProperty directives](#) to set and delete properties for device instances, [device setup classes](#), [device interface classes](#), and device interfaces. This includes [system-defined device properties](#) and [custom device properties](#). However, you should use the following guidelines when you use **AddProperty** and **DelProperty** directives instead of [INF AddReg directives](#) and [INF DelReg directives](#) to set and delete device properties:

- For device properties that were introduced on Windows Vista and later versions of Windows, you should use **AddProperty** and **DelProperty** directives to set and delete device properties.
- For device properties that were introduced on Windows Server 2003, Windows XP, or Windows 2000, and that can be set by the **AddReg** directive and deleted by the **DelReg** directive, you should continue to use **AddReg** and **DelReg** directives to set and delete these device properties. You should not use **AddProperty** and **DelProperty** directives.

You can include the INF **AddProperty** directive and the INF **DelProperty** directive in the following INF file sections to set and delete properties for device instances, device setup classes, device interface classes, and device interfaces:

- [INF DDInstall Section](#)
- [INF ClassInstall32 Section](#)
- An *install-interface-section* that is referenced by an [INF InterfaceInstall32 section](#)
- An *add-interface-section* that is referenced by an [INF AddInterface Directive](#)

## Using the INF AddProperty Directive

To modify a property value, include an INF **AddProperty** directive in the section that installs a device instance, a device setup class, a device interface class, or a device interface. An **AddProperty** directive references one or more *add-property-sections* that include entries that specify the property, how to modify the property, and the value that is used to modify the property. The format of an **AddProperty** directive is as follows:

**AddProperty**=*add-property-section*[,*add-property-section*] ...

Each line in an add-property-section specifies one property. The following shows the two possible line formats that specify property information. The first line format shown specifies a property by its name. This format can be used only with the **DEVPKEY\_DrvPkg\_Xxx** properties. The second line format specifies a property by the property category and property identifier of the corresponding [property key](#). This second format can be used to specify a system-defined property or a [custom device property](#).

[*add-property-section*] *property-name*,,[*flags*],*value*{*property-category-guid*},*property-pid*,*type*,[*flags*],*value*  
The entry values supply the following:

*property-name*

The name that identifies a **DEVPKEY\_DrvPkg\_Xxx** property. For example, **DeviceModel**, which represents the **DEVPKEY\_DrvPkg\_Model** property, or **DeviceVendorWebSite**, which represents the **DEVPKEY\_DrvPkg\_VendorWebSite** property.

*property-category-guid*

The GUID value of the property category to which the property belongs. For example, the system-defined [DEVPKEY\\_Device\\_FriendlyName](#) property. The GUID value can also specify a custom device category.

*property-pid*

The property identifier that identifies a property within a property category. For example, the value of the property identifier for the DEVPKEY\_Device\_FriendlyName property is 14.

*Flags*

An optional flag that indicates how to modify the property value.

*Type*

A [property-data-type identifier](#) that specifies the data type.

*value*

The value that is used to modify the property value.

The following example of an [AddProperty](#) directive includes two line entries. The first line includes the *property-name* entry value "DeviceModel" and the *value* entry value "Sample Device Model Name." This entry sets the DEVPKEY\_DrvPkg\_Model property. The second line entry sets a custom property in a custom property category. The *property-category-guid* entry value is "c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e" and the *property-identifier* entry value is "2." The optional *Flags* entry value is not present and the *type* entry value is "18" (DEVPROP\_TYPE\_STRING). The *value* entry value is "String value for property 1."

```
[Root_Install.NT]
AddProperty=Root_AddProperty

[Root_AddProperty]
DeviceModel,,,,"Sample Device Model Name"
{c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e}, 2, 18,, "String value for property 1"
```

## Using the INF DelProperty Directive

To delete a property, include an INF [DelProperty](#) directive in the section that installs a device instance, a device setup class, a device interface class, or a device interface.

The main purpose of the [INF DelProperty directive](#) is for use in an INF file that updates a device installation. In such a case, the [DelProperty](#) directive can be used to delete a property that was set by a previous installation, but is no longer required by the updated installation. Use the [DelProperty](#) directive with caution. [DelProperty](#) should not be used to delete a property that might be also set by a system component or by another INF file.

The [DelProperty](#) directive has the following format:

```
DelProperty=del-property-section[,del-property-section] ...
```

Each line in a *del-property-section* specifies one property. The following shows the two possible line formats that specify property information. The first line format shown specifies a property by its name. This format can be used only with the DEVPKEY\_DrvPkg\_Xxx properties. The second line format specifies a property by the property category and property identifier of the corresponding [property key](#). The second format can be used to specify a system-defined property or a [custom device property](#).

```
[del-property-section] property-name [, Flags [, value]] {property-category-guid}, property-pid [, Flags [, value]]
```

The entry values supply the following:

*property-name*

The name that identifies a DEVPKEY\_DrvPkg\_Xxx property. For example, [DeviceModel](#), which represents the [DEVPKEY\\_DrvPkg\\_Model](#) property, or [DeviceVendorWebSite](#), which represents the [DEVPKEY\\_Device\\_FriendlyName](#) property.

*property-category-guid*

The GUID value of the property category to which the property belongs. For example, the system-defined [DEVPKEY\\_Device\\_FriendlyName](#) property. The GUID value can also specify a custom device category.

*property-pid*

The property identifier that identifies a property within a property category. For example, the value of the property identifier for the DEVPKEY\_Device\_FriendlyName property is 14.

*Flags*

An optional flag that is valid for use only with a property whose data type is [DEVPROP\\_TYPE\\_STRING\\_LIST](#). If the flag is set, the delete operation deletes the string that is specified by *value* from the property string list.

*value*

The string to delete from a property string list.

The following example of a *del-property-section* includes two line entries.

The first line includes the *property-name* entry value "DeviceModel", which deletes the DEVPKEY\_DrvPkg\_Model property. The second line entry deletes the string "DeleteThisString" from a custom device property value whose data type is DEVPROP\_TYPE\_STRING\_LIST. In the second line, the *property-category-guid* entry value is "c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e", the *property-identifier* entry value is "2", and the *Flags* entry value is "0x00000001."

```
[SampleDelPropertySection]
DeviceModel
{c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e}, 2, 0x00000001, "DeleteThisString"
```

# Device Property Representations (Windows Server 2003, Windows XP, and Windows 2000)

11/2/2020 • 2 minutes to read • [Edit Online](#)

Windows Server 2003, Windows XP, and Windows 2000 do not support the [unified device property model](#) that Windows Vista and later versions of Windows support. However, most of the [system-defined device properties](#) that are included in the unified device property model have corresponding representations that are supported by these earlier versions of Windows. On these earlier versions of Windows, the way a device property is represented, and the mechanism to access a property, depends on the component type and property type. These representations and mechanisms include the following:

- A device property is represented by a system-defined identifier that is supplied as an input parameter to a [SetupAPI function](#) to access the device property.
- A device property does not have an explicit representation. However, the information that is associated with a device property can be retrieved by calls to SetupAPI functions or Plug and Play (PnP) configuration manager functions.
- A device property is represented by a registry entry value that can be accessed by using the Windows registry functions.
- INF file entry values modify device properties.

The following topics provide information about how to access device properties on Windows Server 2003, Windows XP, and Windows 2000:

[INF File Entry Values that Modify Device Properties](#)

[Using SetupAPI and Configuration Manager to Access Device Properties](#)

**Note** To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these mechanisms. However, you should use the unified device property model to access device properties on Windows Vista and later versions.

# INF File Entry Values That Modify Device Properties before Windows Vista

11/2/2020 • 3 minutes to read • [Edit Online](#)

The following are the INF file entry values that modify device properties on Windows Server 2003, Windows XP, and Windows 2000:

- INF file entry values that set device properties that correspond to the [system-defined device properties](#) that are part of the [unified device property model](#) in Windows Vista and later versions of Windows.
- [INF AddReg directives](#) and [INF DelReg directives](#) that set or delete system-defined registry entry values that correspond to the system-defined device properties that are part of the unified device property model in Windows Vista and later versions.
- INF [AddReg](#) directives and INF [DelReg](#) directives that set or delete custom registry entry values that correspond to custom device properties.

For general information about the INF file sections that install device instances, [device setup classes](#), [device interface classes](#), and device interfaces, see the following topics:

[INF DDInstall Section](#)

[INF ClassInstall32 Section](#)

[INF InterfaceInstall32 Section](#)

[INF DDInstall.Interfaces Section](#)

## INF File Entry Values That Correspond to System-Defined Device Properties

Some INF file entry values provide information that Windows uses to set the system-defined registry entry values that correspond to device instance properties and device interface properties. The following are a few examples of registry entry values that are supplied by such INF file entry values:

- The [INF Models section](#) of an INF file includes a *device-description* entry value. This value corresponds to the [DEVPKEY\\_Device\\_DeviceDesc](#) property in the unified device property model and can be retrieved by calling [SetupDiGetDeviceRegistryProperty](#) and setting the *Property* parameter to SPDRP\_DEVICEDESC.
- The INF Class directive of an [INF Version section](#) includes a *class-name* entry value that supplies the name of a [device setup class](#). This value corresponds to the [DEVPKEY\\_DeviceClass\\_ClassName](#) property in the unified device property model. The class name for a device setup class can be retrieved by calling [SetupDiClassNameFromGuid](#), and the class name of a device instance can be retrieved by calling [SetupDiGetDeviceRegistryProperty](#) and setting the *Property* parameter to SPDRP\_CLASS.
- The [INF InterfaceInstall32 section](#) includes an *InterfaceClassGuid* entry value that supplies the GUID of a device interface. This value corresponds to the [DEVPKEY\\_DeviceInterface\\_ClassGuid](#) property in the unified device property model. The GUIDs of installed device interface classes can be retrieved by querying the subkeys of the root of the interface key, which can be opened by calling [SetupDiOpenClassRegKeyEx](#) and setting the *ClassGuid* parameter to **NULL** and the *Flags* parameter to **DIOCR\_INTERFACE**. The GUID of a device interface can be retrieved by calling [SetupDiEnumDeviceInterfaces](#), which retrieves an [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure for the device interfaces that are associated with a device instance. The *InterfaceClassGuid* member of the [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure identifies the interface class GUID.

## **INF AddReg Directives and INF DelReg Directives That Modify System-Defined Device Properties**

Many system-defined device properties have corresponding system-defined registry entry values. For device properties that have corresponding registry entry values, using an [INF AddReg directive](#) to add the corresponding registry entry value sets the corresponding device property. Similarly, using an [INF DelReg directive](#) to delete the corresponding registry entry value, also deletes the corresponding device property.

For example, the INF AddReg directive in the following "Abc\_Device\_Install.HW" section would set the **DeviceCharacteristics** registry entry value for a device instance:

```
[Abc_Device_Install.HW]
...
AddReg = Xxx_AddReg
...
[Xxx_AddReg]
...
[HKR,,DeviceCharacteristics,0x10001,0x00000001
]
```

The **DeviceCharacteristics** registry entry value corresponds to the [DEVPKEY\\_Device\\_Characteristics](#) property in the [unified device property model](#) in Windows Vista and later versions of Windows.

## **INF AddReg Directives and INF DelReg Directives That Modify Custom Registry Entry Values**

Windows manages the correspondence between system-defined registry entry values and system-defined device properties. However, Windows does not manage the correspondence between custom registry entry values and custom device properties. An [INF AddReg directive](#) or an [INF DelReg directive](#) that modifies a custom registry entry value does not affect the system-defined properties that Windows manages.

Custom device instance properties that are set as shown in the following example, can be retrieved by calling [SetupDiGetCustomDeviceProperty](#).

```
[XxxDDInstall.HW]
...
AddReg = Xxx_AddReg
...
[Xxx_AddReg]
...
[HKR,,CustomPropertyName,0x10001,0x00000001
]
```

For more information about how to access custom device properties that have corresponding custom registry entry values, see [Accessing Custom Device Properties](#).

# Using SetupAPI and Configuration Manager to Access Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

Windows Server 2003, Windows XP, and Windows 2000 support device property representations that correspond to most of the system-defined device properties that are supported by the [unified device property model](#) of Windows Vista and later versions of Windows.

On these earlier versions of Windows, [SetupAPI functions](#), [configuration manager functions](#), and Windows registry functions can be used to access device properties, as described in the following topics:

[Accessing Device Instance SPDRP\\_Xxx Properties](#)

[Retrieving a Device Instance Identifier](#)

[Accessing Device Instance Driver Properties](#)

[Retrieving the Status and Problem Code for a Device Instance](#)

[Retrieving Device Relations](#)

[Accessing Device Setup Class Properties](#)

[Accessing Device Interface Class Properties](#)

[Accessing Device Interface Properties](#)

[Accessing Custom Device Properties](#)

# Accessing Device Instance SPDRP\_Xxx Properties

11/2/2020 • 3 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports the device instance properties that correspond to the SPDRP\_Xxx identifiers that are defined in *Setupapi.h*. These properties characterize the configuration of a device instance. The unified device property model uses [property keys](#) to represent these properties. Windows Server 2003, Windows XP, and Windows 2000 also support these device driver properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these earlier Windows versions use the SPDRP\_Xxx identifiers to represent and access the device instance properties.

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support using SPDRP\_Xxx identifiers to access device instance properties. However, you should use the corresponding property keys to access these properties on Windows Vista and later versions of Windows.

For a list of the system-defined device instance properties that have corresponding SPDRP\_Xxx identifiers, see [Device Properties That Have Corresponding SPDRP\\_Xxx Identifiers](#). The device instance properties are listed by the property keys that you use to access the properties in Windows Vista and later versions of Windows. The information that is provided with each property key includes the corresponding SPDRP\_Xxx identifiers that you can use to access the property on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device instance properties in Windows Vista and later versions of Windows, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

## Accessing a Device Property

To access device instance properties that correspond to the SPDRP\_Xxx identifiers on Windows Server 2003, Windows XP, and Windows 2000, use the following SetupAPI functions:

- [SetupDiGetDeviceRegistryProperty](#)

This function retrieves a device property that is specified by a SPDRP\_Xxx identifier.

- [SetupDiSetDeviceRegistryProperty](#)

This function sets the device property that is specified by a SPDRP\_Xxx identifier.

## Retrieving a Device Property

To retrieve a device property on Windows Server 2003, Windows XP, and Microsoft Windows 2000, follow these steps:

1. Call [SetupDiGetDeviceRegistryProperty](#) to retrieve the size, in bytes, of the property value. Supply the following parameter values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device instance for which to retrieve the requested property value.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device instance for which to retrieve the requested property value.
- Set *Property* to an SPDRP\_Xxx identifier. For a list of these identifiers and a description of the corresponding device properties, see the description of the *Property* parameter that is included with [SetupDiSetDeviceRegistryProperty](#).
- Set *PropertyRegDataType* to a pointer to a DWORD-typed variable.
- Set *PropertyBuffer* to NULL.

- Set *PropertyBufferSize* to zero.
- Set *RequiredSize* to a pointer to a DWORD-typed variable that receives, the size, in bytes of the property value.

In response to the call to [SetupDiSetDeviceRegistryProperty](#),

[SetupDiGetDeviceRegistryProperty](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetDeviceRegistryProperty](#) again and supply the same parameter values that were supplied in the first call, except for the following changes:

- Set *PropertyBuffer* to a pointer to a BYTE-typed buffer that receives the requested property value.
- Set *PropertyBufferSize* to the size, in bytes, of the *PropertyBuffer* buffer. The first call to [SetupDiGetDeviceRegistryProperty](#) retrieved the required size of the *PropertyBuffer* buffer in *\*RequiredSize*.

If the second call to [SetupDiGetDeviceRegistryProperty](#) succeeds,

[SetupDiGetDeviceRegistryProperty](#) sets *\*PropertyRegDataType* to the registry data type of the property value, sets the *PropertyBuffer* buffer to the property value, sets *\*RequiredSize* to the size, in bytes, of the property value that was retrieved, and returns **TRUE**. If the function call fails,

[SetupDiGetDeviceRegistryProperty](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

## Setting a Device Property

To set a device property on Windows Server 2003, Windows XP, and Windows 2000, call

[SetupDiSetDeviceRegistryProperty](#) and supply the following parameter values:

- Set *DeviceInfoSet* to a handle to a device information set that contains the device instance for which to set property value.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device instance for which to set property value.
- Set *Property* to an SPDRP\_Xxx identifier.
- Set *PropertyBuffer* to a pointer to a BYTE-typed buffer that contains the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the property value that is supplied in the *PropertyBuffer* buffer.

If this call to [SetupDiSetDeviceRegistryProperty](#) succeeds, [SetupDiSetDeviceRegistryProperty](#) sets the device instance property and returns **TRUE**. If the function call fails, [SetupDiSetDeviceRegistryProperty](#) returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Retrieving a Device Instance Identifier

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports a device property that represents the device instance identifier. The unified device property model uses the [DEVPKEY\\_Device\\_InstanceId](#) property key to represent this property.

Windows Server 2003, Windows XP, and Windows 2000 also support this property. However, these earlier Windows versions do not support the property key of the unified device property model. Instead, you can retrieve a device instance identifier on these earlier versions of Windows by calling [SetupDiGetDeviceInstanceId](#). To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support [SetupDiGetDeviceInstanceId](#). However, you should use the corresponding property key to access this property on Windows Vista and later.

For information about how to use property keys to access device driver properties on Windows Vista and later versions, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

To retrieve a device instance identifier on Windows Server 2003, Windows XP, and Windows 2000, follow these steps:

1. Call [SetupDiGetDeviceInstanceId](#) to retrieve the size, in bytes, of the device instance identifier. Supply the following parameter values:

- Set *DeviceInfoSet* to a handle to the device information set that contains the device information element for which to retrieve the requested device instance identifier.
- Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device information element for which to retrieve a device instance identifier.
- Set *DeviceInstanceId* to **NULL**.
- Set *DeviceInstanceIdSize* to zero.
- Set *RequiredSize* to a pointer to a DWORD-typed variable that receives the number of characters required to store the NULL-terminated device instance identifier.

In response to the first call to [SetupDiGetDeviceInstanceId](#), [SetupDiGetDeviceInstanceId](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code [ERROR\\_INSUFFICIENT\\_BUFFER](#), and returns **FALSE**. A subsequent call to [GetLastError](#) returns the most recently logged error code.

2. Call [SetupDiGetDeviceInstanceId](#) again and supply the same parameter values that were supplied in the first call, except for the following changes:

- Set *DeviceInstanceId* to a pointer to a string buffer that receives the NULL-terminated device instance identifier that is associated with the device information element.
- Set *DeviceInstanceIdSize* to the size, in characters, of the *DeviceInstanceId* buffer. The first call to [SetupDiGetDeviceInstanceId](#) retrieved the required size of the *DeviceInstanceId* buffer in *\*RequiredSize*.

If the second call to [SetupDiGetDeviceInstanceId](#) succeeds, [SetupDiGetDeviceInstanceId](#) sets the *DeviceInstanceId* buffer to the device instance identifier, sets *\*RequiredSize* to the size, in characters, of the device instance identifier that was retrieved, and returns **TRUE**. If the function call fails, [SetupDiGetDeviceInstanceId](#) returns **FALSE** and a call to [GetLastError](#) returns the logged error code.

# Accessing Device Driver Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes device driver properties that characterize a device driver. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support most of these device driver properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows use the following mechanisms to represent and access the corresponding property information:

- [Accessing Device Driver Properties That Have Corresponding Registry Entry Values](#)
- [Using SetupDiGetDriverInstallParams to Retrieve Driver Rank](#)

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these two ways to access information about a device interface. However, you should use the property keys to access these properties on Windows Vista and later versions.

For a list of the system-defined device driver properties, see [Device Driver Properties](#). The device driver properties are listed by the property key identifiers that you use to access the properties on Windows Vista and later versions. The information that is provided with the property keys includes the names of the corresponding system-defined registry entry values that you can use to access the property on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device driver properties on Windows Vista and later versions, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

## Accessing Device Driver Properties That Have Corresponding Registry Entry Values

To access device driver properties on Windows Server 2003, Windows XP, and Windows 2000, follow these steps:

1. Call [SetupDiOpenDevRegKey](#) to retrieve a handle to the software key for a device instance. Supply the following parameter values:
  - Set *DeviceInfoSet* to a handle to the device information set that contains the device information element for which to retrieve the global software key.
  - Set *DeviceInfoData* to a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device information element for which to retrieve the global software key.
  - Set *Scope* to DICS\_FLAG\_GLOBAL.
  - Set *HwProfile* to zero.
  - Set *KeyType* to DIREG\_DRV, which configures [SetupDiOpenDevRegKey](#) to retrieve a handle to the software key for a device instance.
  - Set *samDesired* to a REGSAM-typed value that specifies the access that you require for this key. For all access, set *samDesired* to KEY\_ALL\_ACCESS.

If the call to [SetupDiOpenDevRegKey](#) succeeds, [SetupDiOpenDevRegKey](#) returns a handle to the requested software key. If the function call fails, [SetupDiOpenDevRegKey](#) returns INVALID\_HANDLE\_VALUE. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Supply the handle in a call to [RegQueryValueEx](#) or to [RegSetValueEx](#) to retrieve or set the registry entry value that corresponds to the device instance driver property.

3. Call the [RegCloseKey](#) function to close the software registry key after access to the key is no longer required.

#### **Using SetupDiGetDriverInstallParams to Retrieve Driver Rank**

On Windows Server 2003, Windows XP, and Windows 2000, you can retrieve the rank of a driver that is currently installed for a device by calling [SetupDiGetDriverInstallParams](#). `SetupDiGetDriverInstallParams` retrieves a pointer to an [SP\\_DRVINSTALL\\_PARAMS](#) structure for the driver in the output parameter `DriverInstallParams`. The `Rank` member of the retrieved `SP_DRVINSTALL_PARAMS` structure contains the driver rank.

# Retrieving the Status and Problem Code for a Device Instance

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes a device status property and a problem code property. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 do not support the property keys of the unified property model, nor do they support corresponding registry entry values that represent these properties. However, the corresponding information can be retrieved by calling the [CM\\_Get\\_DevNode\\_Status](#) function. To maintain compatibility with earlier versions of Windows, Windows Vista and later versions also support [CM\\_Get\\_DevNode\\_Status](#). However, you should use the property keys of the unified device property model to access the device driver properties.

The device driver properties are listed by the property key identifiers that you use to access the property in Windows Vista and later versions.

For information about how to use property keys to access device driver properties in Windows Vista and later versions, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

To access the status and problem code for a device instance on Windows Server 2003, Windows XP, and Windows 2000, call [CM\\_Get\\_DevNode\\_Status](#) and supply the following parameters:

- Set *pulStatus* to a pointer to a ULONG-typed value that receives the status bit flags that are set for a device instance. The status value can be any combination of the bit flags with prefix "DN\_" that are defined in *Cfg.h*.
- Set *pulProblemNumber* to a pointer to a ULONG-typed value that receives the problem number that is set for a device instance. The problem number is one of the constants with the prefix "CM\_PROB\_" that are defined in *Cfg.h*. [CM\\_Get\\_DevNode\\_Status](#) sets the problem number only if DN\_HAS\_PROBLEM is set in *pulStatus*.
- Set *dnDevInst* to a device instance handle to the device for which to retrieve the status and problem code.
- Set *ulFlags* to zero.

If the call to [CM\\_Get\\_DevNode\\_Status](#) succeeds, [CM\\_Get\\_DevNode\\_Status](#) retrieves the requested status and problem code for the device instance and returns CR\_SUCCESS. If the function call fails, [CM\\_Get\\_DevNode\\_Status](#) returns one of the error codes with prefix "CR\_" that are defined in *Cfgmgr32.h*.

## Using Device Manager to find problem code and problem status for a device

When there is a problem with a devnode, the problem code appears on the General tab, in the Device status field.

The **Problem status** property appears in the **Property** dropdown on the **Details** tab for the device in Device Manager.

## Using the debugger to find problem code and problem status for a

## device

To view all devices that have a problem code in the kernel debugger, use the [!devnode 0 21](#) extension. This also shows you the ProblemStatus on the device. For example:

```
0: kd> !devnode 0 21
Dumping IopRootDeviceNode (= 0x85d37e30)
DevNode 0x8ad6ab78 for PDO 0x81635c30
  InstancePath is "ROOT\DIINSTALLDRIVER\0003"
  ServiceName is "isolated"
  State = DeviceNodeRemoved (0x312)
  Previous State = DeviceNodeInitialized (0x302)
  Problem = CM_PROB_FAILED_ADD
  Problem Status = 0xc00000bb
```

You can also view problem code and problem status by issuing [[!devnode](#)] (./debugger/-devnode.md) on a DEVICE\_NODE address:

```
0: kd> !devnode 0x8ad6ab78
DevNode 0x8ad6ab78 for PDO 0x81635c30
  Parent 0x85d37e30  Sibling 0x8adee670  Child 0000000000
  ...
  Problem = CM_PROB_FAILED_ADD
  Problem Status = 0xc00000bb
```

You can also see this info on a running system in Device Manager. For info, see [DEVPKEY\\_Device\\_ProblemStatus](#).

## See Also

- [DEVPKEY\\_Device\\_ProblemCode](#)
- [DEVPKEY\\_Device\\_ProblemStatus](#)

# Retrieving Device Relations

11/2/2020 • 4 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes [device relations properties](#). The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 do not support the property keys of the unified property model, nor do they support the corresponding registry entry values that represent these properties. However, you can retrieve the corresponding information by calling Plug and Play (PnP) configuration manager functions. To maintain compatibility with earlier Windows versions, Windows Vista and later versions also support calling PnP configuration manager functions to retrieve device relations properties. However, you should use the property keys of the unified device property model to access the device relation properties.

For information about how to use property keys to access device driver properties, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

For information about how to access the device relations properties on Windows Server 2003, Windows XP, and Windows 2000, see the following topics:

[Retrieving Ejection Relations, Removal Relations, and Power Relations, and Bus Relations](#)

[Retrieving the Parent of a Device Instance](#)

[Retrieving the Children of a Device Instance](#)

[Retrieving the Siblings of a Device Instance](#)

## **Retrieving Ejection Relations, Removal Relations, and Power Relations, and Bus Relations**

To retrieve device relations information on Windows Server 2003, Windows XP, and Windows 2000, call [CM\\_Get\\_Device\\_ID\\_List](#) and supply the following parameter values:

- Set *pszFilter* to a pointer to a NULL-terminated string that specifies the device instance identifier for which to retrieve relations information.
- Set *Buffer* to a pointer to a buffer that receives a list of NULL-terminated device instance identifiers. The list is terminated by an additional NULL character. You can get the required buffer size by calling the [CM\\_Get\\_Device\\_ID\\_List\\_Size](#) function.
- Set *BufferLen* to the size, in characters, of the *Buffer* buffer.
- Set *uFlags* to one of the following flags to retrieve the corresponding relations information:
  - [CM\\_GETIDLIST\\_FILTER\\_EJECTIONRELATIONS](#)

The [CM\\_GETIDLIST\\_FILTER\\_EJECTIONRELATIONS](#) flag retrieves [ejection relations](#), which is the same information that is provided by the [DEVPKEY\\_Device\\_EjectionRelations](#) device property in Windows Vista and later versions.

- [CM\\_GETIDLIST\\_FILTER\\_REMOVALRELATIONS](#)

The [CM\\_GETIDLIST\\_FILTER\\_REMOVALRELATIONS](#) flag retrieves [removal relations](#), which is the same information that is provided by the [DEVPKEY\\_Device\\_RemovalRelations](#) device property in Windows Vista and later versions.

- [CM\\_GETIDLIST\\_FILTER\\_POWERRELATIONS](#)

The CM\_GETIDLIST\_FILTER\_POWERRELATIONS flag retrieves power relations, which is the same information that is provided by the **DEVPKEY\_Device\_PowerRelations** device property in Windows Vista and later versions.

- CM\_GETIDLIST\_FILTER\_BUSRELATIONS

The CM\_GETIDLIST\_FILTER\_BUSRELATIONS flag retrieves **bus relations**, which is the same information that is provided by the **DEVPKEY\_Device\_BusRelations** device property in Windows Vista and later versions.

If the call to **CM\_Get\_Device\_ID\_List** succeeds, **CM\_Get\_Device\_ID\_List** retrieves the requested relations information and returns CR\_SUCCESS. Otherwise, **CM\_Get\_Device\_ID\_List** returns one of the error codes with prefix "CR\_" that are defined in *Cfgmgr32.h*.

### **Retrieving the Parent of a Device Instance**

To retrieve the device instance identifier of a parent device on Windows Server 2003, Windows XP, and Windows 2000, follow these steps:

1. Call the **CM\_Get\_Parent** function to retrieve a device instance handle to the parent device of a device instance.
2. Call **CM\_Get\_Device\_ID** to retrieve the device instance identifier that is associated with the device instance handle to the parent device that was retrieved by the previous call to **CM\_Get\_Parent**.

This information retrieved by using this procedure is the same as that represented by the **DEVPKEY\_Device\_Parent** property in the unified device property model of Windows Vista and later versions.

### **Retrieving the Children of a Device Instance**

To retrieve the device instance identifiers of the child devices of a device instance on Windows Server 2003, Windows XP, and Windows 2000, follow these steps:

1. Call the **CM\_Get\_Child** function to retrieve a device instance handle to the first child device that is associated with a device instance.
2. Call **CM\_Get\_Sibling** as many times as it is necessary to enumerate all the sibling devices of the first child device that was retrieved by the call to **CM\_Get\_Child**.
3. Call **CM\_Get\_Device\_ID** to retrieve the device instance identifiers that are associated with the device instance handles that were returned by the calls to **CM\_Get\_Child** and **CM\_Get\_Sibling**.

This information retrieved by using this procedure is the same as that represented by the **DEVPKEY\_Device\_Children** property in the unified device property model of Windows Vista and later versions.

### **Retrieving the Siblings of a Device Instance**

To retrieve the device instance identifiers of the sibling devices of device instance *Abc* on Windows Server 2003, Windows XP, and Windows 2000, follow these steps:

1. Call the **CM\_Get\_Parent** function to retrieve a device instance handle to the parent device of device instance *Abc*.
2. Call the **CM\_Get\_Child** function to retrieve a device instance handle to the first child device of the parent device of device instance *Abc*.
3. Call **CM\_Get\_Sibling** as many times as is necessary to enumerate all the sibling devices of the first child device of the parent device. This enumeration will also return a handle to device instance *Abc*.
4. Call **CM\_Get\_Device\_ID** to retrieve the device instance identifiers that are associated with the device instance handles that were returned by the previous calls to **CM\_Get\_Sibling**. Remove the handle to device instance *Abc* from the list of sibling devices of the first child device of the parent device.

The information retrieved by using this procedure is the same as that represented by the DEVPKEY\_Device\_Siblings property in the unified device property model of Windows Vista and later versions. If a CM\_Xxx function call listed in this section succeeds, the CM\_Xxx function retrieves the requested information and returns CR\_SUCCESS. Otherwise, the CM\_Xxx function returns one of the error codes with prefix "CR\_" that are defined in *Cfgmgr32.h*.

# Accessing Device Setup Class Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes the following:

- [Device setup class properties](#) that correspond to the SPCRP\_Xxx identifiers that are defined in *Setupapi.h*.
- Properties that do not have SPCRP\_Xxx identifiers, but do have corresponding system-defined registry entry values.

For information about how to access device setup class properties in Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

For information about how to access the corresponding device setup class properties on Windows Server 2003, Windows XP, and Windows 2000, see the following topics:

[Retrieving SPCRP\\_Xxx Properties](#)

[Setting SPCRP\\_Xxx Properties](#)

[Accessing the Friendly Name and Class Name of a Device Setup Class](#)

[Accessing Icon Properties of a Device Setup Class](#)

[Accessing Registry Entry Values Under the Class Registry Key](#)

[Accessing the Co-installers Registry Entry Value of a Device Setup Class](#)

# Retrieving SPCRP\_Xxx Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports the device instance properties that correspond to the SPCRP\_Xxx identifiers that are defined in *Setupapi.h*. These properties characterize a [device setup class](#). The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support most of these device setup class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows use the SPCRP\_Xxx identifiers to represent and access the device setup class properties. To maintain compatibility with earlier versions of Windows, Windows Vista and later versions also support using SPCRP\_Xxx identifiers to access device setup class properties. However, you should use the property keys of the unified device property model to access device setup class properties.

For a list of the system-defined device setup class properties, see [Device Setup Class Properties That Correspond to SPCRP\\_Xxx Identifiers](#). The device setup class properties are listed by the property key identifiers that you use to access the properties in Windows Vista and later versions. The information that is provided with the property keys also includes the corresponding SPCRP\_Xxx identifiers that you can use to access the properties on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device setup class properties in Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

To retrieve device setup class properties on Windows Server 2003, Windows XP, and Windows 2000, use the [SetupDiGetClassRegistryProperty](#) function.

To use [SetupDiGetClassRegistryProperty](#) to retrieve a property that corresponds to an SPCRP\_Xxx identifier, follow these steps:

1. Call [SetupDiGetClassRegistryProperty](#) and supply the following parameter values:

- Set *ClassGuid* to a pointer to a GUID that represents the device setup class for which to retrieve a property.
- Set *Property* to the property identifier with prefix "SPCRP\_" for which to retrieve the size of the property value.
- Set *PropertyRegDataType* to **NULL**.
- Set *PropertyBuffer* to **NULL**.
- Set *PropertyBufferSize* to zero.
- Set *RequiredSize* to a pointer to a DWORD-typed variable that receives the size, in bytes, of the requested property.
- Set *MachineName* to the name of the computer.
- Set *Reserved* to **NULL**.

In response to the call to [SetupDiGetClassRegistryProperty](#), [SetupDiGetClassRegistryProperty](#) sets *\*RequiredSize* to the size, in bytes, of the buffer that is required to retrieve the property value, logs the error code **ERROR\_INSUFFICIENT\_BUFFER**, and returns **FALSE**. A subsequent call to [GetLastError](#) will return the most recently logged error code.

2. Call [SetupDiGetClassRegistryProperty](#) again to retrieve the property value and supply the same parameters that were supplied in the first call, except for the following changes:

- Set *PropertyBuffer* to a pointer to the buffer that receives the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the *PropertyBuffer* buffer. The first call to **SetupDiGetClassRegistryProperty** retrieved the required size of the *PropertyBuffer* buffer in *\*RequiredSize*.

If the second call to **SetupDiGetClassRegistryProperty** succeeds, **SetupDiGetClassRegistryProperty** sets *\*PropertyRegDataType* to the registry data type, sets the *PropertyBuffer* buffer to the property value, sets *\*PropertyBufferSize* to the size, in bytes, of the property value, and returns **TRUE**. If the function call fails, **SetupDiGetClassRegistryProperty** returns **FALSE** and a call to [GetLastError](#) will return the logged error code.

# Setting SPCRP\_Xxx Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports the [device setup class properties](#) that correspond to the SPCRP\_Xxx identifiers that are defined in *Setupapi.h*. These properties characterize a device setup class. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support most of these device setup class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows use the SPCRP\_Xxx identifiers to represent the device setup class properties. To maintain compatibility with earlier versions of Windows, Windows Vista and later versions also support using SPCRP\_Xxx identifiers to access device setup class properties. However, you should use the property keys of the unified device property model to access device setup class properties.

For a list of the system-defined device setup class properties, see [Device Setup Class Properties That Correspond to SPCRP\\_Xxx Identifiers](#). The device setup class properties are listed by the property key identifiers that you use to access the properties in Windows Vista and later versions. The information that is provided with the property keys also includes the corresponding SPCRP\_Xxx identifiers that you can use to access the properties on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device setup class properties in Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

To set device setup class properties on Windows Server 2003, Windows XP, and Windows 2000, call [SetupDiSetClassRegistryProperty](#) and supply the following parameter values:

- Set *ClassGuid* to a pointer to a GUID that represents the device setup class for which to set a property.
- Set *Property* to the identifier of the property to set with prefix "SPCRP\_".
- Set *PropertyBuffer* to a pointer to the buffer that contains the property value.
- Set *PropertyBufferSize* to the size, in bytes, of the property data.
- Set *MachineName* to the computer name.
- Set *Reserved* to NULL.

If this call to [SetupDiSetClassRegistryProperty](#) succeeds, [SetupDiSetClassRegistryProperty](#) sets the device setup class property and returns TRUE. If the function call fails, [SetupDiSetClassRegistryProperty](#) will return FALSE and a call to [GetLastError](#) will return the most recently logged error code.

# Accessing the Friendly Name and Class Name of a Device Setup Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes [device setup class properties](#) that represent the friendly name and class name of a device setup class. The unified device property model uses the [DEVPKEY\\_DeviceClass\\_Name](#) property key and the [DEVPKEY\\_DeviceClass\\_ClassName](#) property key to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support these device setup class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows use the following mechanisms to retrieve the corresponding property information:

- Call [SetupDiGetClassDescriptionEx](#) to retrieve the friendly name of a device setup class.
- Call [SetupDiClassNameFromGuid](#) to retrieve the class name of a device setup class.

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these mechanisms to access the friendly name and class name of a device setup class. However, you should use the property keys to access these properties in Windows Vista and later versions.

# Accessing Icon Properties of a Device Setup Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes [device setup class properties](#) that represent icon properties of a device setup class. The unified device property model uses the [DEVPKEY\\_DeviceClass\\_Icon property key](#) and the [DEVPKEY\\_DeviceClass\\_IconPath](#) property key to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support these device setup class properties. However, these earlier versions of Windows do support the following mechanisms to retrieve information about device setup class icons:

- Call [SetupDiLoadClassIcon](#) to retrieve the index of the mini-icon for a device setup class in the *MinIconIndex* output parameter. You can then pass the retrieved mini-icon index to [SetupDiDrawMinIcon](#) to draw a mini-icon of the retrieved class icon in a specified device context.
- Call [SetupDiLoadClassIcon](#) to load the large icon for a device setup class in the context of the caller and return a handle to the large icon to the caller.

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these mechanisms to access the icons of a device setup class. However, you should use the property keys to access the icon properties in Windows Vista and later versions.

# Accessing Registry Entry Values Under the Class Registry Key

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes [device setup class properties](#) that do not have corresponding SPCRP\_Xxx identifiers. These properties characterize a device setup class. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support most of these device setup class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows represent these properties by using corresponding system-defined registry entry values under the class registry key. To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these system-defined registry entry values. However, you should use the property keys to access these properties on Windows Vista and later versions.

For a list of the system-defined device setup class properties, see [Device Setup Class Properties That Do Not Have Corresponding SPCRP\\_Xxx Identifiers](#). The device setup class properties are listed by the property key identifiers that you use to access the properties in Windows Vista and later versions. The information that is provided with the property keys also includes the corresponding registry entry values that you can use to access the properties on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device setup class properties in Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

To access these properties on Windows Server 2003, Windows XP, and Windows 2000, open the class registry key and use the Windows registry functions to access the registry entry values that correspond to these properties.

To retrieve a handle to the class registry key for a device setup class, call the [SetupDiOpenClassRegKeyEx](#) function and supply the following parameter values:

- Set *ClassGuid* to a pointer to the GUID that identifies the device setup class of the requested class registry key.
- Set *samDesired* to a REGSAM-typed value that specifies the required access permission.
- Set *Flags* to DIOCR\_INSTALLER.
- Set *MachineName* to a pointer to a NULL-terminated string that contains the name of the computer on which to open the requested class registry key. If the computer is the local computer, set *MachineName* to NULL.
- Set *Reserved* to NULL.

If this call to [SetupDiOpenClassRegKeyEx](#) succeeds, [SetupDiOpenClassRegKeyEx](#) returns the requested handle. If the function call fails, [SetupDiOpenClassRegKeyEx](#) returns INVALID\_HANDLE\_VALUE and a call to [GetLastError](#) will return the logged error code.

After you retrieve a handle to the class registry key, supply the handle in a call to [RegQueryValueEx](#) and [RegSetValueEx](#) to retrieve or set the registry entry value that corresponds to the device setup class property.

Call the [RegCloseKey](#) function to close the class registry key after access to the key is no longer required.

# Accessing the Co-installers Registry Entry Value of a Device Setup Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes a [device setup class property](#) that represents the co-installers that are installed for the class. The unified device property model uses the [DEVPKEY\\_DeviceClass\\_ClassCoInstallers property key](#) to represent this property.

Windows Server 2003, Windows XP, and Windows 2000 also support this property. However, these earlier versions of Windows do not support the property key of the unified device property model. Instead, these versions of Windows represent this property by using a corresponding system-defined registry entry value. To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support this system-defined registry entry value. However, you should use the property key to access these properties on Windows Vista and later versions.

For information about how to use property keys to access device setup class properties on Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

On Windows Server 2003, Windows XP, and Windows 2000, you can set or retrieve this property by using the Windows registry functions to access the following registry entry value for a device setup class:

`HLM\System\CurrentControlSet\Control\CoDeviceInstallers\{device-setup-class-guid}`.

For information about registering a class co-installer, see [Registering a Class Co-installer](#).

# Accessing Device Interface Class Properties

11/2/2020 • 3 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes device interface class properties that characterize a device interface class. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 also support most of these device interface class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, you can represent and access the corresponding property information on these versions of Windows by using the following methods:

- [Accessing the Default Interface for a Device Interface Class](#)
- [Accessing Device Interface Class Properties That Have Registry Entry Values Under the Interface Class Registry Key](#)

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these two ways to access information about a device interface. However, you should use the property keys to access these properties in Windows Vista and later versions.

For a list of the system-defined device interface class properties, see [Device Interface Class Properties](#). The [device setup class properties](#) are listed by the property key identifiers that you use to access the properties in Windows Vista and later versions. The information that is provided with the property keys also includes the corresponding registry entry values that you can use to access the properties on Windows Server 2003, Windows XP and Windows 2000.

For information about how to use property keys to access device setup class properties in Windows Vista and later versions, see [Accessing Device Class Properties \(Windows Vista and Later\)](#).

## Accessing the Default Interface for a Device Interface Class

To retrieve the default interface for a device interface class, call [SetupDiGetClassDevs](#) and supply the following parameter values:

- Set *ClassGuid* to the GUID that represents the device interface class for which to retrieve the default interface.
- Set *Enumerator* to **NULL**.
- Set *hwndParent* to **NULL**.
- Set *Flags* to **(DIGCF\_DEVICEINTERFACE | DIGCF\_DEFAULT)**.

This call will return a device information set that contains a device information element. The device information element that is returned represents the device that supports the default interface for the specified device interface class.

To set the default interface for a device interface class, call [SetupDiSetDeviceInterfaceDefault](#) and supply the following parameters values:

- Set *DeviceInfoSet* to a handle to the device information set that contains the device interface to set as the default for a device interface class.
- Set *DeviceInterfaceData* to a pointer to an [\*\*SP\\_DEVICE\\_INTERFACE\\_DATA\*\*](#) structure that specifies the device interface in *DeviceInfoSet*.

## Accessing Device Interface Class Properties That Have Registry Entry Values Under the Interface Class Registry Key

To access properties of a device interface class that have corresponding registry entry values under the interface class registry key, follow these steps:

1. Call the [SetupDiOpenClassRegKeyEx](#) function to open the interface class registry key and supply the following parameter values:

- Set *ClassGuid* to a pointer to the GUID that identifies the device interface class of the requested class registry key.
- Set *samDesired* to a REGSAM-typed value that specifies the required access permission.
- Set *Flags* to DIOCR\_INTERFACE.
- Set *MachineName* to a pointer to a NULL-terminated string that contains the name of the computer on which to open the requested class registry key. If the computer is the local computer, set *MachineName* to NULL.
- Set *Reserved* to NULL.

If this call to [SetupDiOpenClassRegKeyEx](#) succeeds, [SetupDiOpenClassRegKeyEx](#) returns the requested handle. If the function call fails, [SetupDiOpenClassRegKeyEx](#) returns INVALID\_HANDLE\_VALUE and a call to [GetLastError](#) will return the logged error code.

2. Supply the retrieved handle in a call to [RegQueryValueEx](#) and [RegSetValueEx](#) to retrieve or set the registry entry value that corresponds to the device interface class property.
3. Call the [RegCloseKey](#) function to close the class registry key after access to the key is no longer required.

For information about how to install and use device interfaces, see [Device Interface Classes](#) and [INF AddInterface Directive](#).

# Accessing Device Interface Properties before Windows Vista

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) includes device interface properties that characterize a device interface. The unified device property model uses [property keys](#) to represent these properties.

Windows Server 2003, Windows XP, and Windows 2000 support most of these device interface class properties. However, these earlier versions of Windows do not support the property keys of the unified device property model. Instead, these versions of Windows use the following mechanisms to represent and access device interface properties:

- [Accessing Device Interface Properties That Have Corresponding Registry Entry Values](#)
- [Using SetupDiEnumDeviceInterfaces to Retrieve Information About a Device Interface](#).

To maintain compatibility with these earlier versions of Windows, Windows Vista and later versions also support these two ways to access information about a device interface. However, you should use the property keys to access these properties in Windows Vista and later versions.

For a list of the system-defined device interface class properties, see [Device Interface Properties](#). The device interface class properties are listed by their corresponding property keys that you use to access the properties in Windows Vista and later versions. The information that is provided with the property keys includes the corresponding registry entry values, if any, that you can use to access the properties on Windows Server 2003, Windows XP, and Windows 2000.

For information about how to use property keys to access device setup class properties in Windows Vista and later versions, see [Accessing Device Interface Properties \(Windows Vista and Later\)](#).

For information about how to install and use device interfaces, see [Device Interface Classes](#) and [INF AddInterface Directive](#).

## Accessing Device Interface Properties That Have Corresponding Registry Entry Values

To access device interface properties by using registry entry values on Windows Server 2003, Windows XP, and Windows 2000, first call [SetupDiOpenDeviceInterfaceRegKey](#) and supply the following parameters:

- Set *DeviceInfoSet* to a pointer to a device information set that contains the device interface.
- Set *DeviceInterfaceData* to a pointer to an [SP\\_DEVICE\\_INTERFACE\\_DATA](#) structure that identifies the device interface.
- Set *Reserved* to zero.
- Set *samDesired* to a REGSAM-typed value that specifies the required access permissions.

If this call to [SetupDiOpenDeviceInterfaceRegKey](#) succeeds, [SetupDiOpenDeviceInterfaceRegKey](#) returns the requested handle. If the function call fails, [SetupDiOpenDeviceInterfaceRegKey](#) returns INVALID\_HANDLE\_VALUE and a call to [GetLastError](#) will return the logged error code.

After you retrieve a handle to the device interface registry key, supply the handle in a call to [RegQueryValueEx](#) or [RegSetValueEx](#) to retrieve or set the registry entry value that corresponds to the device interface property.

Call the [RegCloseKey](#) function to close the class registry key after access to the key is no longer required.

## Using `SetupDiEnumDeviceInterfaces` to Retrieve Information About a Device Interface

Another way to retrieve information about a device interface on Windows Server 2003, Windows XP, and Windows 2000 is by calling `SetupDiEnumDeviceInterfaces` to retrieve an `SP_DEVICE_INTERFACE_DATA` structure for the interface. An `SP_DEVICE_INTERFACE_DATA` structure contains the following information:

- The `Flags` member indicates whether a device interface is active or removed, and whether the device is the default interface for the interface class.
- The `InterfaceClassGuid` member identifies the interface class GUID.

# Accessing Custom Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the [unified device property model](#) supports using [property keys](#) to create and access custom device properties. For more information, see [Creating Custom Device Properties](#).

On Windows Server 2003, Windows XP, and Windows 2000, you can create custom registry entry values under the system-supplied registry keys for a device-related component. The following list contains the SetupAPI function to call for each type of device component to open the corresponding system-supplied registry key. After you open the system-defined registry key, applications and installers can call the Windows-based registry functions to modify custom registry entry values under the open registry key.

- A custom registry entry value for a device instance hardware property should be located under the hardware registry key of a device instance. Call [SetupDiOpenDevRegKey](#) and supply DIREG\_DEV in the *Flags* parameter to retrieve a handle to the hardware key of a device instance. Custom registry entry values that are set under the hardware registry key for a device instance can be retrieved by calling the [SetupDiGetCustomDeviceProperty](#) function.
- A custom registry entry value for a device instance software property should be located under the software registry key of a device instance. Call [SetupDiOpenDevRegKey](#) and supply DIREG\_DRV in the *Flags* parameter to retrieve a handle to the software key of a device instance.
- A custom registry entry value for a [device setup class property](#) should be located under the device setup class registry key. Call [SetupDiOpenClassRegKeyEx](#) and supply DIOCR\_INSTALLER in the *Flags* parameter to retrieve a handle to the registry key for a device setup class.
- A custom registry entry value for a device interface class property should be located under the device interface class registry key. Call [SetupDiOpenClassRegKeyEx](#) and supply DIOCR\_INTERFACE in the *Flags* parameter to retrieve a handle to the registry key for a device interface class.
- A custom registry entry value for a device interface property should be located under the device interface registry key. Call [SetupDiOpenDeviceInterfaceRegKey](#) to retrieve a handle to the registry key for a device interface class.

After you retrieve a handle to a registry key, supply the handle in a call to [RegQueryValueEx](#) or [RegSetValueEx](#) to retrieve or set the custom registry entry value that corresponds to the custom device property.

Call the [RegCloseKey](#) function to close the registry key after access to the registry key is no longer required.

# Device Installations and System Restarts

11/2/2020 • 2 minutes to read • [Edit Online](#)

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

Installing or removing a non-Plug and Play device.

For these earlier 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 starts.

**Note** The device's setup files should not initiate a system restart, 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 putting 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 restart the system.

**Note** The device's setup files should not initiate a system restart.

Installing a non-WDM filter driver.

If a filter driver is added to a non-WDM driver stack, the system must be restarted. In this case, the driver's installer or co-installer should request a system restart (see [Initiating System Restarts During Device Installations](#)).

**Note** Adding a filter driver to a WDM driver stack does not require a system restart, unless an underlying device is a system boot device.

The section includes the following topics:

[Avoiding System Restarts during Device Installations](#)

[Initiating System Restarts During Device Installations](#)

# Avoiding System Restarts during Device Installations and Driver Updates

11/2/2020 • 2 minutes to read • [Edit Online](#)

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

- Never use **Reboot** or **Restart** entries in **INF DDInstall sections**. These directives were originally provided for compatibility with Windows 9x/Me and should not be used for Windows 2000 and later versions of Windows.
- 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 need for a system restart 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 that are listed under [Updating Driver Files](#).

## Minimizing restarts when updating file-backed drivers

Prior to Windows 10, all kernel-mode drivers were backed by the system's paging file. As a result, a driver binary could be overwritten on disk even while the driver was running.

To improve performance, starting with Windows 10, most non-boot-start drivers are instead backed by the driver binary on disk.

Driver start types that are now file-backed include:

- **SERVICE\_SYSTEM\_START** (0x00000001)
- **SERVICE\_AUTO\_START** (0x00000002)
- **SERVICE\_DEMAND\_START** (0x00000003)

Boot start drivers continue to be backed by the paging file.

To update a file-backed driver, use the following best practices. Otherwise, the update might require two restarts, one to replace the file and a second to load the new version of the driver.

If you are using an INF file, follow these steps:

1. Modify your driver INF file's **CopyFiles** section to use **COPYFLG\_IN\_USE\_RENAME**, as follows:

```
[MyDriver_Install.NT]
CopyFiles=MyDriverCopy

[MyDriverCopy]
MyDriver.sys,,,0x00004000 ; COPYFLG_IN_USE_RENAME
```

If you use this flag, Windows attempts to replace the driver file on disk. For more info, see [INF CopyFiles Directive](#).

2. If the INF is for a PnP driver, during device installation Windows attempts to unload the running driver and restart the devices that use it, in order to pick up the new version of the driver. If that fails, device installation

indicates that the system should be rebooted.

3. If the INF is not for a PnP driver and you are using a method such as [InstallHInfSection](#) to process the INF, then manually stop and restart the driver:

- Close all open handles to the driver and then stop the driver using one of the following:

- `sc.exe stop <mydriver>`
- `ControlService(SERVICE_CONTROL_STOP)`

For more info, see [ControlService function](#).

If you are not using an INF file, use these steps:

1. Stop the driver, as described above. Replace the old driver binary file with the new one.
2. If you can't stop the driver, rename the existing file, copy the new file into place, and set up the existing file to be deleted in the future (for example, using [MoveFileEx](#) with the `MOVEFILE_DELAY_UNTIL_REBOOT` flag). In order to start using the new version of the driver, the system will need to be restarted.

## Related topics

[File-Backed and Page-File-Backed Sections](#)

[What Determines When a Driver Is Loaded](#)

# Initiating System Restarts During Device Installations

11/2/2020 • 2 minutes to read • [Edit Online](#)

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

- During initial installations, a device's installer or co-installer can request a system restart by setting DI\_NEEDRESTART 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 system restart is necessary.

# Device Installations on 64-Bit Systems

11/2/2020 • 2 minutes to read • [Edit Online](#)

If your device will be installed on both 32-bit platforms and 64-bit platforms, you must follow these steps when you create a [driver package](#):

- Provide both 32-bit and 64-bit compilations of all kernel-mode drivers, *device installation application*, *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 the Microsoft Windows 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 inbox drivers. Instead, it must call [CreateProcess](#) (described in the Windows 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.

# Registry Trees and Keys for Devices and Drivers

11/2/2020 • 2 minutes to read • [Edit Online](#)

The operating system, drivers, and device installation components store information about drivers and devices in the registry. In general, drivers and device installation components should use the registry to store data that must be maintained across restarts of the system. For information about how a driver accesses registry information, see [Using the Registry in a Driver](#).

Registry contents should always be treated as untrusted, modifiable information. If one of your driver components writes information to the registry and another component reads it later, do not assume that the information has not been modified in the meantime. After reading information from the registry, your driver components should always validate the information before using it.

For more information about the registry in general, see the Microsoft Windows SDK documentation.

This section contains the following topics which describe the use of registry keys to store information about drivers and devices:

[Registry Trees for Devices and Drivers](#)

[RunOnce Registry Key](#)

[DeviceOverrides Registry Key](#)

Drivers must access Plug and Play (PnP) keys in the registry using system routines such as [IoGetDeviceProperty](#) or [IoOpenDeviceRegistryKey](#). User-mode setup components should use device installation functions such as [SetupDiGetDeviceRegistryProperty](#) or [SetupDiOpenDevRegKey](#). The registry can be accessed from INF files by using [INF AddReg directives](#).

**Important** *Drivers must not access these registry trees and keys directly.* This discussion of registry information in this section is solely for debugging a device installation or configuration problem.

# Registry Trees for Devices and Drivers

9/25/2019 • 2 minutes to read • [Edit Online](#)

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

- [HKLM\SYSTEM\CurrentControlSet\Services Registry Tree](#)
- [HKLM\SYSTEM\CurrentControlSet\Control Registry Tree](#)
- [HKLM\SYSTEM\CurrentControlSet\Enum Registry Tree](#)
- [HKLM\SYSTEM\CurrentControlSet\HardwareProfiles Registry Tree](#)

For information on accessing registry keys from WDF (KMDF or UMDF) drivers, see [Introduction to Registry Keys for Drivers](#).

For information on accessing registry keys from WDM drivers, see [Plug and Play Registry Routines](#).

# HKLM\SYSTEM\CurrentControlSet\Services Registry Tree

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 of a driver in the *RegistryPath* parameter when it calls the driver's **DriverEntry** routine. A driver can store global driver-defined data under the **Parameters** subkey of its key in the **Services** tree. Information that is 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. Windows creates this value by 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 `%SystemRoot%\system32\Drivers\DriverNamesys`, where *DriverName* is the name of the driver's **Services** key.

## Parameters

A key that is 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.

# HKLM\SYSTEM\CurrentControlSet\Control Registry Tree

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 that is 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, and registered class lower-filter drivers.

## **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](#).

# HKLM\SYSTEM\CurrentControlSet\Enum Registry Tree

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **Enum** tree is reserved for use by operating system components, and its layout is subject to change. Drivers and user-mode [Device Installation Components](#) must use system-supplied functions, such as [IoGetDeviceProperty](#) and [SetupDiGetDeviceRegistryProperty](#), to extract information from this tree. *Drivers and Windows applications must not access the **Enum** tree directly.*

# HKLM\SYSTEM\CurrentControlSet\HardwareProfiles Registry Tree

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **HKLM\SYSTEM\CurrentControlSet\HardwareProfiles** registry tree contains information about the hardware profiles on the system. Hardware profiles are deprecated and state should not be stored relative to a hardware profile.

Instead, store information globally using **PLUGPLAY\_REGKEY\_DEVICE** and **PLUGPLAY\_REGKEY\_DRIVER** without also using **PLUGPLAY\_REGKEY\_CURRENT\_HWPROFILE**. For more info, see

[IoOpenDeviceRegistryKey](#).

# RunOnce Registry Key

11/2/2020 • 3 minutes to read • [Edit Online](#)

All versions of Windows support a registry key, **RunOnce**, which can be used to specify commands that the system will execute one time and then delete.

In Windows 8 and Windows 8.1, **RunOnce** entries for installation of software-only SWENUM devices are processed during device installation. Other **RunOnce** entries are added to the **RunOnce** key. These are applied the next time the system processes the **RunOnce** key. Device installation does not force the system to process **RunOnce** entries.

In Windows 7 and previous versions, immediately after a device is installed, Windows executes the command stored under the **RunOnce** key and then removes the key. Additionally, each time the system starts, it executes the command stored under the **RunOnce** key and then removes the key. Therefore, if you put a command under the **RunOnce** key, you cannot easily predict when it is executed.

Immediately after a device has been installed, Windows executes the command stored under the **RunOnce** key and then removes the key. Additionally, each time the system starts, it executes the command stored under the **RunOnce** key and then removes the key. Therefore, if you put a command under the **RunOnce** key, you cannot easily predict when it is executed.

For device installations, **RunOnce** registry keys can be created by using *add-registry-sections*, which are specified through [INF AddReg directives](#). Each *add-registry-section* has the following syntax:

```
reg-root, [subkey], [value-entry-name], [flags], [value]
```

The registry root (*reg-root*) and subkey values for the **RunOnce** registry key are as follows:

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

The *value-entry-name* string is omitted from a **RunOnce** registry entry. The type of the entry, which is indicated by the *Flags* value, must be either [REG\\_SZ](#) (*Flags* value of 0x00000000) or [REG\\_EXPAND\\_SZ](#) (*Flags* value of 0x00010000). For an entry of type [REG\\_SZ](#) (the default), the *Flags* value can be omitted.

The *value* parameter in a **RunOnce** key specifies the command to be executed. This parameter is a quoted string that has the following format:

```
Rundll32[.exe] DllName,EntryPoint[Arguments]
```

By default, a **RunOnce** key is deleted after the specified command is executed. You can prefix a **RunOnce** key *value* parameter with an exclamation point (!) to defer deletion of the key until after the command runs successfully. Without the exclamation point prefix, if the specified command fails, the **RunOnce** key will still be deleted and the command will not be executed the next time that the system starts.

Also, by default, the **RunOnce** keys are ignored when the system is started in Safe Mode. The *value* parameter of **RunOnce** keys can be prefixed with an asterisk (\*) to force the command to be executed even in Safe Mode.

Consider the following guidelines when you create a *value* string entry:

- *Rundll32* can appear either with or without its *.exe* file name extension.
- *DllName* is the full path of a DLL or executable image. Except for a required terminating comma, the expression must not otherwise contain any commas. If no file name extension is supplied, the default extension is *.dll*.

- *EntryPoint* is the name of the entry point within the DLL indicated by *DllName*.
- *Arguments* is an optional substring that contains any arguments that must be passed to the specified DLL.
- Exactly one space must separate the *EntryPoint* string from the *Arguments* substring.

The following code example shows the *add-registry-section* entry that stores a command and its arguments under the **RunOnce** key:

```
;; WDMAud swenum install

HKLM,%RunOnce%,"WDM_WDMAUD",,,
"rundll32.exe streamci.dll,StreamingDeviceSetup
%WDM_WDMAUD.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_WDMAUD%,%17%\WDMAUDIO.inf,WDM_WDMAUD.Interface.Install"

[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}"
```

The following rules apply when you use **RunOnce** registry keys for device installations:

- These registry keys must be used only for installations of software-only devices that are enumerated by SWENUM, the software device enumerator.
- **RunOnce** keys must consist only of calls to *Rundll32.exe*. Otherwise, WHQL will not digitally sign the [driver package](#).
- The code to be executed must not prompt for user input.
- Server-side installations execute in a system context. For this reason, you must be certain that the code to be executed contains no security vulnerabilities and that file permissions prevent the code from being maliciously modified.
- Starting with Windows Vista, the system will not execute the commands specified by the **RunOnce** keys if a user without administrator privileges is logged on to the system. This could lead to incomplete or corrupted installations following a system restart.

Before the *device installation application* creates the **RunOnce** entries, it informs the current user that a user who has administrator privileges must log on after a system restart.

For more information, see [Developing Applications that Run at Logon on Windows Vista](#).

# DeviceOverrides Registry Key

12/1/2020 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **DeviceOverrides** registry key specifies that one or more removable device capability overrides exist in the system. For more information about the removable device capability, see [Overview of the Removable Device Capability](#).

The Plug and Play (PnP) manager uses a new ID ([Container IDs](#)) to group one or more device nodes (*devnodes*) that originated from and belong to each instance of a particular physical device installed in the computer. For legacy devices, the PnP manager generates container IDs through the removable device capability. For more information about how the PnP manager generates container IDs, see [How Container IDs are Generated](#).

Removable device capability overrides let the independent hardware vendor (IHV) or original equipment manufacturer (OEM) change the interpreted value of the removable device capability on a devnode or group of devnodes.

Removable device capability overrides through the **DeviceOverrides** registry key are useful for legacy devices or third-party hardware components that may not report the removable device capability correctly. This causes the PnP manager to incorrectly generate a container ID used to group the devnodes enumerated from a physical device.

These overrides do not actually change the global state of the removable device capability reported by a devnode. Instead, these overrides cause the PnP manager to ignore the reported device capability and use the registry-based setting when generating a [Container ID](#) for devnodes that match an override. Additional subkeys under the **DeviceOverrides** registry subkey provide more details about which devnodes to override.

The following table defines the **DeviceOverrides** registry key's format and requirements.

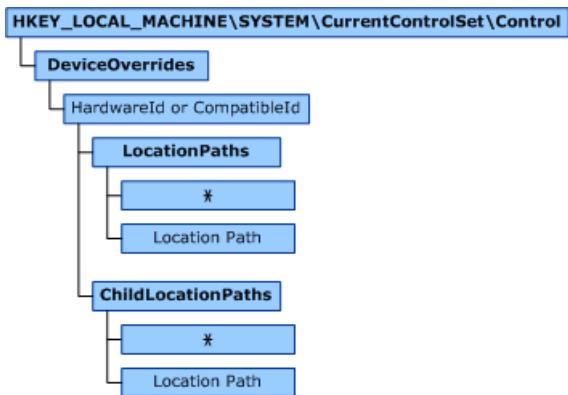
REGISTRY KEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT KEY	CHILD SUBKEYS
DeviceOverrides	Optional	None	None	HardwareID or CompatibleID

Each removable device capability override is specified through either the [HardwareID](#) or [ContainerID](#) registry subkeys.

The **DeviceOverrides** registry key is created and maintained under the [HKLM\SYSTEM\CurrentControlSet\Control](#) registry tree. Within this registry key, one or more removable device capability overrides are created or maintained.

Removable device capability overrides are specific to individual devices specified through either the [HardwareID](#) or [CompatibleID](#) registry subkeys. Additional subkeys define the paths of devnodes enumerated for the specified devices. Generally, the most specific device hardware ID should be used to identify a device, instead of a less specific hardware or compatible ID. This ensures that the removable device capability override is not applied to any unintended devices that share the same hardware or compatible ID as the intended target device.

The following figure shows the topology of the **DeviceOverrides** registry key and its related subkeys.



The **DeviceOverrides** registry key must be created for the first removable device capability override that is added to the system. It may not exist by default on a clean operating system installation.

**Note** The existence of a removable device capability registry override does not change the global state of the removable device capability on a devnode.

# HardwareID Registry Subkey

3/5/2019 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **HardwareID** registry subkey specifies a removable device capability override for a device installed in the computer. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The name of the **HardwareID** registry subkey specifies the [hardware ID](#) of the device, and is formatted based on the requirements described below.

The following table defines the format and requirements of the **HardwareID** registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT KEY	CHILD SUBKEYS
Valid <a href="#">hardware ID</a> value	Required	Must include the hardware ID's bus prefix.  All slash () path separator characters within the hardware ID must be replaced with number (#) characters.	<a href="#">DeviceOverrides</a>	

The hardware ID value must follow the format requirements described in this table. Each **HardwareID** subkey must contain either the **LocationPaths** or **ChildLocationPaths** subkeys. Both subkeys could be specified within the **HardwareID** subkey if necessary.

Because the slash character is not a valid character in a registry subkey name, you must replace it with the number character when specifying a bus prefix for the **HardwareID** registry subkey name. For example, if a removable device capability override is specified for the device node (*devnode*) with a [hardware ID](#) of USB\VID\_1234&PID\_ABCD&REV\_0001, you must create a **HardwareID** registry subkey with a name of USB#VID\_1234&PID\_ABCD&REV\_0001.

# CompatibleID Registry Subkey

3/5/2019 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **CompatibleID** registry subkey specifies a removable device capability override for a device installed in the computer. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The name of the **CompatibleID** registry subkey specifies the [compatible ID](#) of the device, and is formatted based on the requirements described below.

The following table defines the format and requirements of the **CompatibleID** registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT KEY	CHILD SUBKEYS
Valid <a href="#">compatible ID</a> value	Required	Must include the compatible ID's bus prefix.  All slash () path separator characters within the compatible ID must be replaced with number (#) characters.	<a href="#">DeviceOverrides</a>	<a href="#">LocationPaths</a> and/or <a href="#">ChildLocationPaths</a>

The compatible ID value must follow the format requirements described in this table. Each **CompatibleID** subkey must contain either the **LocationPaths** or **ChildLocationPaths** subkeys. Both subkeys could be specified within the **CompatbleID** subkey if necessary.

Because the slash character is not a valid character in a registry subkey name, you must replace it with the number character when specifying a bus prefix for the **CompatibleID** registry subkey name. For example, if a removable device capability override is specified for the device node (*devnode*) with a [hardware ID](#) of PCI\VEN\_ABCD&DEV\_1234&SUBSYS\_000, you must create a **CompatibleID** registry subkey with a name of PCI#VEN\_ABCD&DEV\_1234&SUBSYS\_000.

# LocationPaths Registry Subkey

12/1/2020 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **LocationPaths** registry subkey is used in the specification of a removable device capability override for a device identified through either the [HardwareID](#) or [CompatibleID](#) registry subkey. The **LocationPaths** registry subkey specifies that only the location path of the device's parent device node (*devnode*) will have the removable device capability override applied. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The **LocationPaths** registry subkey applies to only the parent devnode of the device specified through the name of the [HardwareID](#) or [CompatibleID](#) subkeys. As a result, only the parent devnode of the specified device is affected by the removable device capability override value. Child devnodes of the specified device are not affected by the removable device capability override, unless a [ChildLocationPaths](#) registry subkey is also specified.

The following table defines the format and requirements of the **LocationPaths** registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT SUBKEY	CHILD SUBKEYS
<b>LocationPaths</b>	Optional	None	<a href="#">HardwareID</a> or <a href="#">CompatibleID</a>	<a href="#">LocationPath</a> or *

Either the **LocationPaths** or [ChildLocationPaths](#) registry subkeys must be present to indicate the parent/child relationship to which the removable device capability override applies.

# ChildLocationPaths Registry Subkey

12/1/2020 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **ChildLocationPaths** registry subkey is used in the specification of a removable device capability override for a device identified through either the **HardwareID** or **CompatibleID** registry subkey. The **ChildLocationPaths** registry subkey specifies that only the location path of the device's child device nodes (*devnodes*) will have the removable device capability override applied. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The **ChildLocationPaths** registry subkey applies to only the child devnodes of the device specified through the name of the parent **HardwareID** or **CompatibleID** subkeys. As a result, only the child devnodes of the specified device are affected by the removable device capability override value. The parent devnode of the specified device are not affected by the removable device capability override, unless a **LocationPaths** registry subkey is also specified or a **ChildLocationPaths** registry subkey is specified for the parent devnode.

The following table defines the format and requirements of the **ChildLocationPaths** registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT SUBKEY	CHILD SUBKEYS
ChildLocationPaths	Optional	None	HardwareID or CompatibleID	LocationPath or *

**Note** Either the **LocationPaths** or **ChildLocationPaths** registry subkeys must be present to indicate the parent/child relationship to which the removable device capability override applies.

# LocationPath Registry Subkey

3/6/2019 • 3 minutes to read • [Edit Online](#)

Beginning with Windows 7, the **LocationPath** registry subkey specifies the location path for a removable device capability override of a single device identified through either the **HardwareID** or **CompatibleID** registry subkey. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The **LocationPath** registry subkey applies the removable device capability value to only the device node (*devnode*) that exists at the specified location path. This lets the removable device capability override be applied to a single instance of a device installed in the system. Other devices with the same **HardwareID** or **CompatibleID** at other location paths are not affected by such a removable device capability override.

By convention, the location path string takes the form *ServiceName(BusSpecificLocation)*. For example, PCI devices use PCI (XXYY), where XX is the device number and YY is the function number. The string is unique to the device in relation to its bus. The Plug and Play (PnP) manager assembles the location path for each node in the devnode tree. Each devnode in the tree concatenates its service name string to the end of the location path string that its parent devnode supplied. Therefore, the position of any devnode in the tree can be uniquely identified through the location path.

The following table defines the format and requirements of the **LocationPath** registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT SUBKEY	CHILD SUBKEYS
Valid "LocationPath" value	Optional (* or a valid location path must be present to indicate the scope of the removable device capability override)	None	<a href="#">LocationPaths</a> or <a href="#">ChildLocationPaths</a>	None

Either the **LocationPath** or \* registry subkeys must be present to indicate the scope of the removable device capability override.

The **LocationPath** subkey must contain a **Removable** DWORD value that specifies whether the device is removable or not. The following table defines the valid **Removable** values.

REMovable VALUE	EXPLANATION
0	The devnode should be regarded as not removable
1	The devnode should be regarded as removable

The location path string for a given devnode can be displayed through Device Manager through the following steps:

1. Open Device Manager and locate the devnode on which the registry override is to be applied. To do this, you may be required to change the view to **Devices by connection**.

2. Right-click the devnode, click **Properties** and then click the **Details** tab.
3. In the **Property** drop-down list, find the **LocationPaths** property. This property contains the location path string for this devnode and is the value that should be used for the **LocationPath** registry subkey.

**Note** It is possible that the devnode does not have a **LocationPaths** value. This is because the driver for this devnode or one of its parents does not implement the [GUID\\_PNP\\_LOCATION\\_INTERFACE](#) interface. In this case, you must check the parent devnode for a **LocationPaths** property.

The **LocationPaths** registry subkey is intended to be used for overriding the removable device capability of devices that are hardwired to a fixed bus location. This typically occurs in portable computers, and includes the following devices:

- Wireless network adapters
- Bluetooth adapters
- Keyboards or pointing devices

These devices exist on different internal buses at fixed locations that the user cannot change. The **LocationPaths** override lets you specify that only the device at the given bus location is affected by the removable device capability override. This prevents the override from affecting devices at other bus locations that may share the same **HardwareID** or **CompatibleID** subkey value as the override target. This is common when devices specify only a **CompatibleID** subkey value to match an inbox driver.

When you use a **ChildLocationPaths** registry subkey to override the removable device capability of child devnodes, it is often useful to target only child devnodes at specific locations, regardless of what kind of devices they are.

For example, a laptop may have an internal USB hub with both internal and external ports. If this USB hub is misreporting its internal ports as being external, any device that is internally hardwired to these ports is incorrectly recognized as being removable. Similarly, if all ports are misreported as being internal, any externally connected device is treated as if it is a nondetachable part of the laptop.

To discover the location paths value for a device that is connected to an external USB port, you can plug any device into the port and observe its location paths property. Any other USB devices that are plugged into the same port should receive the same location paths value, because the parent bus and how it internally identifies a port never changes.

# \* Registry Subkey

10/8/2019 • 2 minutes to read • [Edit Online](#)

Beginning with Windows 7, the \* (asterisk) registry subkey specifies that a removable device capability override applies to all device nodes (*devnodes*) enumerated for the device identified through either the [HardwareID](#) or [CompatibleID](#) registry subkey. For more information about removable device capability overrides, see [DeviceOverrides Registry Key](#).

The \* registry subkey applies the removable device capability override to all devnodes specified through the [HardwareID](#) or [CompatibleID](#) registry subkeys, based on the rules of the [LocationPaths](#) or [ChildLocationPaths](#) registry subkey specified for the override. For example, if the \* registry subkey is specified within a [LocationPaths](#) subkey, the removable device capability override applies to all parent devnodes for devices identified through the parent [HardwareID](#) or [CompatibleID](#) registry subkey.

The following table defines the format and requirements of the \* registry subkey.

REGISTRY SUBKEY NAME	REQUIRED/OPTIONAL	FORMAT REQUIREMENTS	PARENT SUBKEY	CHILD SUBKEYS
*	Optional	None	<a href="#">LocationPaths</a> or <a href="#">ChildLocationPaths</a>	None

Either the [LocationPath](#) or \* registry subkeys must be present to indicate the scope of the removable device capability override.

The \* registry subkey must contain a **Removable** DWORD value that specifies whether the device is removable or not. The following table defines the valid **Removable** values.

REMovable VALUE	EXPLANATION
0	The applicable devnodes should be regarded as not removable
1	The applicable devnodes should be regarded as removable

# INF ClassInstall32 Section

12/1/2020 • 8 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

A **ClassInstall32** section installs a new [device setup class](#) (and possibly a class installer) for devices in the new class.

```
[ClassInstall32] |
[ClassInstall32.nt] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntarm] | (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] | (Windows 10 version 1709 and later versions of Windows)
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)

AddReg=add-registry-section[,add-registry-section]...
[AddProperty=add-property-section[,add-property-section] ...] (Windows Vista and later versions of Windows)
[Copyfiles=@filename | file-list-section[,file-list-section]...]
[DelReg=del-registry-section[,del-registry-section]...]
[DelProperty=del-property-section[,del-property-section] ...] (Windows Vista and later versions of Windows)
[Delfiles=file-listsection[,file-list-section]...]
[Renfiles=file-list-section[,file-list-section]...]
[BitReg=bit-registry-section[,bit-registry-section]...]
[UpdateInis=update-ini-section[,update-ini-section]...]
[UpdateIniFields=update-inifields-section[,update-inifields-section]...]
[Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...]
```

## Entries

### **AddReg** = *add-registry-section*[,*add-registry-section*] ...

References one or more named sections that contain class-specific value entries to be written into the registry. Typically, this is used to give the new device setup class at least a friendly name that other components can later retrieve from the registry and use to open installed devices of this new device class, to "install" any new device class installer and/or property-page provider for this device setup class, and so forth.

An **HKR** specification in any *add-registry-section* designates the `..Class\{SetupClassGUID}` registry key. For additional information, see the following **Remarks** section.

For more information, see [INF AddReg Directive](#).

### **AddProperty** = *add-property-section*[,*add-property-section*] ...

(Windows Vista and later versions of Windows) References one or more INF file sections that modify [device properties](#) that are set for a [device setup class](#). You should use an [INF AddProperty directive](#) only to set a device setup class property that is new to Windows Vista or later versions of Windows operating systems.

For device class properties that were introduced earlier on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry entry values, you should continue to use [INF AddReg directives](#) to set the device setup class properties. These guidelines apply to system-defined properties and custom properties.

For more information about how to use the **AddProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

**Copyfiles**=@*filename* | *file-list-section*[,*file-list-section*] ...

Either specifies one named file to be copied from the source media to the destination or references one or more named sections in which class-relevant files on the source media are specified for transfer to the destination. The **DefaultDestDir** entry in the **DestinationDirs** section of the INF specifies the destination directory for any class-specific single file to be copied.

For more information, see [INF CopyFiles Directive](#).

**Note** System-supplied INF files for device setup classes (and class installers) do not use this directive in this section.

**DelReg**=*del-registry-section*[,*del-registry-section*] ...

References one or more named sections in which value entries or keys are specified to be removed from the registry during installation of the class installer.

However, if a particular {*SetupClassGUID*} subkey exists in the registry ..Class branch, the system setup code subsequently ignores the **ClassInstall32** section of any INF that specifies the same GUID value in its **Version** section. Consequently, an INF cannot replace an existing class installer or modify its behavior from a **ClassInstall32** section. To modify the behavior of existing class installer, use a class-specific co-installer.

For more information, see [INF DelReg Directive](#).

**DelProperty**=*del-property-section*[,*del-property-section*] ...

(Windows Vista and later versions of Windows) References one or more INF file sections that delete **device properties** that are set for a **device setup class**. You should use an **INF DelProperty directive** only to delete a device setup class property that is new to Windows Vista or later versions of Windows operating systems.

For device class properties that were introduced earlier on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry entry values, you should continue to use **INF DelReg directives** to delete the device setup class properties. These guidelines apply to system-defined properties and custom properties.

For more information about how to use the **DelProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

**Delfiles**=*file-listsection*[,*file-list-section*] ...

References one or more named sections in which previously installed class-relevant files on the destination are specified for deletion.

For more information, see [INF DelFiles Directive](#).

**Renfiles**=*file-list-section*[,*file-list-section*] ...

References one or more named sections in which class-relevant files to be renamed on the destination are listed.

For more information, see [INF RenFiles Directive](#).

**BitReg**=*bit-registry-section*[,*bit-registry-section*]...

Is valid in this section but almost never used.

For more information, see [INF BitReg Directive](#).

**UpdateInis**=*update-ini-section*[,*update-ini-section*]...

Is valid in this section but almost never used.

For more information, see [INF UpdateInis Directive](#).

**UpdateIniFields=** *update-inifields-section[, update-inifields-section]...*

Is valid in this section but almost never used.

For more information, see [INF UpdateIniFields Directive](#).

**Ini2Reg=** *ini-to-registry-section[, ini-to-registry-section]...*

Is valid in this section but almost never used.

For more information, see [INF UpdateIniFields Directive](#).

## Remarks

You should include a **ClassInstall32** section in a device INF file only to install a new custom device setup class. INF files for devices in an installed class, whether a [system-supplied device setup class](#) or a custom class, should not include a **ClassInstall32** section. Because the system processes a **ClassInstall32** section only if a class is not already installed, you cannot use a **ClassInstall32** section to reinstall or change the settings for a class that is already installed. In particular, you cannot use a **ClassInstall32** section to add a class co-installer or a class filter driver for a class that is already installed. For information about how to install co-installers and filter drivers, see [Writing a Co-installer](#) and [Installing a Filter Driver](#).

Usually, a **ClassInstall32** section has one or more **AddReg** directives to add entries under a system-provided *SetupClassGUID* subkey in the registry. These entries can include a class-specific "friendly name," class installer path, class icon, property page provider, and so forth.

Except for **AddReg** and **CopyFiles**, the other directives shown here are rarely used in a **ClassInstall32** section.

To support a multiplatform distribution of driver files, construct platform-specific **ClassInstall32** sections. For example, all system SetupAPI functions that process a **ClassInstall32** section will search first for a **ClassInstall32.ntx86** section on an x86 platform and only examine an undecorated **ClassInstall32** section if they cannot find a **ClassInstall32.ntx86** section. For more information about how to use the system-defined **.nt**, **.ntx86**, **.ntia64**, **.ntamd64**, **.ntarm**, and **.ntarm64** extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

**Note** The **ClassInstall32** section name is also used for installations on 64-bit platforms.

Starting with Windows 2000, every installed device is associated with a [device setup class](#) in the registry. If the INF for a device to be installed is not associated with a new device class installer, or if its **ClassGUID=** specification in the **Version** section does not match a system-defined setup class GUID, that device's registry subkey is created under **..Class\{UnknownClassGUID}**.

The INF for any device class installer typically has an **AddReg** directive in its **ClassInstall32** section, to define at least one named section that creates a friendly name for its kind of device. The setup code automatically creates a *SetupClassGUID* subkey in the registry from the value supplied for the **ClassGUID=** entry in the INF's **Version** section when the first device of that (new) setup class is installed.

Under this *SetupClassGUID* subkey, such an INF also provides registry information for *Models*-specific subkeys by using additional **AddReg** directives in its **DD\Install** section. In addition, the INF can use the add-registry sections referenced in its **ClassInstall32** section to specify a property-page provider and to exert control over how its class of devices is handled in the user interface.

Such a class-specific add-registry section has the following general form:

```
[SetupClassAddReg]

HKR,,,,%DevClassName% ; device-class friendly name
[HKR,,Installer32,, "class-installer.dll, class-entry-point"]
[HKR,,EnumPropPages32,, "prop-provider.dll, provider-entry-point"]
HKR,,Icon,, "icon-number"
[HKR,,SilentInstall,,1]
[HKR,,NoInstallClass,,1]
[HKR,,NoDisplayClass,,1]
```

The system uses the specified icon to represent your installer to the user.

- If the Icon value is positive, it represents a resource identifier for a resource. The resource is extracted from the class installer DLL, if the Installer32 key is specified, or from the property page DLL, if the EnumPropPages32 key is specified. The value "0" represents the first icon in the DLL. The value "1" is reserved.
- If the Icon value is negative, the absolute value is the resource identifier of the icon in SetupApi.DLL.

Setting the predefined **SilentInstall**, **NoDisplayClass**, and **NoInstallClass** Boolean value entries in a class-specific registry key has the following effects:

- Setting SilentInstall directs installers to send no popup messages to the user that require a response while installing devices of this class, whether specified in the DDInstall sections of the class installer's INF file or in separate INF files for subsequently installed devices that declare themselves of this class by setting the same ClassGuid={ClassGUID} specification in their respective Version sections. For example, the system class installers of CD-ROM and disk devices and the system parallel port class installer set SilentInstall in their respective registry keys.

If a class-specific installer requires the computer to be restarted for any device that it installs, the class-specific add-registry section in its INF cannot have this value entry.

- Setting NoDisplayClass suppresses the user-visible display of all devices of this class by Device Manager. For example, the system class installers for printers and for network drivers (including clients, services, and protocols) set NoDisplayClass in their respective registry keys.
- Setting NoInstallClass indicates that no device of this type will ever require manual installation by an end-user. For example, the system class installers for exclusively Plug and Play (PnP) devices set NoInstallClass in their respective registry keys.

A **ClassInstall32** section can contain **AddReg** directives to set the **DeviceType**, **DeviceCharacteristics**, and **Security** for devices of its setup class. See the [INF AddReg Directive](#) for more information.

## Examples

This example shows the **ClassInstall32** section, along with the named section it references with the [AddReg directive](#), of the INF for the system display class installer.

```
[ClassInstall32]
AddReg=display_class_addreg

[display_class_addreg]
HKR,,,,%DisplayClassName%
HKR,,Installer32,, "Desk.Cpl,DisplayClassInstaller"
HKR,,Icon,, "-1"
```

By contrast, this example shows the add-registry section referenced in the system CD-ROM INF's **ClassInstall32** section. It sets up a class-specific property-page provider for the CD-ROM devices/drivers

that it installs. This INF also sets the **SilentInstall** and **NoInstallClass** value entries in the CD-ROM class key to **TRUE** (1).

```
[cdrom_class_addrreg]
HKR,,,,%CDCClassName%
HKR,,EnumPropPages32,,SysSetup.Dll,CdromPropPageProvider"
HKR,,SilentInstall,,1
HKR,,NoInstallClass,,1
HKR,,Icon,,101"
```

## See also

[AddProperty](#)

[AddReg](#)

[BitReg](#)

[CopyFiles](#)

*DDInstall*

[DelFiles](#)

[DelProperty](#)

[DelReg](#)

[Ini2Reg](#)

*Models*

[RenFiles](#)

[SetupDiBuildClassInfoListEx](#)

[UpdateIniFields](#)

[UpdateInis](#)

[Version](#)

# INF ClassInstall32.Services Section

11/2/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

A **ClassInstall32** section installs a new [device setup class](#) (and possibly a class installer) for devices in the new class.

```
[ClassInstall32.Services] |
[ClassInstall32.nt.Services] |
[ClassInstall32.ntx86.Services] |
[ClassInstall32.ntarm.Services] | (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64.Services] | (Windows 10 version 1709 and later versions of Windows)
[ClassInstall32.ntia64.Services] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64.Services] | (Windows XP and later versions of Windows)

AddService=ServiceName,[flags],service-install-section
    [,event-log-install-section[,,[EventLogType][,EventName]]]...
[DelService=ServiceName[,,[flags][,[EventLogType][,EventName]]]]...
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
```

Each **ClassInstall32.Services** section contains one or more [INF AddService directives](#) that reference additional INF-writer-defined sections in an INF file.

INF files typically use the **ClassInstall32.Services** section with at least one **AddService** directive to control how and when the services of a particular device class are loaded, any dependencies it might have on other services, and so forth. Optionally, they can also set up event-logging services for the device class.

## Entries

**AddService**=*ServiceName*[*flags*],*service-install-section*

[,*event-log-install-section*[,*[EventLogType]*[,*EventName*]]]...

This directive references an INF-writer-defined service-install-section and, possibly, an event-log-install-section elsewhere in the INF file for the drivers of the device class covered by the [ClassInstall32](#) section. For more information, see [INF AddService Directive](#).

**DelService**=*ServiceName*[*[flags]*[,*[EventLogType]*[,*EventName*]]]...

This directive removes a previously installed service from the target computer. This directive is very rarely used. For more information, see [INF DelService Directive](#).

**Include**=*filename.inf*[,*filename2.inf*]...

This optional entry specifies one or more additional system-supplied named INF files that contain sections needed to install this device class. If this entry is specified, usually so is a **Needs** entry.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**=*inf-section-name*[,*inf-section-name*]...

This optional entry specifies the particular named section that must be processed during the installation of this device class. Typically, such a named section is an **ClassInstall32.Services** section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within such a

**ClassInstall32.Services** section.

Needs entries cannot be nested. (For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#)).

## Remarks

**ClassInstall32.Services** sections should have the same platform and operating system decorations as their related [ClassInstall32 sections](#). For example, a **ClassInstall32.ntx86** section would have a corresponding **ClassInstall32.ntx86.Services** section.

The case-insensitive **.nt**, **.ntx86**, **.ntia64**, **.ntamd64**, **.ntarm**, and **.ntarm64** extensions can be inserted into a **ClassInstall32.Services** section name in cross-platform INF files, as shown in the formal syntax statement. For more information, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## See also

[ClassInstall32](#)

[AddService](#)

*DDInstall*

[DDInstall.HW](#)

[DelService](#)

*Models*

# INF ControlFlags Section

4/29/2020 • 5 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

A **ControlFlags** section identifies devices for which Windows should take certain unique actions during installation.

```
[ControlFlags]

ExcludeFromSelect=* |
ExcludeFromSelect=device-identification-string[,device-identification-string] ...] |
[ExcludeFromSelect.nt=device-identification-string[,device-identification-string] ...] |
[ExcludeFromSelect.ntx86=device-identification-string[,device-identification-string] ...] |
[ExcludeFromSelect.ntia64=device-identification-string[,device-identification-string] ...] | (Windows XP and
later versions of Windows)
[ExcludeFromSelect.ntamd64=device-identification-string[,device-identification-string] ...] | (Windows XP
and later versions of Windows)
[ExcludeFromSelect.ntarm=device-identification-string[,device-identification-string] ...] | (Windows XP and
later versions of Windows)
[ExcludeFromSelect.ntarm64=device-identification-string[,device-identification-string] ...] | (Windows XP
and later versions of Windows)

[CopyFilesOnly=device-identification-string[,device-identification-string] ...]
[InteractiveInstall=device-identification-string[,device-identification-string] ... ]
[RequestAdditionalSoftware=*] |
[RequestAdditionalSoftware=device-identification-string[,device-identification-string] ...] (Windows 7 and
later versions of Windows)
```

## Entries

### *device-identification-string*

Identifies a [hardware ID](#) or [compatible ID](#) that was specified in a per-manufacturer [INF Models section](#). Each string must be separated from the next with a comma (,).

### **ExcludeFromSelect**

Removes all (if \* is specified) or the specified list of devices from certain user interface displays, from which a user is expected to select a particular device for installation.

For Windows 2000 and later versions of Windows, the specified devices are displayed by the Found New Hardware Wizard and the Hardware Update Wizard.

To exclude a set of OS-incompatible or platform-incompatible devices from this display, one or more **ExcludeFromSelect** entries can have the following case-insensitive extensions appended:

#### **.nt**

Do not display these devices on computers that are running Windows 2000 or later versions of Windows.

#### **.ntx86**

Do not display these devices on x86-based computers that are running Windows 2000 or later versions of Windows.

#### **.ntia64**

Do not display these devices on Itanium-based computers that are running Windows XP or later versions of

Windows.

#### **.ntamd64**

Do not display these devices on x64-based computers that are running Windows XP or later versions of Windows.

#### **.ntarm**

Do not display these devices on ARM-based computers that are running Windows XP or later versions of Windows.

#### **.ntarm64**

Do not display these devices on ARM64-based computers that are running Windows XP or later versions of Windows.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

#### **CopyFilesOnly**

Installs only the INF-specified files for the given devices because the device hardware is not accessible or available yet.

This entry is rarely used. However, it can be used to preinstall the drivers of a device for which the card will later be seated in a particular slot that is currently in use. For example, if a device currently seated in the particular slot is necessary to transfer INF-specified files to the target, the INF would have this entry.

#### **InteractiveInstall**

Forces the specified list of devices to be installed in a user's context. Each line can specify one or more hardware IDs or compatible IDs, and there can be one or more lines.

This entry is optional. The preferred way to install devices is to omit this entry and allow Windows to install the device in the context of a trusted system thread, if possible. However, if a device absolutely requires a user to be logged in when the device is installed, include this entry in the device INF.

#### **RequestAdditionalSoftware**

Specifies that all (if \* is specified) or the specified list of devices may require additional software than what was installed through the [driver package](#) for the device. For example, the RequestAdditionalSoftware entry can be used to install new or updated device-specific software that was not included in the driver package.

**Note** If \* is not specified, each device specified by a RequestAdditionalSoftware entry must be defined within the [INF Models section](#).

This entry is optional, and is supported in Windows 7 and later versions of Windows operating system.

After Windows installs the [driver package](#) for the device, the Plug and Play (PnP) manager performs the following steps if the RequestAdditionalSoftware entry is specified within the INF file:

1. The PnP manager generates a Problem Report and Solution (PRS) error report with the type of RequestAdditionalSoftware. This report contains information about the specific hardware ID of the device and the system architecture of the computer.
2. If there is a solution provided by the independent hardware vendor (IHV) for the device-specific software, the solution is downloaded to the computer.

**Note** The download of the solution does not install the software itself.

3. If the device-specific software is not installed on the computer, the PnP manager presents the solution to the user and provides a link for downloading the software. The user can then choose to download and install this software by following the instructions presented in the solution.

## **Remarks**

Typically, a **ControlFlags** section has one or more **ExcludeFromSelect** entries to identify devices that are listed in the per-manufacturer [INF Models section](#), but which should not be displayed to the end-user as options during manual installations.

Listing a device's **hardware ID** or **compatible ID** in an **ExcludeFromSelect** entry removes it from the display shown to the end-user. Specifying an asterisk (\*) for the **ExcludeFromSelect** value removes all devices/models defined in the INF file from this user-visible list.

An INF writer should use the **InteractiveInstall** directive sparingly and only in the following situations:

- To install drivers for devices that have corrupted or otherwise incorrectly defined hardware IDs. For example, when two or more different devices share the same Hardware ID. This case is strictly forbidden by the Plug and Play standard, but some hardware vendors have made this error in hardware.
- To install drivers for devices that require their own driver and absolutely cannot use the generic class driver or another driver supplied with the operating system. The **InteractiveInstall** entry forces Device Manager to ask the user for confirmation for compatible ID matches.

**Note** In the future, WHQL might not grant the Windows Logo to devices whose INF files include **InteractiveInstall** entries.

INF files that exclusively install PnP devices can have a **ControlFlags** section unless they set the **NoInstallClass** value entry in their respective *SetupClassGUID* registry keys to **TRUE**. For more information about these registry keys, see [INF ClassInstall32 Section](#).

## Examples

This example of the **ControlFlags** section in the system mouse class installer INF suppresses the display of devices/models that cannot be installed on x86 platforms.

```
[ControlFlags]
; Exclude all bus mice and InPort mice for x86 platforms
ExcludeFromSelect.ntx86=*PNP0F0D,*PNP0F11,*PNP0F00,*PNP0F02,*PNP0F15
; Hide this entry always
ExcludeFromSelect=UNKNOWN_MOUSE
```

The following INF file fragment shows two devices: one that is fully PnP-capable and requires no user intervention during installation and another that requires its own driver and cannot use any other driver. Specifying **InteractiveInstall** for the second device forces Windows to install this device in a user's context (a user who has administrative rights). This includes prompting the user for the location of the driver files (INF file, driver file, and so on) as required.

```
; ...
[Manufacturer]
%Mfg% = ModelsSection

[ModelsSection]
; Models section, with two entries
%Device1.DeviceDesc% = Device1.Install, \
    PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_01
%Device2.Device.Desc% = Device2.Install, \
    PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02

[ControlFlags]
InteractiveInstall = \
    PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02
; ...
```

## See also

[ClassInstall32](#)

[Manufacturer](#)

*Models*

# INF DDInstall Section

11/2/2020 • 11 minutes to read • [Edit Online](#)

Each per-Models *DDInstall* section contains an optional **DriverVer** directive and one or more directives referencing additional named sections in the INF file, shown here with the most frequently specified INF directives, **CopyFiles** and **AddReg**, listed first.

The sections referenced by these directives contain instructions for installing driver files and writing any device-specific and/or driver-specific information into the registry.

```
[install-section-name] |
[install-section-name.nt] |
[install-section-name.ntx86] |
[install-section-name.ntia64] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64] (Windows XP and later versions of Windows)

[DriverVer=mm/dd/yyyy[,x.y.v.z] ]
[CopyFiles=@filename | file-list-section[,file-list-section] ...]
[CopyINF=filename1.inf[,filename2.inf]...] (Windows XP and later versions of Windows)
[AddReg=add-registry-section[,add-registry-section]...]
[AddProperty=add-registry-section[,add-registry-section]...] (Windows Vista and later versions
of Windows)
[Include=filename1.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
[Delfiles=file-list-section[,file-list-section]...]
[Renfiles=file-list-section[,file-list-section]...]
[DelReg=del-registry-section[,del-registry-section]...]
[DelProperty=add-registry-section[,add-registry-section]...] (Windows Vista and later versions
of Windows)
[FeatureScore=featurescore]... (Windows Vista and later versions of Windows)
[BitReg=bit-registry-section[,bit-registry-section]...]
[LogConfig=log-config-section[,log-config-section]...]
[ProfileItems=profile-items-section[,profile-items-section]...] (Microsoft Windows 2000 and
later versions of Windows)
[UpdateInis=update-ini-section[,update-ini-section]...]
[UpdateIniFields=update-inifields-section[,update-inifields-section]...]
[Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...]
[RegisterDlls=register-dll-section[,register-dll-section]...]
[UnregisterDlls=unregister-dll-section[,unregister-dll-section]...]
[ExcludeID=device-identification-string[,device-identification-string]...]... ((Windows XP and
later versions of Windows)
[Reboot]
```

## Entries

**DriverVer= mm/dd/yyyy[,x.y.v.z]**

This optional entry specifies version information for the [driver package](#).

For information about how to specify this entry, see [INF DriverVer Directive](#).

**CopyFiles=@ filename | file-list-section[,file-list-section] ...**

This directive either specifies one named file to be copied from the source media to the destination or references one or more INF-writer-defined sections in which device-relevant files on the source media are specified for transfer to the destination. The **CopyFiles** directive is optional, but is present in most *DDInstall* sections.

The **DefaultDestDir** entry in the [DestinationDirs](#) section of the INF specifies the destination for

any single file to be copied. The **SourceDiskNames** and **SourceDiskFiles** sections, or an additional INF specified in the **LayoutFile** entry of this INF's **Version** section, provide the location on the distribution media of the driver files.

For more information, see [INF CopyFiles Directive](#).

**CopyINF=filename1.inf[,filename2.inf]...**

(Windows XP and later) This directive causes specified INF files to be copied to the target system.

For more information, see [INF CopyINF Directive](#).

**AddReg=add-registry-section[,add-registry-section]...**

This directive references one or more INF-writer-defined sections in which new subkeys, possibly with initial value entries, are specified to be written into the registry or in which the value entries of existing keys are modified.

An **HKR** specification in such an add-registry section designates the **.Class\SetupClassGUID\device-instance-id** registry path of the user-accessible driver. This type of **HKR** specification is also referred to as a "software key".

For more information, see [INF AddReg Directive](#).

**AddProperty=add-registry-section[,add-registry-section]...**

(Windows Vista and later) References one or more INF file sections that modify **device properties** that are set for a device instance. You should use an [INF AddProperty directive](#) only to set a device instance property that is new to Windows Vista or later versions of Windows operating systems.

For device instance properties that were introduced earlier on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry entry values, you should continue to use [INF AddReg directives](#) to set the device instance properties. These guidelines apply to system-defined properties and custom properties. For more information about how to use the **AddProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

**Include=filename1.inf[,filename2.inf]...**

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to install this device and/or driver. If this entry is specified, usually so is a **Needs** entry.

For example, the system INF files for device drivers that depend on the system's kernel-streaming support specify this entry as follows:

```
Include= ks.inf[, [kscaptur.inf,] [ksfilter.inf]]...
```

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs=inf-section-name[,inf-section-name]...**

This optional entry specifies sections within system-supplied INF files that must be processed during the installation of this device. Typically, such a named section is a **DDInstall** (or **DDInstall.xxx**) section within one of the INF files that are listed in an **Include** entry. However, it can be any section that is referenced within such a **DDInstall** or **DDInstall.xxx** section of the included INF.

For example, the INF files for device drivers that have the preceding **Include** entry specify this entry as follows:

```
Needs= KS.Registration[, KSCAPTUR.Registration | KSCAPTUR.Registration.NT,  
MSPCLOCK.Installation]
```

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Delfiles**= *file-list-section*[, *file-list-section*]...

This directive references one or more INF-writer-defined sections listing files on the target to be deleted.

For more information, see [INF DelFiles Directive](#).

**Renfiles**= *file-list-section*[, *file-list-section*]...

This directive references one or more INF-writer-defined sections listing files to be renamed on the destination before device-relevant source files are copied to the target computer.

For more information, see [INF RenFiles Directive](#).

**DelReg**= *del-registry-section*[, *del-registry-section*]...

This directive references one or more INF-writer-defined sections in which keys and/or value entries are specified to be removed from the registry during installation of the devices.

Typically, this directive is used to handle upgrades when an INF must clean up old registry entries from a previous installation of this device.

An **HKR** specification in such a delete-registry section designates the ..Class\SetupClassGUID\device-instance-id registry path of the user-accessible driver. This type of HKR specification is also referred to as a "software key".

For more information, see [INF DelReg Directive](#).

**DelProperty**= *add-registry-section*[, *add-registry-section*]...

(Windows Vista and later) References one or more INF file sections that delete **device properties** that are set for a device instance. You should use an [INF DelProperty directive](#) only to delete a device instance property that is new to Windows Vista or later versions of Windows.

For device instance properties that were introduced earlier on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry entry values, you should continue to use [INF DelReg directives](#) to delete the device instance properties. These guidelines apply to system-defined properties and custom properties. For more information about how to use the **DelProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

**FeatureScore**= *featurescore*

**Warning** The **FeatureScore** directive is only processed when specified directly in the [DDInstall] section.

(Windows Vista and later) This directive provides an additional ranking criterion for drivers that are based on the features that a driver supports. For example, feature scores might be defined for a [device setup class](#) that distinguishes between drivers based on class-specific criteria.

For more information about how drivers are ranked, see [How Windows Ranks Drivers \(Windows Vista and Later\)](#).

For more information about this directive, see [INF FeatureScore Directive](#).

**Note** Although a *DDInstall* section can contain multiple **FeatureScore** entries, only the first entry is processed for the section.

**BitReg**=*bit-registry-section*[,*bit-registry-section*]...

This directive references one or more INF-writer-defined sections in which existing registry value entries of type **REG\_BINARY** are modified.

An **HKR** specification in such a bit-registry section designates the `..Class\SetupClassGUID\device-instance-id` registry path of the user-accessible driver. This type of **HKR** specification is also referred to as a "software key".

For more information, see [INF BitReg Directive](#).

**LogConfig**=*log-config-section*[,*log-config-section*]...

This directive references one or more INF-writer-defined sections within an INF for a root-enumerated device or for a manually installed device. In these named sections, the INF for such a "detected" or manually installed device specifies one or more logical configurations of bus-relative hardware resources that the device must have to be operational. The INF for such a manually installed device that is not software-configurable should also have a `DDInstall.FactDef` section.

The **LogConfig** directive is never used to install Plug and Play (PnP) devices. However, you can use an [INF DDInstall.LogConfigOverride section](#) to provide an override configuration for PnP devices.

This directive is irrelevant to all higher-level (nondevice) drivers and components.

For more information, see [INF LogConfig Directive](#).

**ProfileItems**=*profile-items-section*[,*profile-items-section*]...

(Microsoft Windows 2000 and later versions of Windows) This directive references one or more INF-writer-defined sections that describe items to be added to, or removed from, the Start menu.

For more information, see [INF ProfileItems Directive](#).

**UpdateInis**=*update-ini-section*[,*update-ini-section*]...

This rarely used directive references one or more INF-writer-defined sections, specifying a source INI file from which a particular section or line within such a section is to be read into a destination INI file of the same name during installation. Optionally, line-by-line modifications to an existing INI file on the destination from a given source INI file of the same name can be specified in the update-ini section.

For more information, see [INF UpdateInis Directive](#).

**UpdateIniFields**=*update-inifields-section*[,*update-inifields-section*]...

This rarely used directive references one or more INF-writer-defined sections in which modifications within the lines of a device-specific INI file are specified.

For more information, see [INF UpdateIniFields Directive](#).

**Ini2Reg**=*ini-to-registry-section*[,*ini-to-registry-section*]...

This rarely used directive references one or more INF-writer-defined sections in which sections or lines from a device-specific INI file, supplied on the source media, are to be moved into the registry.

For more information, see [INF Ini2Reg Directive](#).

**RegisterDlls**=*register-dll-section*[,*register-dll-section*]...

This directive references one or more INF sections used to specify files that are OLE controls and require self-registration.

For more information, see [INF RegisterDlls Directive](#).

**UnregisterDlls**=*unregister-dll-section*[,*unregister-dll-section*]...

This directive references one or more INF sections used to specify files that are OLE controls and require self-unregistration (self-removal).

For more information, see [INF UnregisterDIs Directive](#).

**ExcludeID=** *device-identification-string*[,*device-identification-string*]...

**Warning** The **ExcludeID** directive is only processed when specified directly in the [DDInstall] section.

(Windows XP and later) This directive specifies one or more device identification strings (either **hardware IDs** or **compatible IDs**). The **DDInstall** section does not install devices that have **device IDs** that match any of the hardware IDs or compatible IDs listed.

#### Reboot

This directive indicates that the caller should be prompted to reboot the system after installation is complete.

For more information, see [INF Reboot Directive](#).

## Remarks

Throughout the Windows Driver Kit (WDK) documentation, the term **DDInstall** is used to refer to an *install-section-name*, with or without platform extensions. Therefore, "**DDInstall** section" means "a named section within an INF, having the format [*install-section-name*] or [*install-section-name*.**ntxxx**]". When you create names for **DDInstall** sections, you should include a device-specific prefix, such as [**WDMPNPB003\_Device**] or [**GPR400.Install.NT**].

Each **DDInstall** section must be referenced in a device/models-specific entry under the per-manufacturer [INF Models section](#) of the INF file.

Except for devices that have no associated files to be transferred from the source media, an INF file that installs a WDM driver on different operating system platforms must have at least one of the following **DDInstall** sections:

- An *install-section-name.ntx86* section that specifies the entries for device/driver installations specific to x86-based platforms.
- An *install-section-name.ntia64* section that specifies the entries for device/driver installations specific to Itanium-based platforms.
- An *install-section-name.ntamd64* section that specifies the entries for device/driver installations specific to x64-based platforms.
- An *install-section-name.ntarm* section that specifies the entries for device/driver installations specific to ARM-based platforms.
- An *install-section-name.ntarm64* section that specifies the entries for device/driver installations specific to ARM64-based platforms.
- An *install-section-name* or *install-section-name.nt* section that specifies the entries for device/driver installations that are not specific to a particular hardware platform.

For more information about how to use the system-defined **.nt**, **.ntx86**, **.ntia64**, **.ntamd64**, **.ntarm**, and **.ntarm64** extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

Starting with Windows 2000, an INF file that installs drivers must have **DDInstall.Services** sections to specify device/driver registry information to be stored in the registry's ...\\CurrentControlSet\\Services tree. Depending on the device, it can also have one or more

*DDInstall.HW*, *DDInstall.CoInstallers*, *DDInstall.Interfaces*, and/or *DDInstall.LogConfigOverride* sections.

Each directive in a *DDInstall* section can reference more than one section name. However, each additional named section must be separated from the next with a comma (,).

Each section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

Any **AddReg** directive specified in a *DDInstall* section is assumed to reference an add-registry section that cannot be used to store information about upper or lower-filter drivers, about multifunction devices, or about driver-independent but device-specific parameters. If a device/driver INF must store this type of information in the registry, it must use an **AddReg** directive in its undecorated and decorated *DDInstall.HW* sections, if any, to reference another INF-writer-defined *add-registry-section*.

Depending on the [device setup class](#) that was specified in the [INF Version section](#), additional class-specific directives can be specified in a *DDInstall* section. For more information about class-specific directives, see the following topics:

- [Building an INF File for a Windows SideShow-Compatible Device](#)
- [DDInstall Section in a Network INF File](#)
- [INF Files for Still Image Devices](#)
- [INF Files for WIA Devices](#)
- [Installation Requirements for Network Components](#)
- [Specifying WDF Directives in INF Files](#)

## Examples

This example shows the expansion of the *DDInstall* sections, **Ser\_Inst** and **Inp\_Inst**. These sections are referenced in the example for the [INF Models section](#).

```
[Ser_Inst]
CopyFiles=Ser_CopyFiles, mouclass_CopyFiles

[Ser_CopyFiles]
sermouse.sys

[mouclass_CopyFiles] ; section name referenced by > 1 CopyFiles
mouclass.sys

[Inp_Inst]
CopyFiles=Inp_CopyFiles, mouclass_CopyFiles

[Inp_CopyFiles]
inport.sys
```

The following example provides a general illustration of using platform extensions:

```

[Manufacturer]
%MSFT% = Microsoft

[Microsoft]
%Device.DeviceDesc% = DeviceInstall, HWID
[DeviceInstall.NTx86]
;
; This section is used for installations on x86 systems.
;
...
[DeviceInstall.NTx86.Services]
;
; Services installation for x86 systems.
;
...
[DeviceInstall.NT]
;
; This section is used for installations on systems (all other architectures).
;
...
[DeviceInstall.NT.Services]
;
; Services installation for systems (all other architectures).
;
...

```

The following example shows the *DD\Install* section of an INF file that installs a system-supplied WDM driver for an audio device on various operating system platforms:

```

[WDMNPB003_Device.NT]
DriverVer=01/14/1999,5.0
Include=ks.inf, wdmaudio.inf
Needs=KS.Registration, WDMAUDIO.Registration.NT
LogConfig=SB16.LC1,SB16.LC2,SB16.LC3,SB16.LC4,SB16.LC5
; a few log-config-sections omitted here for brevity
CopyFiles=MSSB16.CopyList
AddReg=WDM_SB16.AddReg

```

The following example shows the sections referenced by the preceding **Needs** entry in the system-supplied *ks.inf* and *wdmaudio.inf* files. In the preceding example, these files are specified in the **Includes** entry. When the operating system's device installer and/or media class installer process this device's *install-section-name* section, these next two sections are also processed.

```

[KS.Registration]
; following AddReg= is actually a single line in the ks.inf file
AddReg=ProxyRegistration,CategoryRegistration, \
    TopologyNodeRegistration,PlugInRegistration,PinNameRegistration, \
    DeviceRegistration
CopyFiles=KSProxy.Files,KSDriver.Files

[WDMAUDIO.Registration.NT]
AddReg=WDM.AddReg
CopyFiles=WDM.CopyFiles.Sys, WDM.CopyFiles.Drv
;
; INF-writer-defined add-registry and file-list sections
; referenced by preceding directives are omitted here for brevity
;
```

## See also

[AddProperty](#)

[\*DDInstall\*.ColInstallers](#)

[\*DDInstall\*.FactDef](#)

[\*DDInstall\*.HW](#)

[\*DDInstall\*.Interfaces](#)

[\*DDInstall\*.LogConfigOverride](#)

[\*DDInstall\*.Services](#)

[DefaultInstall](#)

[DefaultInstall.Services](#)

[DelProperty](#)

[FeatureScore](#)

# INF DDInstall.CoInstallers Section

12/1/2020 • 7 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

This optional section registers one or more device-specific co-installers supplied on the distribution media to supplement the operations of existing device class installers.

```
[install-section-name.CoInstallers] |
[install-section-name.nt.CoInstallers] |
[install-section-name.ntx86.CoInstallers] |
[install-section-name.ntarm.CoInstallers] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.CoInstallers] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.CoInstallers] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.CoInstallers] | (Windows XP and later versions of Windows)

AddReg=add-registry-section[,add-registry-section]...
CopyFiles=@filename | file-list-section[,file-list-section]...
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
[DelFiles=file-list-section[,file-list-section]...]
[RenFiles=file-list-section[,file-list-section]...]
[DelReg=del-registry-section[,del-registry-section]...]
[BitReg=bit-registry-section[,bit-registry-section]...]
[UpdateInis=update-ini-section[,update-ini-section]...]
[UpdateIniFields=update-inifields-section[,update-inifields-section]...]
[Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...]
...
```

## Entries

**AddReg**=*add-registry-section*[,*add-registry-section*]...

References one or more INF-writer-defined *add-registry-sections* that store registry information about the supplied co-installers.

An **HKR** specified in such an add-registry section designates the `..Class\SetupClassGUID\device-instance-id` registry path of the user-accessible driver, which is also referred to as a "software key". Therefore, for a device-specific co-installer, it writes (or modifies) a **CoInstallers32** value entry in this user-accessible per-device/driver "software" subkey.

For a class-specific co-installer, it registers the new co-installers by modifying the contents of the appropriate `..CoDeviceInstallers\SetupClassGUID` subkeys. The path of the appropriate registry `SetupClassGUID` subkeys must be explicitly specified in the referenced add-registry sections.

For more information, see [INF AddReg Directive](#).

**CopyFiles**=@*filename*|*file-list-section*[,*file-list-section*]...

Transfers the source co-installer files to the destination on the target computer, usually by referencing one or more INF-writer-defined *file-list-sections* elsewhere in the INF file. Such a file-list section specifies the co-installer files to be copied from the source media to the destination directory on the target.

However, system INF files that install co-installers never use this directive in a `DDInstall.CoInstallers` section.

For more information, see [INF CopyFiles Directive](#).

**Include**= *filename.inf*[, *filename2.inf*]...

Specifies one or more additional system-supplied INF files that contain sections needed to install the co-installers for this device or [device setup class](#). If this entry is specified, usually so is a **Needs** entry. (For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#)).

**Needs**= *inf-section-name*[, *inf-section-name*]...

Specifies the particular sections that must be processed during the installation of this device. Typically, such a named section is a *DDInstall*.**CoInstallers** section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall*.**CoInstallers** section of the included INF.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**DelFiles**= *file-list-section*[, *file-list-section*]...

References a file-list section specifying files to be removed from the target. This directive is rarely used.

For more information, see [INF DelFiles Directive](#).

**RenFiles**= *file-list-section*[, *file-list-section*]...

References a file-list section specifying files on the destination to be renamed before co-installer source files are copied to the target. This directive is also rarely used.

For more information, see [INF RenFiles Directive](#).

**DelReg**= *del-registry-section*[, *del-registry-section*]...

References one or more INF-writer-define *delete-registry-sections*. Such a section specifies stale registry information about the co-installers for a previous installation of the same devices that should be removed from the registry. An HKR specified in such a delete-registry section designates the same registry subkey as already described for the **AddReg** entry. This directive is very rarely used in a *DDInstall*.**CoInstallers** section.

For more information, see [INF DelReg Directive](#).

**BitReg**= *bit-registry-section*[, *bit-registry-section*]...

This entry is valid in this section but almost never used. An HKR specified in such a bit-registry section designates the same registry subkey as already described for the **AddReg** entry.

For more information, see [INF BitReg Directive](#).

**UpdateInis**= *update-ini-section*[, *update-ini-section*]...

This entry is valid in this section but almost never used.

For more information, see [INF UpdateInis Directive](#).

**UpdateIniFields**= *update-inifields-section*[, *update-inifields-section*]...

This entry is valid in this section but almost never used.

For more information, see [INF UpdateIniFields Directive](#).

**Ini2Reg**= *ini-to-registry-section*[, *ini-to-registry-section*]...

This entry is valid in this section but almost never used.

For more information, see [INF Ini2Reg Directive](#).

## Remarks

The specified *DDInstall* section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file.

If an INF includes a `DDInstall.CoInstallers` section, there must be one for each platform-decorated and undecorated `DDInstall` section. For example, if an INF contains an `[install-section-name.ntx86]` section and an `[install-section-name]` section and it registers device-specific co-installers, then the INF must include both an `[install-section-name.ntx86.CoInstallers]` section and an `[install-section-name.CoInstallers]` section. For more information about how to use the system-defined `.nt`, `.ntx86`, `.ntia64`, `.ntamd64`, `.ntarm`, and `.ntarm64` extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

Each directive in a `DDInstall.CoInstallers` section can reference more than one INF-writer-defined section name. However, each additional named section must be separated from the next with a comma (,).

Each directive-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

A co-installer is a Win32 DLL that typically writes additional configuration information to the registry or performs other installation tasks that require dynamically generated, system-specific information that is not available when an INF is created. A device-specific co-installer supplements the installation operations either of the OS's device installer or of the appropriate class installer when that device is installed.

For more information about how to write and using co-installers, see [Writing a Co-installer](#).

## Installing Co-installer Images

All co-installer files must be copied into the `%SystemRoot%\system32` directory. Like any INF `CopyFiles` operation, the destination is controlled explicitly for a named `file-list-section` in the `DestinationDirs` section of the INF file by the `dirid` value 11 or by supplying this `dirid` value for the `DefaultDestDir` entry.

## Registering Device-Specific Co-installers

Registering one or more device-specific co-installers requires adding a `REG_MULTI_SZ`-typed value entry to the registry. Specify an `add-registry-section` referenced by the `AddReg` directive, by using the following general form:

```
[DDInstall.CoInstallers_DeviceAddReg]  
  
HKR,,CoInstallers32,0x00010000,"DevSpecificCoInstall.dll  
[,DevSpecificEntryPoint]"[,,"DevSpecific2CoInstall.dll  
[,DevSpecific2EntryPoint]"...]
```

The `HKR` entry is listed as a single line within the INF file, and each supplied device-specific co-installer DLL must have a unique name. After the listed co-installers are registered, the system's device installer calls them at each subsequent step of the installation process for that device.

If the optional `DevSpecificEntryPoint` is omitted, the default `CoDeviceInstall` routine name is used as the entry point of the co-installer DLL.

For more information, see [Registering a Device-Specific Co-installer](#).

## Registering Device-Class Co-installers

To add a value entry (and setup-class subkey, if it does not exist already) for one or more device-class co-installers to the registry, an `add-registry-section` referenced by the `AddReg` directive has the following general form:

```
[DDInstall.CoInstallers_ClassAddReg]  
  
HKLM,System\CurrentControlSet\Control  
  \CoDeviceInstallers,{SetupClassGUID},  
  0x00010008,"DevClssCoInst.dll[,DevClssEntryPoint]"  
  ...
```

Each entry in such an add-registry section is listed as a single line within the INF file, and each supplied class co-installer DLL must have a unique name. If the supplied co-installers should be used for more than one device setup class, this add-registry section can have more than one entry, each with the appropriate *SetupClassGUID* value.

Such a supplemental device-class co-installer must not replace any already registered co-installers for an existing class installer. Therefore, the class co-installer must have a unique name and the **REG\_MULTI\_SZ**-type value supplied must be appended (as indicated by the 8 in the *flags* value **0x0010008**) to the class-specific co-installer entries, if any, already present in the {*SetupClassGUID*} subkey.

**Note** The [SetupAPI](#) functions never append a duplicate *DevC\ssCoInstall.dll* to a value entry if a co-installer of the same name is already registered.

The INF for a supplemental device-class co-installer can be activated by a right-click install or through a call to [SetupInstallFromInfSection](#) made by a *device installation application*.

## Examples

This example shows the *DDInstall.CoInstallers* section for IrDA serial network adapters. The system-supplied INF for these IrDA (serial) NICs supplies a co-installer to the system IrDA class installer.

```
; DDInstall section
[PNP.NT]
AddReg=ISIR.reg, Generic.reg, Serial.reg
PromptForPort=0      ; This is handled by IRCLASS.DLL
LowerFilters=SERIAL ; This is handled by IRCLASS.DLL
BusType=14
Characteristics=0x4 ; NCF_PHYSICAL

; ... PNP.NT.Services section omitted here
[PNP.NT.CoInstallers]
AddReg = ISIR.CoInstallers.reg
; ...

[IRSIR.reg]
HKR, Ndi, HelpText, 0, %IRSIR.Help%
HKR, Ndi, Service, 0, "IRSIR"
HKR, Ndi\Interfaces, DefUpper, 0, "ndisirda"
HKR, Ndi\Interfaces, DefLower, 0, "nolower"
HKR, Ndi\Interfaces, UpperRange, 0, "ndisirda"
HKR, Ndi\Interfaces, LowerRange, 0, "nolower"

[Generic.reg]
HKR,,InfraredTransceiverType,0,"0"

[Serial.reg]
HKR,,SerialBased,0, "0"

[ISIR.CoInstallers.reg]
HKR,,CoInstallers32,0x00010000,"IRCLASS.dll,IrSIRClassCoInstaller"

; ... Services and Event Log registry sections omitted here
[Strings]
; ...
IRSIR.Help = "An IrDA serial infrared device is a built-in COM port or
external transceiver which transmits infrared pulses. This NDIS
miniport driver installs as a network adapter and binds to the FastIR
protocol."
```

The preceding PNP.NT.CoInstallers section only referenced a co-installer-specific *add-registry* section. It has no [CopyFiles](#) directive because this system-supplied INF installs a set of IrDA network devices. Like all system INF files, this INF file uses the [LayoutFile](#) entry in its [Version](#) section to transfer the co-installer file to the

destination.

Be aware that any *DDInstall\CoInstallers* section in an INF supplied by an IHV or OEM also has a **CopyFiles** directive, along with **SourceDisksNames** and **SourceDisksFiles** sections.

## See also

[AddReg](#)

[BitReg](#)

[CopyFiles](#)

[DDInstall](#)

[DelFiles](#)

[DelReg](#)

[DestinationDirs](#)

[Ini2Reg](#)

[RenFiles](#)

[SourceDisksFiles](#)

[SourceDisksNames](#)

[UpdateIniFields](#)

[UpdateInis Version](#)

# INF DDInstall.Components Section

4/29/2020 • 2 minutes to read • [Edit Online](#)

This optional section contains one or more [INF AddComponent directives](#) that reference additional INF-writer-defined sections in a driver package INF file. This section is supported for Windows 10 Version 1703 and later.

```
[install-section-name.Components] |
[install-section-name.nt.Components] |
[install-section-name.ntx86.Components] |
[install-section-name.ntia64.Components] |
[install-section-name.ntamd64.Components] |
[install-section-name.ntarm.Components] |
[install-section-name.ntarm64.Components] |

AddComponent=ComponentName,[flags],component-install-section
```

You can provide a *DDInstall.Components* section with one or more **AddComponent** directives to create a symbolic relationship between a driver package and any number of software components.

## Entries

**AddComponent**=*ComponentName*,*[flags]*,*component-install-section*

This directive references an INF-writer-defined component-install-section elsewhere in the INF file for the drivers of the devices covered by this *DDInstall* section. For more information, see [INF AddComponent Directive](#).

## Remarks

*DDInstall.Components* sections should have the same platform and operating system decorations as their related *DDInstall* sections. For example, an *install-section-name*.**ntx86** section would have a corresponding *install-section-name*.**ntx86.Components** section.

The specified *DDInstall* section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall.Components* section name in cross-platform INF files.

For more information about how to use the system-defined .**nt**, .**ntx86**, .**ntia64**, .**ntamd64**, .**ntarm**, and .**ntarm64** extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

```
[ContosoGrafx.NT.Components]
AddComponent = ContosoControlPanel1,,Component_Inst

[Component_Inst]
ComponentIDs = VID0001&PID0001&SID0001
DisplayName = %ContosoControlPanelDisplayName%
```

## See also

[Using a Component INF File](#)

## INF AddComponent Directive

# INF DDInstall.Events Section

11/2/2020 • 2 minutes to read • [Edit Online](#)

Each per-Models *DDInstall.Events* section contains one or more [INF AddEventProvider directives](#) that reference additional INF-writer-defined sections in an INF file. This section is supported for Windows 10 version 1809 and later.

```
[install-section-name.Events] |
[install-section-name.nt.Events] |
[install-section-name.ntx86.Events] |
[install-section-name.ntia64.Events] |
[install-section-name.ntamd64.Events] |
[install-section-name.ntarm.Events] |
[install-section-name.ntarm64.Events] |

AddEventProvider={ProviderGUID},event-provider-install-section
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
```

You can provide a *DDInstall.Events* section with at least one **AddEventProvider** directive to register [Event Tracing for Windows](#) (ETW) providers.

## Entries

**AddEventProvider={ProviderGUID},event-provider-install-section**

This directive references an INF-writer-defined *event-provider-install-section* elsewhere in the INF file for the drivers of the devices covered by this *DDInstall* section. For more information, see [INF AddEventProvider Directive](#).

**Include=filename.inf[,filename2.inf]...**

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to install this device. If this entry is specified, a **Needs** entry is also usually required.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs=inf-section-name[,inf-section-name]...**

This optional entry specifies the section that must be processed during the installation of this device. Typically, the section is a *DDInstall.Events* section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within a *DDInstall.Events* section.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

## Remarks

*DDInstall.Events* sections should have the same platform and operating system decorations as their related *DDInstall* sections. For example, an *install-section-name.ntx86* section would have a corresponding *install-section-name.ntx86.Events* section.

The specified *DDInstall* section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall.Events* section name in cross-platform INF files.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

This example shows the *install-section-name.Events* section and its event-provider-install-sections in the INF file.

```
[Device_Inst.NT.Events]
AddEventProvider={071acb53-ccfb-42e0-9a68-5336b7301507},foo_Event_Provider_Inst
AddEventProvider={6d3fd9ef-bcbb-42d7-9fdb-1bf2d926b394},bar_Event_Provider_Inst

; entries in the following xxx_Inst sections omitted here for brevity,
; but fully specified as the example for the AddEventProvider directive
;
[foo_Event_Provider_Inst]
; ...

[bar_Event_Provider_Inst]
; ...
```

## See also

[AddEventProvider](#)

[\*DDInstall\*](#)

# INF DDInstall.FactDef Section

4/29/2020 • 4 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

This section should be used in an INF for any manually installed non-PnP device that an end-user might install. This section specifies the factory-default hardware configuration settings, such as the bus-relative I/O ports and IRQ (if any), for such a card.

```
[install-section-name.FactDef] |
[install-section-name.nt.FactDef] |
[install-section-name.ntx86.FactDef] |
[install-section-name.ntarm.FactDef] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.FactDef] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.FactDef] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.FactDef] (Windows XP and later versions of Windows)

ConfigPriority=Priority-Value
[DMAConfig=[DMAAttrs:]DMANum]
[IOConfig=io-range]
[MemConfig=mem-range]
[IRQConfig=[IRQAttrs:]IRQNum]
...
...
```

## Entries

### **ConfigPriority = Priority-Value**

Specifies one of the following priority values for this factory-default logical configuration.

PRIORITY VALUE	MEANING
FORCECONFIG	Specifies a forced configuration, which identifies the resources that the PnP manager must assign to a device.
DESIRED	Provides the highest device performance. The PnP manager can dynamically configure the device with this configuration.
NORMAL	Provides greater device performance than SUBOPTIMAL, but less performance than DESIRED. This is the typical priority value. The PnP manager can dynamically configure the device with this configuration.
SUBOPTIMAL	Provides the lowest device performance. This configuration is not desirable, but it will work. The PnP manager can dynamically configure this configuration.
RESTART	Requires a system restart.
REBOOT	Requires a system restart.
POWEROFF	Requires a power cycle.

PRIORITY VALUE	MEANING
HARDRECONFIG	Requires a jumper change.
HARDWIRED	Cannot be changed.
DISABLED	Indicates that the device is disabled.

**DMAConfig=[DMA attrs.] DMA Num**

Specifies the bus-relative DMA channel as a decimal number. *DMA attrs* is optional if the device is connected on a bus that has only 8-bit DMA channels and the device uses standard system DMA. Otherwise, it can be one of the letters **D** for 32-bit DMA, **W** for 16-bit DMA, and **N** for 8-bit DMA, with **M** if the device uses bus-master DMA, and with one of the following (mutually exclusive) letters that indicate the type of DMA channel used: **A**, **B**, or **F**. If none of **A**, **B**, or **F** is specified, a standard DMA channel is assumed.

**IOConfig=io-range**

Specifies the I/O port range for the device in the following form:

```
start-end[[decode-mask][:alias-offset][:attr]])
```

*start*

Specifies the (bus-relative) starting address of the I/O port range as a 64-bit hexadecimal value.

*end*

Specifies the ending address of the I/O port range, also as a 64-bit hexadecimal value.

*decode-mask*

Defines the alias type and can be any of the following.

MASK VALUE	MEANING	IOR_ALIAS VALUE
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	Positive decode	0xFF

*alias-offset*

Not used.

*attr*

Specifies the letter **M** if the specified range is in system memory. If omitted, the specified range is in I/O port space.

**MemConfig=mem-range**

Specifies the memory range for the device in the following form:

```
start-end[(attr)]
```

*start*

Specifies the starting (bus-relative) address of the device memory range as a 64-bit hexadecimal value.

*end*

Specifies the ending address of the memory range, also as a 64-bit hexadecimal value.

*attr*

Specifies the attributes of the memory range as one or more of the following letters:

- **R** (read-only)
- **W** (write-only)
- **RW** (read/write)
- **C** (combined write allowed)
- **H** (cacheable)
- **F** (prefetchable)
- **D** (card decode addressing is 32-bit, instead of 24-bit)

If both **R** and **W** are specified or if neither is specified, read/write is assumed.

**IRQConfig=[/RQAttrs]/RQNum**

Specifies the bus-relative IRQ that the device uses as a decimal number. */RQAttrs* is omitted if the device uses a bus-relative, edge-triggered IRQ. Otherwise, specify **L** to indicate a level-triggered IRQ, and **LS** if the device can share the IRQ line listed in this entry.

## Remarks

The specified *DDInstall* section must be referenced in a device-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall.FactDef* section name in cross-operating system and/or cross-platform INF files. For more information about these system-defined extensions, see [Creating an INF File](#).

This section must contain complete factory-default information for installing one device. The INF should specify this set of entries in the order best suited to how the driver initializes its device. If necessary, it can have more than one of any particular kind of entry.

For example, the INF for a device that used two DMA channels would have two **DMAConfig=** lines in its *DDInstall.FactDef* section.

The INF files of manually installed devices for which the factory-default logical configuration settings can be changed should also use the **LogConfig** directive in their *DDInstall* sections. In general, such an INF should specify the entries in each of its log config sections and in its *DDInstall.FactDef* section in the same order.

## Examples

This **IOConfig** entry specifies an I/O port region, 8 bytes in size, which can start at 2F8.

```
IOConfig=2F8-2FF
```

This **MemConfig** entry specifies a memory region of 32K bytes that can start at D0000.

```
MemConfig=D0000-D7FFF
```

## See also

[DDInstall](#)

[LogConfig](#)



# DDInstall.HW Section

11/2/2020 • 4 minutes to read • [Edit Online](#)

*DDInstall.HW* sections are typically used for installing multifunction devices, for installing PnP filter drivers, and for setting up any user-accessible device-specific but driver-independent information in the registry, whether with explicit **AddReg** directives or with **Include** and **Needs** entries.

```
[install-section-name.HW] |
[install-section-name.nt.HW] |
[install-section-name.ntx86.HW] |
[install-section-name.ntarm.HW] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.HW] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.HW] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.HW] | (Windows XP and later versions of Windows)

[AddReg=add-registry-section[,add-registry-section]...] ...
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
[DelReg=del-registry-section[,del-registry-section]...] ...
[BitReg=bit-registry-section[,bit-registry-section] ...]
```

## Entries

**AddReg**=*add-registry-section*[,*add-registry-section*]...

References one or more INF-writer-defined *add-registry-sections* elsewhere in the INF file for the devices covered by this *DDInstall.HW* section. The *add-registry-section* typically installs filters and/or stores per-device information in the registry. An **HKR** specification in such an *add-registry-section* specifies the device's *hardware key*, a device-specific registry subkey that contains information about the device. A hardware key is also called a device key. For more info, see [Registry Trees and Keys for Devices and Drivers](#). A driver package can add settings via an INF by using an **HKR** specification in an *add-registry-section* referenced by a *DDInstall.HW* section.

For more information, see [INF AddReg Directive](#).

**Include**=*filename.inf*[,*filename2.inf*]...

Specifies one or more additional system-supplied INF files that contain sections needed to install this device. If this entry is specified, usually so is a **Needs** entry.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**=*inf-section-name*[,*inf-section-name*]...

Specifies the named sections that must be processed during the installation of this device. Typically, such a named section is a *DDInstall.HW* section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall.HW* section of the included INF.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**DelReg**=*del-registry-section*[,*del-registry-section*]...

References one or more INF-writer-defined *delete-registry-sections* elsewhere in the INF file for the drivers of the devices covered by this *DDInstall* section. Such a delete-registry section removes stale registry information for a previously installed device/driver from the target computer. An **HKR** specification in such a delete-

registry section designates the same subkey as for **AddReg**.

This directive is rarely used, except in an INF file that upgrades a previous installation of the same devices/models listed in the per-manufacturer per-*Models* section that defined the name of this *DDInstall* section. For more information, see [INF DelReg Directive](#).

**BitReg**=*bit-registry-section*[, *bit-registry-section*] ...

Is valid in this section, but almost never used. An HKR specification in a referenced bit-registry section designates the same subkey as for **AddReg**. For more information, see [INF BitReg Directive](#).

## Remarks

The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall.HW* section name in cross-platform INF files. For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

Any *DDInstall.HW* section must have one of the following:

- An **AddReg** directive.
- An **Include** entry that specifies another INF file. In this case, the *DDInstall.HW* section must also contain a corresponding **Needs** entry that specifies a section in the other INF file. This section is used to set up the necessary registry information.

Each directive in a *DDInstall.HW* section can reference more than one INF-writer-defined section. However, each additional named section must be separated from the next with a comma (,).

Each such section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

For more information about how to install multifunction devices, see [Supporting Multifunction Devices](#).

## Examples

This example shows how the CD-ROM device class installer uses *DDInstall.HW* sections and *DDInstall.Services* sections to support both CD audio and changer functionality by creating the appropriate registry sections, and setting these up as PnP upper filter drivers.

```

;;
;; Installation section for cdaudio. Sets cdrom as the service
;; and adds cdaudio as a PnP upper filter driver.
;;
[cdaudio_install]
CopyFiles=cdaudio_copyfiles,cdrom_copyfiles

[cdaudio_install.HW]
AddReg=nosync_addreg,cdaudio_addreg
; cdaudio_addreg required to register this as a PnP filter driver

[cdaudio_install.Services]
AddService=cdrom,0x00000002,cdrom_ServiceInstallSection
AddService=cdaudio,,cdaudio_ServiceInstallSection

[changer_install]
CopyFiles=changer_copyfiles,cdrom_copyfiles

[changer_install.HW]
AddReg=changer_addreg

; ... changer_install.Services section similar to cdaudio's

; ... some similar cdrom_install(.HW)/addreg sections omitted

[cdaudio_addreg] ; changer_addreg section has similar entry
HKR,, "UpperFilters", 0x00010000, "cdaudio" ; [REG_MULTI_SZ]
(https://docs.microsoft.com/windows/desktop/SysInfo/registry-value-types) value

;
; Use next section to disable synchronous transfers to this device.
; Sync transfers will always be turned off by default in this INF
; for any cdrom-type device.
;
[nosync_addreg]
HKR,, "DefaultRequestFlags", 0x00010001, 8

[autorun_addreg]
HKLM,"System\CurrentControlSet\Services\cdrom","AutoRun",0x00010003,1
;;
;; service-install sections for cdrom, cdaudio, and changer
;;
[cdrom_ServiceInstallSection]
DisplayName      = %cdrom_ServiceDesc%
ServiceType     = 1
StartType       = 1
ErrorControl    = 1
ServiceBinary   = %12%\cdrom.sys
LoadOrderGroup  = SCSI CDROM Class
AddReg          = autorun_addreg

[cdaudio_ServiceInstallSection]
DisplayName      = %cdaudio_ServiceDesc%
ServiceType     = 1
StartType       = 1
ErrorControl    = 1
ServiceBinary   = %12%\cdaudio.sys

; ... changer_ServiceInstallSection similar to cdaudio's

```

## See also

[AddReg](#)

[BitReg](#)

*DDInstall*

*DDInstall.Services*

*DelReg*

# INF DDInstall.Interfaces Section

11/2/2020 • 3 minutes to read • [Edit Online](#)

Each per-Models `DDInstall.Interfaces` section can have one or more [AddInterface](#) directives, depending on how many device interfaces a particular device/driver supports.

```
[install-section-name.Interfaces] |
[install-section-name.nt.Interfaces] |
[install-section-name.ntx86.Interfaces] |
[install-section-name.ntarm.Interfaces] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.Interfaces] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.Interfaces] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.Interfaces] | (Windows XP and later versions of Windows)

AddInterface={InterfaceClassGUID} [, [reference string] [, [add-interface-section] [, flags]]] ...
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
```

To support existing device interfaces, such as any of the system's predefined kernel-streaming interfaces, specify the appropriate `InterfaceClassGUID` values in this section.

To install a component, such as a class driver, that exports a new class of device interfaces, an INF must also have an [INF InterfaceInstall32 section](#).

For more information about device interfaces, see [Device Interface Classes](#).

## Entries

**AddInterface**=*{InterfaceClassGUID}* [, *[reference string]* [, *[add-interface-section]* [, *flags*]]] ...

This directive installs support for a device interface class, designated by the specified `InterfaceClassGUID` value that the driver exports to higher level components. Usually, it also references an INF-writer-defined `add-interface-section` elsewhere in the INF file. For detailed information about how to specify this directive, see [INF AddInterface Directive](#).

**Include**=*filename.inf[,filename2.inf]...*

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to register the interface classes supported by this device/driver. If this entry is specified, usually so is a **Needs** entry.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**=*inf-section-name[,inf-section-name]...*

This optional entry specifies the particular sections that must be processed during the installation of this device. Typically, such a named section is a `DDInstall.Interfaces` section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within such a `DDInstall.Interfaces` section of the included INF.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

## Remarks

The *DDInstall* section name must be referenced by a device/models-specific entry under the per-manufacturer *Models* section of the INF file. For information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions in cross-platform INF files, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

If a specified {*InterfaceClassGUID*} is not installed already, the operating system's setup code installs that device interface class in the system. If an INF file installs one or more new device interface classes, it also has an [InterfaceInstall32] section identifying the GUID for the new class..

For more information about how to create a GUID, see [Using GUIDs in Drivers](#). For the system-defined interface class GUIDs, see the appropriate system-supplied header, such as *Ks.h* for the kernel-streaming interface class GUIDS.

When a driver is loaded, it must call [IoSetDeviceInterfaceState](#) once for each {*InterfaceClassGUID*} value specified in the INF's *DDInstall.Interfaces* section that the driver supports on the underlying device, to enable the interface for run-time use by higher level components. Instead of registering support for a device interface in an INF, a device driver can call [IoRegisterDeviceInterface](#) before making its initial call to [IoSetDeviceInterfaceState](#). Usually, a PnP function or filter driver makes this call from its [AddDevice](#) routine.

## Examples

This example shows the *DDInstall.nt.Interfaces* section in the INF file for the system-supplied WDM audio device/driver shown as examples for the [INF DDInstall section](#) and the [INF DDInstall.Services section](#) .

```
; following AddInterface= are all single lines (without
; backslash line continuators) in the system-supplied INF file
;
[WDMNPB003_Device.NT.Interfaces]
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Wave%, \
    WDM_SB16.Interface.Wave
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Topo%, \
    WDM_SB16.Interface.Topology
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_UART%, \
    WDM_SB16.Interface.UART
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_FMSynth%, \
    WDM_SB16.Interface.FMSynth
; ...

[Strings] ; only immediately preceding %strkey% tokens shown here
%KSCATEGORY_AUDIO% = "{6994ad04-93ef-11d0-a3cc-00a0c9223196}"
KSNAME_Wave = "Wave"
KSNAME_UART = "UART"
KSNAME_FMSynth = "FMSynth"
KSNAME_Topo = "Topology"
; ...
```

## See also

[AddInterface](#)

[DDInstall](#)

[InterfaceInstall32](#)

[IoRegisterDeviceInterface](#)

[IoSetDeviceInterfaceState](#)

[Models](#)



# INF DDInstall.LogConfigOverride Section

11/2/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

**DDInstall.LogConfigOverride** sections are used to create an [override configuration](#), which overrides the hardware resource requirements that a Plug and Play device's bus driver reports.

```
[install-section-name.LogConfigOverride] |
[install-section-name.nt.LogConfigOverride] |
[install-section-name.ntx86.LogConfigOverride] |
[install-section-name.ntarm.LogConfigOverride] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.LogConfigOverride] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.LogConfigOverride] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.LogConfigOverride] | (Windows XP and later versions of Windows)

LogConfig=log-config-section[,log-config-section]...
```

## Entries

The section entries and values that are used with **DDInstall.LogConfigOverride** sections are specified within *log-config-sections* referenced by [INF LogConfig directives](#).

## Remarks

The configuration data that is specified in a *log-config-section* for a Plug and Play device is a preferred hardware resource configuration, but is not an absolute requirement. Some or all of the specified hardware resource configuration data might not be accepted by the device's underlying bus driver. In this situation, the device driver is assigned the hardware resources that were originally reported by the bus driver.

## Examples

The following example shows a **DDInstall.LogConfigOverride** section and a corresponding *log-config-section* for a PCMCIA device.

```
[XYZDevice.LogConfigOverride]
LogConfig = XYZDevice.Override0

[XYZDevice.Override0]
IOConfig=2f8-2ff
IOConfig=20@100-FFFF%FFE0
IRQConfig=3,4,5,7,9,10,11
MemConfig=4000@0-FFFFFFFF%FFFFC000
PcCardConfig=41:10000(W)
```

For more information about the hardware resource configuration data values that are specified in a *log-config-section*, see [INF LogConfig Directive](#).

## See also

[DDInstall](#)

# INF DDInstall.Services Section

4/29/2020 • 3 minutes to read • [Edit Online](#)

Each per-Models *DDInstall.Services* section contains one or more [INF AddService directives](#) that reference additional INF-writer-defined sections in an INF file.

```
[install-section-name.Services] |
[install-section-name.nt.Services] |
[install-section-name.ntx86.Services] |
[install-section-name.ntarm.Services] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.Services] | (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.Services] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.Services] (Windows XP and later versions of Windows)

AddService=ServiceName,[flags],service-install-section
    [,event-log-install-section[,,[EventLogType][,,EventName]]]...
[DelService=ServiceName[,,[flags][,,[EventLogType][,,EventName]]]]...
[Include=filename.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]]...
```

You can provide a *DDInstall.Services* section with at least one **AddService** directive to control how and when the services of a particular driver are loaded, dependencies on other services or drivers, and so forth.

Optionally, you can also specify event-logging services.

## Entries

**AddService**=*ServiceName*[*flags*],*service-install-section*[,*event-log-install-section*[,[*EventLogType*]  
[,,*EventName*]]]...]

This directive references an INF-writer-defined *service-install-section* and, possibly, an *event-log-install-section* elsewhere in the INF file for the drivers of the devices covered by this *DDInstall* section. For more information, see [INF AddService Directive](#).

**DelService**=*ServiceName*[,*flags*][,[*EventLogType*][,,*EventName*]]]...

This directive removes a previously installed service from the target computer. This directive is very rarely used. For more information, see [INF DelService Directive](#).

**Include**=*filename.inf*[,*filename2.inf*]...

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to install this device. If this entry is specified, usually so is a **Needs** entry.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**=*inf-section-name*[,*inf-section-name*]...

This optional entry specifies the section that must be processed during the installation of this device. Typically, the section is a *DDInstall.Services* section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within a *DDInstall.Services* section.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

## Remarks

**DDInstall.Services** sections should have the same platform and operating system decorations as their related **DDInstall** sections. For example, an *install-section-name.ntx86* section would have a corresponding *install-section-name.ntx86.Services* section.

The specified **DDInstall** section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a **DDInstall.Services** section name in cross-platform INF files.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

This example shows the **DDInstall.Services** section for the **Ser\_Inst** section shown as an example for the [INF DDInstall section](#).

```
[Ser_Inst.Services]
AddService=sermouse, 0x00000002, sermouse_Service_Inst, \
    sermouse_EventLog_Inst
;
; flags value in preceding entry indicates function driver of device
;
AddService = mouclass,, mouclass_Service_Inst, mouclass_EventLog_Inst

; entries in the following xxx_Inst sections omitted here for brevity,
; but fully specified as the example for the AddService directive
;
[sermouse_Service_Inst]
;

[sermouse_EventLog_Inst]
;

[mouclass_Service_Inst]
;

[mouclass_EventLog_Inst]
;
```

This example shows the *install-section-name.NT.Services* section and its service-install-sections in the INF file for the system-supplied WDM audio device/driver shown as an example for the [INF DDInstall section](#).

```

[WDMPNPB003_Device.NT.Services]
AddService = wdmaud,0x00000000,wdmaud_Service_Inst
AddService = swmidi,0x00000000,swmidi_Service_Inst
AddService = sb16, 0x00000002,sndblst_Service_Inst

[wdmaud_Service_Inst]
DisplayName = %wdmaud.SvcDesc% ; friendly name (see Strings)
ServiceType = 1 ; SERVICE_KERNEL_DRIVER
StartType = 1 ; SERVICE_SYSTEM_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %10%\system32\drivers\wdmaud.sys

[swmidi_Service_Inst]
DisplayName = %swmidi.SvcDesc%
ServiceType = 1
StartType = 1
ErrorControl = 1
ServiceBinary = %10%\system32\drivers\swmidi.sys

[sndblst_Service_Inst]
DisplayName = %sndblst.SvcDesc%
ServiceType = 1
StartType = 1
ErrorControl = 1
ServiceBinary = %10%\system32\drivers\mssb16.sys

[Strings] ; only immediately preceding %strkey% tokens shown here
%wdmaud.SvcDesc%="Microsoft WDM Virtual Wave Driver (WDM)"
%swmidi.SvcDesc%="Microsoft Software Synthesizer (WDM)"
%sndblst.SvcDesc%="WDM Sample Driver for SB16"

```

See [INF DDInstall.HW Section](#) for more examples of *DDInstall.Services* sections with some *service-install-* sections referenced by the [AddService](#) directive. This includes one for a PnP filter driver.

## See also

[AddService](#)

[DDInstall](#)

[DDInstall.HW](#)

[DelService](#)

[Models](#)

# INF DDInstall.Software Section

4/29/2020 • 2 minutes to read • [Edit Online](#)

Each per-Models **DDInstall\Software** section contains one or more [INF AddSoftware directives](#) that reference additional INF-writer-defined sections in a software component INF file. This section is supported for Windows 10 Version 1703 and later.

```
[install-section-name.Software] |
[install-section-name.nt.Software] |
[install-section-name.ntx86.Software] |
[install-section-name.ntia64.Software] |
[install-section-name.ntamd64.Software] |
[install-section-name.ntarm.Software] |
[install-section-name.ntarm64.Software]

AddSoftware=SoftwareName,[flags],software-install-section
```

You can provide a **DDInstall\Software** section with at least one [AddSoftware directive](#) to install software from a software component.

The software installation must be non-interactive.

## Entries

**AddSoftware**=*SoftwareName,[flags],software-install-section*

This directive references an INF-writer-defined *software-install-section* elsewhere in the software component INF file. For more information, see [INF AddSoftware directive](#).

## Remarks

**DDInstall\Software** sections should have the same platform and operating system decorations as their related **DDInstall** sections. For example, an *install-section-name.ntx86* section would have a corresponding *install-section-name.ntx86.Software* section.

The specified **DDInstall** section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a **DDInstall\Software** section name in cross-platform INF files.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

```
[ContosoCtrlPnl.NT.Software]
AddSoftware = ContosoGfx1CtrlPnl,, Software_Inst

[Software_Inst]
SoftwareType = 1
SoftwareBinary = %13%\ContosoCtrlPnl.exe
SoftwareArguments = <>DeviceInstanceID>>
SoftwareVersion = 1.0.0.0
```

## See also

[Using a Component INF File.](#)

[INF AddSoftware Directive](#)

# INF DDInstall.WMI Section

4/29/2020 • 2 minutes to read • [Edit Online](#)

An INF **DDInstall.WMI** section contains one or more **WMIInterface** directives that specify characteristics for each WMI class that the driver provides.

```
[install-section-name.WMI] |
[install-section-name.nt.WMI] |
[install-section-name.ntx86.WMI] |
[install-section-name.ntarm.WMI] | (Windows 8 and later versions of Windows)
[install-section-name.ntarm64.WMI] (Windows 10 version 1709 and later versions of Windows)
[install-section-name.ntia64.WMI] | (Windows XP and later versions of Windows)
[install-section-name.ntamd64.WMI] (Windows XP and later versions of Windows)

WMIInterface={WmiClassGUID},[flags,]WMI-class-section
```

## Entries

### *WmiClassGUID*

Specifies a GUID value that identifies a WMI class.

### *flags*

Specifies one of the following bitmask flags:

0x00000001 (SCWMI\_CLOBBER\_SECURITY)

If set, and if a security descriptor already exists in the registry, the existing security descriptor is replaced by the one specified in the INF file. If not set, and if a security descriptor already exists in the registry, the existing security descriptor is used instead of the one specified in the INF file.

### *WMI-class-section*

Specifies an INF file section that contains directives for setting characteristics of the WMI class.

The following directives can be specified within a *WMI-class-section*:

#### **Security="security-descriptor-string"**

Specifies a security descriptor that will be stored in the registry and applied to the GUID that is specified by *WmiClassGUID*. This security descriptor specifies the permissions that are required to access data blocks associated with the class. The *security-descriptor-string* value is a string with tokens that indicate the DACL (D:) security component.

Only one **Security** entry can be present. If more than one **Security** entry is present, security is not set for the WMI class.

## Remarks

The INF **DDInstall.WMI** section is available on Microsoft Windows Server 2003 and later versions of the operating system.

A security descriptor is associated with every WMI GUID. For Windows XP and earlier operating system versions, the default security descriptor for WMI GUIDs allows full access to all users. For Windows Server 2003 and later versions, the default security descriptor allows access only to administrators.

If your driver defines WMI classes, and if you do not want to use the default descriptor, include a **DDInstall.WMI**

section to specify a security descriptor that is stored in the registry and overrides the system's default descriptor.

For more information about how to specify security descriptors in INF files, see [Creating Secure Device Installations](#).

## Examples

The following example shows a single `DDInstall.WMI` section that contains two `WMIInterface` directives. Each directive identifies a WMI class and specifies a *WMI-class-section* for the class.

```
[InstallA.NT.WMI]
WMIInterface = {99999999-4cf9-11d2-ba4a-00a0c9062910},,WMISecurity1
WMIInterface = {99999998-4cf9-11d2-ba4a-00a0c9062910},1,WMISecurity2

[WmiSecurity1]
security = "0:BAG:BAD:(A;;0x120ffff;;;BA)(A;;CC;;;WD)(A;;0x120ffff;;;SY)"

[WmiSecurity2]
security = "0:BAG:BAD:(A;;0x120ffff;;;BA)(A;;CC;;;WD)(A;;0x120ffff;;;SY)"
```

## See also

[\*DDInstall\*](#)

[\*Models\*](#)

# INF DefaultInstall Section

11/2/2020 • 6 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is valid only if it has an architecture decoration, for example `[DefaultInstall.NTAMD64]`. Alternatively, use the [INF Manufacturer Section](#). Using both **DefaultInstall** and **Manufacturer** sections in your INF will cause Universal INF validation failures and can lead to inconsistent installation behaviors. See [Using a Universal INF File](#).

An INF file's **DefaultInstall** section is accessed if a user selects the "Install" menu item after selecting and holding (or right-clicking) on the INF file name.

```
[DefaultInstall] |
[DefaultInstall.nt] |
[DefaultInstall.ntx86] |
[DefaultInstall.ntarm] | (Windows 8 and later versions of Windows)
[DefaultInstall.ntarm64] | (Windows 10 version 1709 and later versions of Windows)
[DefaultInstall.ntia64] | (Windows XP and later versions of Windows)
[DefaultInstall.ntamd64] | (Windows XP and later versions of Windows)

[CopyFiles=@filename | file-list-section[,file-list-section] ...]
[CopyINF=filename1.inf[,filename2.inf]...]
[AddReg=add-registry-section[,add-registry-section]...]
[Include=filename1.inf[,filename2.inf]...]
[Needs=inf-section-name[,inf-section-name]...]
[Delfiles=file-list-section[,file-list-section]...]
[Renfiles=file-list-section[,file-list-section]...]
[DelReg=del-registry-section[,del-registry-section]...]
[BitReg=bit-registry-section[,bit-registry-section]...]
[ProfileItems=profile-items-section[,profile-items-section]...]
[UpdateInis=update-ini-section[,update-ini-section]...]
[UpdateIniFields=update-inifields-section[,update-inifields-section]...]
[Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...]
[RegisterDlls=register-dll-section[,register-dll-section]...]
[UnregisterDlls=unregister-dll-section[,unregister-dll-section]...] ...
```

## Entries

**CopyFiles=@filename | file-list-section[,file-list-section] ...**

This optional directive either specifies one named file to be copied from the source medium to the destination, or references one or more INF-writer-defined sections that specify files to be transferred from the source media to the destination.

The **DefaultDestDir** entry in the **DestinationDirs** section of the INF specifies the destination for any single file to be copied. The **SourceDiskNames** and **SourceDiskFiles** sections, or an additional INF specified in the **LayoutFile** entry of this INF's **Version** section, provide the location on the distribution media of the driver files.

For more information, see [INF CopyFiles Directive](#).

**CopyINF=filename1.inf[,filename2.inf]...**

(Windows XP and later versions of Windows.) This directive causes specified INF files to be copied to the target system.

For more information, see [INF CopyINF Directive](#).

**AddReg=add-registry-section[,add-registry-section]...**

This directive references one or more INF-writer-defined sections in which new subkeys, possibly with initial

value entries, are specified to be written into the registry or in which the value entries of existing keys are modified.

For more information, see [INF AddReg Directive](#).

**Include**= *filename1.inf*[,*filename2.inf*]...

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to install this device and/or driver. If this entry is specified, usually so is a **Needs** entry.

For example, the system INF files for device drivers that depend on the system's kernel-streaming support specify this entry as follows:

```
Include= ks.inf[,kscaptur.inf,][ksfilter.inf]]
```

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**= *inf-section-name*[,*inf-section-name*]...

This optional entry specifies sections within system-supplied INF files that must be processed during the installation of this device. Typically, such a named section is a *DDInstall* (or *DDInstall.xxx*) section within one of the INF files that are listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall* or *DDInstall.xxx* section of the included INF.

For example, the INF files for device drivers that have the preceding **Include** entry specify this entry as follows:

```
Needs= KS.Registration[,KSCAPTUR.Registration |  
KSCAPTUR.Registration.NT,MSPCLOCK.Installation]
```

**Needs** entries cannot be nested. (For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#)).

**Delfiles**= *file-list-section*[,*file-list-section*]...

This directive references one or more INF-writer-defined sections listing files on the target to be deleted.

For more information, see [INF DelFiles Directive](#).

**Renfiles**= *file-list-section*[,*file-list-section*]...

This directive references one or more INF-writer-defined sections listing files to be renamed on the destination before device-relevant source files are copied to the target computer.

For more information, see [INF RenFiles Directive](#).

**DelReg**= *del-registry-section*[,*del-registry-section*]...

This directive references one or more INF-writer-defined sections in which keys and/or value entries are specified to be removed from the registry during installation of the devices.

Typically, this directive is used to handle upgrades when an INF must clean up old registry entries from a previous installation of this device. An HKR specification in such a delete-registry section designates the *..Class\SetupClassGUID\device-instance-id* registry path of the user-accessible driver. This type of HKR specification is also referred to as a "software key".

For more information, see [INF DelReg Directive](#).

**BitReg**= *bit-registry-section*[,*bit-registry-section*]...

This directive references one or more INF-writer-defined sections in which existing registry value entries of type **REG\_BINARY** are modified. For more information, see [INF AddReg Directive](#).

An HKR specification in such a bit-registry section designates the `..Class\SetupClassGUID\device-instance-id` registry path of the user-accessible driver. This type of HKR specification is also referred to as a "software key".

For more information, see [INF BitReg Directive](#).

**ProfileItems**=*profile-items-section*[,*profile-items-section*]...

This directive references one or more INF-writer-defined sections that describe items to be added to, or removed from, the Start menu.

For more information, see [INF ProfileItems Directive](#).

**UpdateInis**=*update-ini-section*[,*update-ini-section*]...

This rarely used directive references one or more INF-writer-defined sections, specifying a source INI file from which a particular section or line within such a section is to be read into a destination INI file of the same name during installation. Optionally, line-by-line modifications to an existing INI file on the destination from a specified source INI file of the same name can be specified in the update-ini section.

For more information, see [INF UpdateInis Directive](#).

**UpdateIniFields**=*update-inifields-section*[,*update-inifields-section*]...

This rarely used directive references one or more INF-writer-defined sections in which modifications within the lines of a device-specific INI file are specified.

For more information, see [INF UpdateIniFields Directive](#).

**Ini2Reg**=*ini-to-registry-section*[,*ini-to-registry-section*]...

This rarely used directive references one or more INF-writer-defined sections in which sections or lines from a device-specific INI file, supplied on the source media, are to be moved into the registry.

For more information, see [INF Ini2Reg Directive](#).

**RegisterDlls**=*register-dll-section*[,*register-dll-section*]...

This directive references one or more INF sections used to specify files that are OLE controls and require self-registration.

For more information, see [INF RegisterDlls Directive](#).

**UnregisterDlls**=*unregister-dll-section*[,*unregister-dll-section*]...

This directive references one or more INF sections used to specify files that are OLE controls and require self-unregistration (self-removal).

For more information, see [INF UnregisterDlls Directive](#).

## Remarks

**DefaultInstall** sections must not be used for device installations. Use **DefaultInstall** sections only for the installation of class filter drivers, class co-installers, file system filters, and kernel driver services that are not associated with a device node (*devnode*).

**Note** The INF file of a [driver package](#) must not contain an INF **DefaultInstall** section if the driver package is to be digitally signed. For more information about signing driver packages, see [Driver Signing](#).

Providing a **DefaultInstall** section is optional. If an INF file does not include a **DefaultInstall** section, selecting "Install" after selecting and holding (or right-clicking) on the file name causes an error message to be displayed.

**Note** Unlike a **DDInstall** section, a **DefaultInstall** section cannot contain **DriverVer** or **LogConfig** directives.

To install a **DefaultInstall** section from a *device installation application*, use the following call to **InstallHinfSection**:

```
InstallHinfSection(NULL,NULL,TEXT("DefaultInstall 132 path-to-inf\infname.inf"),0);
```

For more information about **InstallHinfSection**, see the Microsoft Windows SDK documentation.

For more information about how to use the system-defined **.nt**, **.ntx86**, **.ntia64**, **.ntamd64**, **.ntarm**, and **.ntarm64** extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

The following example shows a typical **DefaultInstall** section:

```
[DefaultInstall]
CopyFiles=MyAppWinFiles, MyAppSysFiles, @SRSutil.exe
AddReg=MyAppRegEntries
```

In this example, the **DefaultInstall** section is executed if a user selects "Install" after selecting and holding (or right-clicking) on the INF file name.

## See also

[\*\*DDInstall\*\*](#)

[\*\*DriverVer\*\*](#)

[\*\*LogConfig\*\*](#)

# INF DefaultInstall.Services Section

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this section is not valid. See [Using a Universal INF File](#).

A **DefaultInstall.Services** section contains one or more **AddService** directives referencing additional INF-writer-defined sections in an INF file. This section is equivalent to the [INF DDInstall.Services](#) section, and is used in association with an [INF DefaultInstall](#) section.

```
[DefaultInstall.Services] |
[DefaultInstall.nt.Services] |
[DefaultInstall.ntx86.Services] |
[DefaultInstall.ntarm.Services] | (Windows 8 and later versions of Windows)
[DefaultInstall.ntarm64.Services] (Windows 10 version 1709 and later versions of Windows)
[DefaultInstall.ntia64.Services] | (Windows XP and later versions of Windows)
[DefaultInstall.ntamd64.Services] (Windows XP and later versions of Windows)

AddService=ServiceName,[flags],service-install-section
    [,event-log-install-section[,,[EventLogType][,EventName]]]...
[DelService=ServiceName[,,[flags][,[EventLogType][,EventName]]]]...
[Include=filename.inf[,filename2.inf]]...
[Needs=inf-section-name[,inf-section-name]]...
```

## Entries

**AddService**=*ServiceName*,[*flags*],*service-install-section*

,*event-log-install-section*[,*EventLogType*[,*EventName*]...]...

This directive references an INF-writer-defined *service-install-section* and, possibly, an *event-log-install-section* elsewhere in the INF file for the drivers of the devices covered by this **DefaultInstall** section.

For more information, see [INF AddService Directive](#).

**DelService**=*ServiceName*[,[*flags*][,[*EventLogType*][,*EventName*]]]...

This directive removes a previously installed service from the target computer. This directive is very rarely used.

For more information, see [INF DelService Directive](#).

**Include**=*filename.inf*[,*filename2.inf*]...

This optional entry specifies one or more additional system-supplied INF files that contain sections needed to install this device. If this entry is specified, usually so is a **Needs** entry.

For more information about the **Include** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

**Needs**=*inf-section-name*[,*inf-section-name*]...

This optional entry specifies the particular named section that must be processed during the installation of this device. Typically, such a named section is a *DDInstall.Services* section within a system-supplied INF file that is listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall.Services* section.

**Needs** entries cannot be nested. For more information about the **Needs** entry and restrictions on its use, see [Specifying the Source and Target Locations for Device Files](#).

## Remarks

The [AddService](#) directive controls how and when the services of a particular driver are loaded, any dependencies on other services or on underlying (legacy) drivers it might have, and so forth. Optionally, it can set up event-logging services for the driver as well.

**Note** INF files use the **DefaultInstall.Services** section only if they also use an [INF DefaultInstall](#) section. Otherwise, they use [INF DDInstall.Services](#) sections together with [INF DDInstall](#) sections.

**DefaultInstall.Services** sections should have the same platform and operating system decorations as their related **DefaultInstall** sections. For example, a **DefaultInstall.ntx86** section would have a corresponding **DefaultInstall.ntx86.Services** section. For more information about how to use the system-defined **.nt**, **.ntx86**, **.ntia64**, **.ntamd64**, **.ntarm**, and **.ntarm64** extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Examples

See the examples provided for the [INF DDInstall.Services](#) section.

## See also

[DDInstall](#)

[DefaultInstall](#)

# INF DestinationDirs Section

11/2/2020 • 3 minutes to read • [Edit Online](#)

A **DestinationDirs** section specifies the target destination directory or directories for all copy, delete, and/or rename operations on files referenced by name elsewhere in the INF file.

```
[DestinationDirs]  
[DefaultDestDir=dirid[,subdir]]  
[file-list-section=dirid[,subdir]]...
```

## Entries

### **DefaultDestDir** = *dirid[,subdir]*

Specifies the default destination directory for all copy, delete, and/or rename operations on files that are not explicitly listed in a *file-list-section* referenced by other entries here. To ensure that file operations always occur in the correct directory, an INF file that includes **Include** and **Needs** entries should not specify a default destination directory. For more information, see the following Remarks section.

### *file-list-section* = *dirid[,subdir]* ...

Specifies the INF-writer-determined name of a section referenced by a **CopyFiles**, **RenFiles**, or **DelFiles** directive elsewhere in the INF file. Such an entry is optional if this section has a **DefaultDestDir** entry and all copy-file operations specified in this INF have the same target destination. However, any *file-list-section* referenced by a **RenFiles** or **DelFiles** directive elsewhere in the INF must be listed here.

### *dirid*

Specifies the directory identifier of the target directory for operations on files that are referenced by name, possibly within a named *file-list-section* of the INF. For lists of commonly used *dirids*, see [Using Dirids](#).

### *subdir*

Specifies the subdirectory (and the rest of its path, if any, under the directory identified by *dirid*) to be the destination of the file operations in the given *file-list-section*.

## Remarks

The **DestinationDirs** section is required in any INF file that uses an **INF CopyFiles directive** or that references a *file-list-section*, whether with a **CopyFiles**, **DelFiles**, or **RenFiles** directive.

If *Abc.inf* includes sections from another INF file, *Def.inf*, and both INF files include a **DefaultDestDir** entry for copy-file, rename-file, or delete-file operations, Windows ignores the default destination directory that is specified in *Def.inf* and performs all the corresponding file operations in the default destination directory that is specified in *Abc.inf*.

To ensure that file operations always occur in the correct directories, an INF file that includes **Include** and **Needs** entries should not include a **DefaultDestDir** entry in a **DestinationDirs** section. Instead, such an INF file should explicitly reference all the *file-list-section* names that are specified by **CopyFiles**, **RenFiles**, and **DelFiles** directives in the **DestinationDirs** section.

If an INF file does not include **Include** and **Needs** entries, the INF can use the **DefaultDestDir** entry to specify a default destination for copy, rename, and delete file operations that appear elsewhere in the INF file:

- **CopyFiles** directives that use the direct copy (@*filename*) notation must have a **DefaultDestDir** entry in

the **DestinationDirs** section of the INF in which the direct-copy entry appears.

- **CopyFiles**, **RenFiles**, or **DelFiles** sections that are not directly referenced in the **DestinationDirs** section must have a **DefaultDestDir** entry in the **DestinationDirs** section of the INF in which the copy, rename, and delete file sections appear.

## Examples

This example sets the default target directory for all copy-file, delete-file, and rename-file operations. Such a simple **DestinationDirs** section is common to INF files for new peripheral devices, because such an INF usually just copies a set of source files into a single directory on the target computer.

```
[DestinationDirs]
DefaultDestDir = 12 ; dirid = \Drivers on WinNT platforms
```

This example shows a fragment of the **DestinationDirs** section of the INF for display/video drivers.

```
[DestinationDirs]
DefaultDestDir      = 11 ; dirid = \system32 on WinNT platforms

; ...

; list of per-Manufacturer, per-Models, per-DDInstall-section, and
; CopyFiles-referenced xxx.Miniport/xxx.Display sections omitted here
; along with several other miniport/display paired drivers
; ...
vga.Miniport      = 12
vga.Display       = 11
xga.Miniport      = 12
xga.Display       = 11

; all video miniports copied into \system32\drivers on WinNT platforms
; all paired display drivers copied into \system32
```

## See also

[ClassInstall32](#)

[CopyFiles](#)

[DDInstall](#)

[DelFiles](#)

[RenFiles](#)

[SourceDiskFiles](#)

[SourceDiskNames](#)

[Using Dirids](#)

[Version](#)

# INF InterfaceInstall32 Section

11/2/2020 • 3 minutes to read • [Edit Online](#)

This section creates one or more new [device interface classes](#). After a new class is created, subsequently installed devices/drivers can be registered to support the new device interface class by using [INF DDInstall.Interfaces sections](#) in their respective INF files, or by calling [IoRegisterDeviceInterface](#).

```
[InterfaceInstall32]  
  
{InterfaceClassGUID}=install-interface-section[,flags]  
...
```

## Entries

### *InterfaceClassGUID*

Specifies a GUID value identifying the newly exported [device interface class](#).

To register an instance of the interface class, a specified GUID value in this section must be referenced by an [INF AddInterface directive](#) in an [INF DDInstall.Interfaces section](#), or else the newly installed device's driver must call [IoRegisterDeviceInterface](#) with this GUID.

For more information about how to create a GUID, see [Using GUIDs in Drivers](#). For the system-defined interface class GUIDS, see the appropriate headers, such as *Ks.h* for the kernel-streaming interfaces.

### *install-interface-section*

References an INF-writer-defined section, possibly with any of the system-defined extensions, elsewhere in this INF.

### *flags*

If specified, this entry must be zero.

## Remarks

When a specified *InterfaceClassGUID* is not already installed in the system, that interface class is installed as the corresponding *DDInstall.Interfaces* section is processed by the [SetupAPI](#) functions during device installation or when that device's driver makes the initial call to [IoRegisterDeviceInterface](#).

Each *install-interface-section* name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

Any specified *install-interface-section* has the following general form:

```

[interface-install-section] |
[interface-install-section.nt] |
[interface-install-section.ntx86] |
[interface-install-section.ntia64] | (Windows XP and later versions of Windows)
[interface-install-section.ntamd64] (Windows XP and later versions of Windows)
[interface-install-section.ntarm] (Windows 8 and later versions of Windows)
[interface-install-section.ntarm64] (Windows 10 and later versions of Windows)

AddReg=add-registry-section[, add-registry-section] ...
[AddProperty=add-property-section[, add-property-section] ...] (Windows Vista and later versions of Windows)
[Copyfiles=@filename | file-list-section[, file-list-section] ...]
[DelReg=del-registry-section[, del-registry-section] ...]
[DelProperty=del-property-section[, del-property-section] ...] (Windows Vista and later versions of Windows)
[BitReg=bit-registry-section[,bit-registry-section]...]
[Delfiles=file-list section[, file-list-section] ...]
[Renfiles=file-list-section[, file-list-section] ...]
[UpdateInis=update-ini-section[,update-ini-section]...]
[UpdateIniFields=update-inifields-section[,update-inifields-section]...]
[Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...]
...

```

For more information about the entries in the *interface-install-section*, see [INF DDInstall Section](#).

Starting with Windows Vista, you can set [device interface class](#) properties by including [INF AddProperty directives](#) in an interface-install section. You can also delete device interface class properties by including [INF DelProperty directives](#) in an interface-install section. However, you should use an [AddProperty](#) or [DelProperty](#) directive only to modify device interface class properties that are new to Windows Vista or later versions of Windows operating systems. For device interface class properties that were introduced on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry value entries, you should continue to use [INF AddReg directives](#) and [INF DelReg directives](#) to set and delete the device interface class properties. These guidelines apply to system-defined properties and custom properties. For more information about how to use the [AddProperty](#) directive and [DelProperty](#) directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

An [AddReg](#) directive references one or more add-registry sections that set device-interface-specific information in the registry during installation of this interface. An HKR specified in such an add-registry section designates the `..DeviceClasses\{InterfaceClassGUID}` key.

The registry information about this interface class should include at least a friendly name for the new [device interface class](#) and whatever information the higher level components need when they open and use this interface.

In addition, such an *install-interface-section* might use any of the optional directives shown here to specify interface-specific installation operations.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## See also

[AddProperty](#)

[AddReg](#)

[BitReg](#)

[ClassInstall32](#)

[CopyFiles](#)

*[DDInstall](#)*

*[DDInstall.Interfaces](#)*

[DelFiles](#)

[DelProperty](#)

[DelReg](#)

[Ini2Reg](#)

[IoRegisterDeviceInterface](#)

[RenFiles](#)

[UpdateIniFields](#)

[UpdateInis](#)

# INF Manufacturer Section

11/2/2020 • 10 minutes to read • [Edit Online](#)

The **Manufacturer** section identifies the manufacturer of one or more devices that can be installed by using the INF file.

```
[Manufacturer]  
  
manufacturer-identifier  
[manufacturer-identifier]  
[manufacturer-identifier]  
...
```

## Entries

### *manufacturer-identifier*

Uniquely identifies a manufacturer and an INF section that contains information that identifies a manufacturer's device models. Each *manufacturer-identifier* entry must exist on a separate line and use the following format:

```
manufacturer-name |  
%strkey%=models-section-name |  
%strkey%=models-section-name [,TargetOSVersion] [,TargetOSVersion] ... (Windows XP and later versions of  
Windows)
```

These entries are defined as follows:

### *manufacturer-name*

Identifies the devices' manufacturer. The INF must also contain a corresponding **INF Models section** of the same name. The maximum length of a manufacturer's name, in characters, is LINE\_LEN. (An entry specified in this manner cannot be localized.)

### *strkey*

Specifies a token, unique within the INF file that represents the name of a manufacturer. Each such %*strkey*% token must be defined in an **INF Strings section** of the INF file.

### *models-section-name*

Specifies an INF-writer-defined name for the per-manufacturer **INF Models section** within the INF file. This value must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

### *TargetOSVersion*

Specifies one or more target operating system versions with which various **INF Models** sections can be used. Windows chooses the **INF Models** section that most closely matches the operating system version on which it is executing.

For a description of the *TargetOSVersion* decoration, see the following **Remarks** section, and related info in Example 3 below.

**Important:** Starting with Windows Server 2003 SP1, INF files must decorate models-section-name entries in the **INF Manufacturer section**, as well as the associated **INF Models** section names, with .ntia64 or .ntamd64 platform extensions to specify non-x86 target operating system versions. These platform extensions are not required in INF files for x86-based target operating system versions or non-PnP driver INF files, such as file

system driver INF files for x64-based architectures.

## Remarks

Any INF file that installs one or more devices must have a **Manufacturer** section. An IHV/OEM-supplied INF file typically specifies only a single entry in this section. If multiple entries are specified, each entry must be on a separate line of the INF.

Using a `%strkey% = models-section-name` entry simplifies the localization of the INF file for the international market, as described in [Creating International INF Files](#) and the reference page for the [INF Strings section](#).

If an INF file specifies one or more entries in the *manufacturer-name* format, each such entry implicitly specifies the name of the corresponding **Models** section elsewhere in the INF.

You can think of each system-supplied INF file's **Manufacturer** section as a table of contents, because this section sets up the installation of every manufacturer's device models for a [device setup class](#). Each entry in an INF file's **Manufacturer** section specifies both an easily localizable `%strkey%` token for the name of a manufacturer and a unique-to-the-INF per-manufacturer **Models** section name.

The *models-section-name* entries in the **Manufacturer** section can be decorated to specify target operating system versions. Different [INF Models sections](#) can be specified for different versions of the operating system. The specified versions indicate operating system versions with which the INF **Models** sections is used. If no versions are specified, Windows uses a specified **Models** section for all versions of all operating systems.

For Windows XP to Windows 10, version 1511, the format of *TargetOSVersion* decoration is as follows:

```
nt[Architecture][.[OSMajorVersion][.[OSMinorVersion][.[ProductType][.[SuiteMask]]]]]
```

Starting with Windows 10, version 1607 (Build 14310 and later), the format of the *TargetOSVersion* decoration is as follows:

```
nt[Architecture][.[OSMajorVersion][.[OSMinorVersion][.[ProductType][.[SuiteMask][.[BuildNumber]]]]]]
```

Each field is defined as follows:

**nt**

Specifies the target operating system is NT-based. Windows 2000 and later versions of Windows are all NT-based.

*Architecture*

Identifies the hardware platform. If specified, this must be **x86**, **ia64**, **amd64**, **arm**, or **arm64**.

Prior to Windows Server 2003 SP1, if *Architecture* is not specified, the associated INF **Models** section can be used with any hardware platform.

Starting with Windows Server 2003 SP1, *Architecture* must be specified in [INF Models sections](#) names for non-x86 target operating system versions. *Architecture* is optional in INF **Models** section names for x86-based target operating system versions.

*OSMajorVersion*

A number that represents the operating system's major version number. The following table defines the major version for the Windows operating system.

WINDOWS VERSION	MAJOR VERSION
Windows 10	10
Windows Server 2012 R2	6
Windows 8.1	6
Windows Server 2012	6
Windows 8	6
Windows Server 2008 R2	6
Windows 7	6
Windows Server 2008	6
Windows Vista	6
Windows Server 2003 R2	5
Windows Server 2003	5
Windows XP	5
Windows 2000	5

#### *OSMinorVersion*

A number that represents the operating system's minor version number. The following table defines the minor version for the Windows operating system.

WINDOWS VERSION	MINOR VERSION
Windows 10	0
Windows Server 2012 R2	3
Windows 8.1	3
Windows Server 2012	2
Windows 8	2
Windows Server 2008 R2	1
Windows 7	1
Windows Server 2008	0
Windows Vista	0

WINDOWS VERSION	MINOR VERSION
Windows Server 2003 R2	2
Windows Server 2003	2
Windows XP	1
Windows 2000	0

#### *ProductType*

A number that represents one of the VER\_NT\_xxxx flags defined in *Winnth*, such as the following:

0x00000001 (VER\_NT\_WORKSTATION)

0x00000002 (VER\_NT\_DOMAIN\_CONTROLLER)

0x00000003 (VER\_NT\_SERVER)

If a product type is specified, the INF file is used only if the operating system matches the specified product type. If the INF supports multiple product types for a single operating system version, multiple *TargetOSVersion* entries are required.

#### *SuiteMask*

A number representing a combination of one or more of the VER\_SUITE\_xxxx flags defined in *Winnth*. These flags include the following:

0x00000001 (VER\_SUITE\_SMALLBUSINESS)

0x00000002 (VER\_SUITE\_ENTERPRISE)

0x00000004 (VER\_SUITE\_BACKOFFICE)

0x00000008 (VER\_SUITE\_COMMUNICATIONS)

0x00000010 (VER\_SUITE\_TERMINAL)

0x00000020 (VER\_SUITE\_SMALLBUSINESS\_RESTRICTED)

0x00000040 (VER\_SUITE\_EMBEDDEDNT)

0x00000080 (VER\_SUITE\_DATACENTER)

0x00000100 (VER\_SUITE\_SINGLEUSERTS)

0x00000200 (VER\_SUITE\_PERSONAL)

0x00000400 (VER\_SUITE\_SERVERAPPLIANCE)

If one or more suite mask values are specified, the INF is used only if the operating system matches all the specified product suites. If the INF supports multiple product suite combinations for a single operating system version, multiple *TargetOSVersion* entries are required.

#### *BuildNumber*

A number that represents the minimum OS build number of the Windows 10 release to which the section is applicable, starting with build 14310 or later.

The build number is assumed to be relative to some specific OS major/minor version only, and may be reset for some future OS major/minor version. Any build number specified by the *TargetOSVersion* decoration is evaluated only when the OS major/minor version of the *TargetOSVersion* matches the current OS (or

`AltPlatformInfo`) version exactly. If the current OS version is greater than the OS version specified by the `TargetOSVersion` decoration (`OSMajorVersion,OSMinorVersion`), the section is considered applicable regardless of the build number specified. Likewise, if the current OS version is less than the OS version specified by `TargetOSVersion` decoration, the section is not applicable.

If build number is supplied, the OS version and BuildNumber of the `TargetOSVersion` decoration must both be greater than the OS version and build number of the Windows 10 build 14310 where this decoration was first introduced. Earlier versions of the operating system without these changes (for example, Windows 10 build 10240) will not parse unknown decorations, so an attempt to target these earlier builds will actually prevent that OS from considering the decoration valid at all.

For more information about the `TargetOSVersion` decoration, see [Combining Platform Extensions with Operating System Versions](#).

**Important** We highly recommend that you always decorate `models-section-name` entries in the `Manufacturer` and `Models` sections with platform extensions for target operating systems of Windows XP or later versions of Windows. For x86-based hardware platforms, you should avoid the use of the `.nt` platform extension and use `.ntx86` instead.

If your INF contains `Manufacturer` section entries with decorations, it must also include [INF Models sections](#) with names that match the operating system decorations. For example, if an INF contains the following `Manufacturer` section:

```
%FooCorp% = FooMfg, NTx86....0x80, NTamd64
```

Then the INF must also contain [INF Models sections](#) with the following names:

- [FooMfg.NTx86....0x80]

This name applies to the Data Center suite of Windows XP and later versions of Windows on x86-based hardware platforms.

- [FooMfg.NTamd64]

This name applies to all product types and suites of Windows XP and later versions of Windows on x64-based hardware platforms.

During installation, Windows selects an [INF Models section](#) in the following way:

1. If Windows is running in an x86-based version of the operating system (Windows XP or later versions) that includes the Data Center product suite, Windows selects the [FooMfg.NTx86....0x80] `Models` section.
2. If Windows is running in an x64-based version of the operating system (Windows XP or later versions) for any product suite, Windows selects the [FooMfg.NTamd64] `Models` section.

If the INF is intended for use with operating system versions earlier than Windows XP, it must also contain an undecorated `Models` section named [FooMfg].

If an INF supports multiple manufacturers, these rules must be followed for each manufacturer.

The following are additional examples of `TargetOSVersion` decorations:

- %FooCorp% = FooMfg, NTx86

In this example, the resultant INF `Models` section name is [FooMfg.NTx86], and is applicable for any x86 version of the operating system (Windows XP or later).

- %FooCorp% = FooMfg, NT.7.8

In this example, for version 7.8 and later of the operating system, the resultant INF `Models` section name is [FooMfg.NT.7.8]. For earlier versions of the operating system such as Windows XP,

[FooMfg.NT] is used.

Setup's selection of which INF *Models* section to use is based on the following rules:

- If the INF contains **INF Models sections** for several major or minor operating system version numbers, Windows uses the section with the highest version numbers that are not higher than the operating system version on which the installation is taking place.
- If the INF *Models* sections that match the operating system version also include product type and/or product suite decorations, Windows selects the section that most closely matches the running operating system.

Suppose, for example, Windows is executing on Windows XP (version 5.1), without the Data Center product suite, and it finds the following entry in a **Manufacturer** section:

```
%FooCorp% = FooMfg, NT, NT.5, NT.5.5, NT....0x80
```

In this case, Windows looks for an **INF Models section** named [FooMfg.NT.5]. Windows also uses the [FooMfg.NT.5] section if it is executing on a Datacenter version of Windows XP, because a specific version number takes precedence over the product type and suite mask.

If you want an INF to explicitly exclude a specific operating system version, product type, or suite, create an empty **INF Models section**. For example, an empty section named [FooMfg.NTx86.6.0] prohibits installation on x86-based operating system versions 6.0 and higher.

## Examples

This example shows a **Manufacturer** section typical to an INF for a single IHV.

```
[Manufacturer]
%Mfg% = Contoso ; Models section == Contoso

[Contoso]

; ...

[Strings]
Mfg = "Contoso, Ltd."
```

The next example shows part of a **Manufacturer** section typical to an INF for a device-class-specific installer:

```
[Manufacturer]
%CONTOSO% = Contoso_Section
; several entries omitted here for brevity
%FABRIKAM% = Fabrikam_Section
%ADATUM% = Adatum_Section
```

The following example shows a **Manufacturer** section that is specific to x86 platforms, Windows XP and later:

```
[Manufacturer]
%foo% = foosec,NTx86.5.1

[foosec.NTx86.5.1]
```

The following example shows a **Manufacturer** section that is specific to x64 platforms, Windows 10 build 14393 and later:

```
[Manufacturer]
%foo%=foosec,NTamd64.10.0...14393

[foosec.NTamd64.10.0...14393]
```

The following two examples show skeletal INF files with a variety of OS-specific INF *Models* sections:

Example 1:

```
[Manufacturer]
%MyName% = MyName,NTx86.5.1

[MyName]
%MyDev% = InstallA,hwid

[MyName.NTx86.5.1]
%MyDev% = InstallB,hwid

[InstallA] ; Windows 2000

[InstallB] ; Windows XP and later, x86 only
```

Example 2:

```
[Manufacturer]
%MyName% = MyName,NTx86.6.0,NTx86.5.1,
.

[MyName.NTx86.6.0] ; Empty section, so this INF does not support
; NT 6.0 and later.

[MyName.NTx86.5.1] ; Used for NT 5.1 and later
; (but not NT 6.0 and later due to the NTx86.6.0 entry)
%MyDev% = InstallB,hwid

[MyName] ; Empty section, so this INF does not support
; Win2000
```

Example 3:

```
[Manufacturer]
%MyMfg% = MyMfg, NTamd64.6.1, NTamd64.10.0, NTamd64.10.0...14310

[MyMfg.NTamd64.6.1] ; Used for Windows 7 and later
; (but not for Windows 10 and later due to the NT.10.0 entry)

[MyMfg.NTamd64.10.0] ; Used for Windows 10
; (but not for Windows 10 build 14393 and later due to the NT.10.0...14393
entry)

[MyMfg.NTamd64.10.0...14393] ; Used for Windows 10 build 14393 and later
```

**Note:** When specifying multiple TargetOSVersions, string them together in one entry as seen in this example.  
Do not represent each target as a separate entry.

## See also

[Combining Platform Extensions with Operating System Versions](#)

[\*Models\*](#)

[\*\*Strings\*\*](#)

# INF Models Section

4/29/2020 • 4 minutes to read • [Edit Online](#)

A per-manufacturer *Models* section identifies at least one device, references the *DDInstall* section of the INF file for that device, and specifies a unique-to-the-model-section [hardware identifier \(ID\)](#) for that device.

Any entry in the per-manufacturer *Models* section can also specify one or more additional device IDs for models that are compatible with the device designated by the initial hardware ID and are controlled by the same drivers.

```
[models-section-name] |
[models-section-name.TargetOSVersion] (Windows XP and later versions of Windows)

device-description=install-section-name,[hw-id][,compatible-id...]
[device-description=install-section-name,[hw-id][,compatible-id]...] ...
```

## NOTE

INFs are required to specify at least one device ID for each entry in the models section. This may be either a hardware ID or compatible ID.

## Entries

### *device-description*

Identifies a device to be installed, expressed as any unique combination of visible characters or as a %*strkey*% token defined in an [INF Strings section](#). The maximum length, in characters, of a device description is LINE\_LEN.

### *install-section-name*

Specifies the undecorated name of the INF install sections to be used for the device (and compatible models of device, if any). For more information, see [INF DDInstall Section](#).

### *hw-id*

Specifies a vendor-defined [hardware ID](#) string that identifies a device, which the PnP manager uses to find an INF-file match for this device. Such a hardware ID has one of the following formats:

### *enumerator\enumerator-specific-device-id*

Is the typical format for individual PnP devices reported to the PnP manager by a single enumerator. For example, `USB\VID_045E&PID_00B` identifies the Microsoft HID keyboard device on a USB bus. Depending on the enumerator, such a specification can even include the device's hardware revision number as, for example, `PCI\VEN_1011&DEV_002&SUBSYS_00000000&REV_02`.

### *\*enumerator-specific-device-id*

Indicates with the asterisk (\*) that the device is supported by more than one enumerator. For example, `*PNP0F01` identifies the Microsoft serial mouse, which also has a compatible-id specification of `SERENUM\PNP0F01`.

### *device-class-specific-ID*

Is an I/O bus-specific format, as described in the hardware specification for the bus, for the

hardware IDs of all peripheral devices on that type of I/O bus.

Be aware that a single device can have more than one *hw-id* value. The PnP manager uses each such *hw-id* value, which is usually provided by the underlying bus when it enumerates its child devices, to create a subkey for each such device in the registry **Enum** branch. For manually installed devices, the system's setup code uses their *hw-id* values as specified in their respective INF files to create each such registry subkey.

#### *compatible-id*

Specifies a vendor-defined [compatible ID](#) string that identifies compatible devices. Any number of *compatible-id* values can be specified for an entry in the *Models* section, each separated from the next by a comma (,). All such compatible devices and/or device models are controlled by the same driver as the device designated by the initial *hw-id*.

## Remarks

Each *models-section-name* must be listed in the [INF Manufacturer section](#) of the INF file.

There can be one or more entries in any per-manufacturer *Models* section, depending on how many devices (and drivers) the INF file installs for a particular manufacturer.

Each *install-section-name* must be unique within the INF file and must follow the general rules for defining section names, described in [General Syntax Rules for INF Files](#). The [\*\*DDInstall\*\*](#) section name referenced in a per-manufacturer *Models* section can also have extensions appended to the given *install-section-name*, thus defining additional [\*\*DDInstall\*\*](#) sections for the OS-specific or platform-specific installation of the given devices. For more information about how to use extensions in cross-platform system files, see also [Creating an INF File](#).

Any specified *hw-id* or *compatible-id* value can also be specified in the [INF ControlFlags section](#) to prevent that device from being displayed to the end-user during manual installations. For more information about *hw-id* and *compatible-id* values, see [Device Identification Strings](#).

For each device and driver that is installed by using an INF file, the device installers use the information supplied in the [INF Manufacturer section](#) and per-manufacturer *Models* sections to generate Device Description, Manufacturer Name, Device ID (if the installation is manual), and, possibly, Compatibility List value entries in the registry.

A *models section name* can include a *TargetOSVersion* decoration. For more information about this decoration, see [INF Manufacturer Section](#), specifically the Remarks section.

**Important** Starting with Windows Server 2003 SP1, INF files must decorate *models-section-name* entries in the [INF Manufacturer section](#), along with the associated INF *Models* section names, with **.ntia64** or **.ntamd64** platform extensions to specify non-x86 target operating system versions. These platform extensions are not required in INF files for x86-based target operating system versions or non-PnP driver INF files (such as file system driver INF files for x64-based architectures). Each entry in a *Models* section is sometimes called a *driver node*.

## Examples

This example shows a per-manufacturer *Models* section with some representative entries from the system mouse class installer's INF file, defining the [\*\*DDInstall\*\*](#) sections for some devices/models.

```
[Manufacturer]
%StdMfg%      =StdMfg          ; (Standard types)
%MSMfg%      =MSMfg          ; Microsoft
; ... %otherMfg% omitted here

[StdMfg] ; per-Manufacturer Models section
; Std serial mouse
%*pnp0f0c.DeviceDesc%= Ser_Inst,*PNP0F0C,SERENUM\PNP0F0C,SERIAL_MOUSE
; Std InPort mouse
%*pnp0f0d.DeviceDesc%      = Inp_Inst,*PNP0F0D
; ... more StdMfg entries
```

## See also

[ControlFlags](#)

*[DDInstall](#)*

[Manufacturer](#)

[Strings](#)

# INF SignatureAttributes Section

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section allows users to request additional signatures as required by certain certification scenarios. For example, the following scenarios require this section: Protected Environment media playback, [Early Launch Antimalware](#), and third party HAL extensions. These additional signatures will only be applied if your Hardware Certification Kit package contains the proper Features and passing Tests.

```
[SignatureAttributes]
FileOne = SignatureAttributes.SigType

[SignatureAttributes.SigType]
Attribute = Value
```

## Entries

**SigType**=*signature-type*

Defines which signature or catalog attribute needs to be applied to the file. Should be one of the following:

- Elam
- HalExt
- PETrust
- DRM
- WindowsHello

**Attribute**=*attribute-name*

Each Signature Type has a corresponding attribute and value, as listed below. Use these definitions for your SignatureAttributes subsections:

- **SignatureAttributes.Elam**: Elam = true
- **SignatureAttributes.HalExt**: HalExt = true
- **SignatureAttributes.DRM**: DRMLevel = {1300 | 1200}
- **SignatureAttributes.PETrust**: PETrust = true
- **SignatureAttributes.WindowsHello**: WindowsHello = true

## Remarks

These additional signatures will only be applied if your Hardware Certification Kit package contains the proper Features and passing Tests. These are additions to the normal behavior of Hardware Certification, and the corresponding Certification Requirements for Elam, HalExt, PETrust, and DRM. For more info, see [Windows Hardware Lab Kit](#).

These INF sections should be used when requesting additional signatures regardless of the target OS.

## Examples

The following examples demonstrate how to enumerate and request additional signatures for audio:

```
[SignatureAttributes]
ExampleFile1.dll=SignatureAttributes.PETrust
ExampleFile2.dll=SignatureAttributes.DRM

[SignatureAttributes.DRM]
DRMLevel=1300

[SignatureAttributes.PETrust]
PETrust=true
```

The following examples demonstrate how to enumerate and request additional signatures for video:

```
[SignatureAttributes]
ExampleFile1.dll=SignatureAttributes.PETrust

[SignatureAttributes.PETrust]
PETrust=true
```

The following examples demonstrate how to enumerate and request additional signatures for HAL:

```
[SignatureAttributes]
HALFILE.dll=SignatureAttributes.HalExt

[SignatureAttributes.HalExt]
HalExt=true
```

The following examples demonstrate how to enumerate and request additional signatures for ELAM:

```
[SignatureAttributes]
ELAMFILE.dll=SignatureAttributes.Elam

[SignatureAttributes.Elam]
Elam=true
```

The following examples demonstrate how to enumerate and request additional signatures for Windows Hello:

```
[SignatureAttributes]
WindowsHelloFile.dll=SignatureAttributes.WindowsHello

[SignatureAttributes.WindowsHello]
WindowsHello=true
```

## See also

[Dashboard Help](#)

# INF SourceDisksFiles Section

4/29/2020 • 2 minutes to read • [Edit Online](#)

The **SourceDisksFiles** section names the source files that are used during installation, identifies the installation disks that contain those files, and provides the directory paths, if any, on the distribution disks that contain individual files.

In order for a driver file or an application file to be included as part of a signed [driver package](#), the file must have a corresponding INF **SourceDisksFiles** section entry and a corresponding [INF CopyFiles directive](#).

```
[SourceDisksFiles] |
[SourceDisksFiles.x86] |
[SourceDisksFiles.arm] | (Windows 8 and later versions of Windows)
[SourceDisksFiles.arm64] | (Windows 10 version 1709 and later versions of Windows)
[SourceDisksFiles.ia64] | (Windows XP and later versions of Windows)
[SourceDisksFiles.amd64] (Windows XP and later versions of Windows)

filename=diskid[, [ subdir][,size]]
...
```

## Entries

### *filename*

Specifies the name of the file on the source disk.

### *diskid*

Specifies the integer identifying the source disk that contains the file. This value, along with the initial *subdir*(ectory) path (if any) that contains the named file, must be defined in a [SourceDisksNames](#) section of the same INF.

### *subdir*

This optional value specifies the subdirectory (relative to the *path* value of the [SourceDisksNames](#) section, if any) on the source disk where the named file resides.

If this value is omitted from an entry, the named source file is assumed to be in the *path* directory that was specified in the **SourceDisksFiles** section for the given disk or, if no *path* directory was specified, in the *installation root*.

### *size*

This optional value specifies the uncompressed size, in bytes, of the given file.

## Remarks

A **SourceDisksFiles** section can have any number of entries, one for each file on the distribution disks. Any INF with a **SourceDisksFiles** section must also have an [INF SourceDisksNames section](#). By convention, **SourceDisksNames** and **SourceDisksFiles** sections follow the [INF Version section](#). (These sections are omitted from a system-supplied INF, which instead specifies a **LayoutFile** entry in its **Version** section.)

Each *filename* entry must specify the exact name of a file on the source disk. You cannot use a %*strkey*% token to specify the file name. For more information about %*strkey*% tokens, see [INF Strings Section](#).

To support distribution of driver files on multiple system architectures, you can specify an architecture-specific **SourceDisksFiles** section by adding an **.x86**, **.ia64**, **.amd64**, **.arm**, or **.arm64** extension to

**SourceDisksFiles**. Be aware that, unlike other sections such as a **DDInstall** section, the platform extensions for a **SourceDisksFiles** section are not **.ntx86**, **.ntia64**, or **.ntamd64**.

For example, to specify a source disk names section for an x86-based system, use a **SourceDisksFiles.x86** section, not a **SourceDisksFiles.ntx86** section. Similarly, use a **SourceDisksFiles.ia64** section to specify an Itanium-based system and a **SourceDisksFiles.amd64** section to specify an x64-based system.

During installation, SetupAPI functions look for architecture-specific **SourceDisksFiles** sections before using the generic section. For example, if, during installation on an x86-based platform, Windows is copying a file that is named *driver.sys*, it will look for the file's description in [**SourceDisksFiles.x86**] before looking in [**SourceDisksFiles**].

**Important** Do not use a **SourceDisksFiles** section to copy INF files. For more information about how to copy INF files, see [Copying INFs](#).

## Examples

The following example shows a **SourceDisksNames** section and a corresponding **SourceDisksFiles** section. Note that this example has only a **SourceDisksFiles.x86** section, specifying the files for the x86 architecture. An INF that supports another architecture will need a corresponding **SourceDisksFiles** section for that architecture, or the use of an undecorated [**SourceDisksFiles**] section, which supports all architectures.

```
[SourceDisksNames]
;
; diskid = description[, [tagfile] [, <unused>, subdir]]
;
1 = %Floppy_Description%,,\WinNT

[SourceDisksFiles.x86]
aha154x.sys = 1,\x86 ; on distribution disk 1, in subdir \WinNT\x86

; ...
```

## See also

[CopyFiles](#),

[DestinationDirs](#)

[RenFiles](#)

[SourceDisksNames](#)

[Strings](#)

[Version](#)

# INF SourceDisksNames Section

12/1/2020 • 6 minutes to read • [Edit Online](#)

A **SourceDisksNames** section identifies the distribution disks or CD-ROM discs that contain the source files to be transferred to the target computer during installation.

```
[SourceDisksNames] |
[SourceDisksNames.x86] |
[SourceDisksNames.arm] | (Windows 8 and later versions of Windows)
[SourceDisksNames.arm64] | (Windows 10 version 1709 and later versions of Windows)
[SourceDisksNames.ia64] | (Windows XP and later versions of Windows)
[SourceDisksNames.amd64] (Windows XP and later versions of Windows)

diskid = disk-description[,tag-or-cab-file] |
diskid = disk-description[,,[tag-or-cab-file][,[unused][,path]]] |
diskid = disk-description[,,[tag-or-cab-file],[unused],[path][,flags]] |
diskid = disk-description[,,[tag-or-cab-file],[unused],[path],[flags][,tag-file]] (Windows XP and later
versions of Windows)
...
...
```

## Entries

### *diskid*

Specifies a nonnegative integer, in decimal format, that identifies a source disk. This value cannot require more than 4 bytes of storage. If there is more than one source disk for the distribution, each *diskid* entry in this section must have a unique value, such as 1, 2, 3, and so forth.

### *disk-description*

Specifies a %strkey% token or a "quoted string" that describes the contents and/or purpose of the disk identified by *diskid*. The installer can display the value of this string to the end-user during installation, for example, to identify a source disk to be inserted into a drive at a particular stage of the installation process.

Every %strkey% specification in this section must be defined in the INF's **Strings** section. Any *disk-description* that is not a %strkey% token is a user-visible string that must be delimited by double quotation marks characters ("") if it has any leading or trailing spaces.

### *tag-or-cab-file*

This optional value specifies the name of a *tag file* or *cabinet (.cab)* file supplied on the distribution disk, either in the *installation root* or in the subdirectory specified by *path*, if any. The value should specify only the file name and extension, not any directory or subdirectory.

Windows uses a tag file to verify that the user inserted the correct installation disk. Tag files are required for removable media, and are optional for fixed media.

If Windows cannot find installation files by name on the installation medium, and if *tag-or-cab-file* has the extension .cab, Windows uses it as the name of a cabinet file that contains the installation files.

If a .cab extension is specified, Windows treats the file as both a tag file and a cabinet file, as explained in the following **Remarks** section.

For Windows XP and later versions of Windows, also see the *flags* and *tag-file* entry values.

### *unused*

This entry is no longer supported for Windows 2000 and later versions of Windows.

#### *path*

This optional value specifies the directory path on the distribution disk that contains source files. The *path* is relative to the *installation root* and is expressed as \dirname\dirname2... and so forth. If this value is omitted from an entry, files are assumed to be in the installation root of the distribution disk.

You can use an [INF SourceDisksFiles section](#) to specify subdirectories, relative to a given path directory, that contain source files. However, tag files and *cabinet file* must reside either in the given path directory or in the installation root.

#### *flags*

Starting with Windows XP, setting this to **0x10** forces Windows to use *tag-or-cab-file* as a cabinet file name, and to use *tag-file* as a tag file name. Otherwise, *flags* is for internal use only.

#### *tag-file*

Starting with Windows XP, if *flags* is set to **0x10**, this optional value specifies the name of a *tag file* supplied on the distribution medium, either in the *installation root* or in the subdirectory specified by *path*. The value should specify the file name and extension without path information. For more information, see the Remarks section.

## Remarks

A **SourceDisksNames** section can have any number of entries, one for each distribution disk. Any INF with a **SourceDisksNames** section must also have an [INF SourceDisksFiles section](#). (By convention, **SourceDisksNames** and **SourceDisksFiles** sections follow the [INF Version section](#).)

These sections never appear in system-supplied INF files. Instead, system-supplied INF files specify **LayoutFile** entries in their **Version** sections.

Entries in a **SourceDisksNames** section can have either of two formats, one of which is supported only in Windows XP and later versions of Windows.

In the first format, the *tag-or-cab-file* parameter can specify either a *tag file* or a *cabinet file*. When encountering this format, Windows uses the following algorithm:

1. Treat the *tag-or-cab-file* value as a tag file name and look for the file on the installation medium. If the medium is removable and the tag file is not found, prompt the user for the correct medium. If the medium is fixed and neither the tag file nor the first file to be installed can be found, prompt the user for the correct medium.
2. Attempt to copy installation files directly from the medium.
3. Treat the *tag-or-cab-file* value as a .cab file and look for the file.
4. Attempt to copy installation files from the .cab file.
5. Prompt the user for files not found.

The second format is supported in Windows XP and later versions of Windows. With this format, you can use the *tag-or-cab-file*, *flags*, and *tag-file* entries to specify both a .cab file and a tag file. When encountering this format, Windows uses the following algorithm:

1. If the installation medium is removable, look for a tag file that matches the file name that is specified by *tag-file*. If the file is not found, prompt the user for the correct medium. If the medium is fixed, look for either the tag file or the cabinet file. If neither file is found, prompt the user for the correct medium.
2. Attempt to copy installation files from the .cab file specified by *tag-or-cab-file*.
3. Prompt the user for files not found.

For either format, you must provide a different tag file, with a different file name, for each version of the driver files.

To support distribution of driver files on multiple system architectures, you can specify an architecture-specific **SourceDiskNames** section by adding an **.x86**, **.ia64**, or **.amd64** extension to **SourceDiskNames**.

Be aware that, unlike other sections such as a *DD\Install* section, the platform extensions for a **SourceDiskNames** section are not **.ntx86**, **.ntia64**, or **.ntamd64**. For example, to specify a source disk names section for an x86-based system, use a **SourceDiskNames.x86** section, not a **SourceDiskNames.ntx86** section. Similarly, use a **SourceDiskNames.ia64** section to specify an Itanium-based system and a **SourceDiskNames.amd64** section to specify an x64-based system.

During installation, SetupAPI functions look for architecture-specific **SourceDiskNames** sections before using the generic section. For example, if, during installation on an x86-based platform, an INF file references disk "2", the **SetupAPI** functions will look for an entry for disk "2" in **SourceDiskNames.x86** before looking in **SourceDiskNames**.

SetupAPI functions use the **SourceDiskNames** and **SourceDiskNames.architecture** sections that are in the same INF file as the relevant **SourceDiskFiles** section.

## Examples

In the following example, the *write.exe* file is the same for all Windows platforms and is located in the *\common* subdirectory, under the installation root, on a CD-ROM distribution disc. The *cmd.exe* file is a platform-specific file that is only used on x86-based platforms.

```
[SourceDiskNames]
1 = "Windows NT CD-ROM",file.tag,,\common

[SourceDiskNames.x86]
2 = "Windows NT CD-ROM",file.tag,,\x86

[SourceDiskFiles]
write.exe = 1
cmd.exe = 2
```

The following example uses entries that contain separate specifications for *.tag* files and *.cab* files.

```
[Version]
signature = "$Windows NT$"
Provider = %Msft%

[SourceDisksNames]
1 = "Dajava","Dajava.cab",,0x10,"Dajava.tag"
2 = "Osc","Osc.cab",,0x10,"OSC.tag"
3 = "Win","Win.cab",,0x10,"Win.tag"
4 = "XMLDSO","XMLDSO.cab",,0x10,"XMLDSO.tag"

[SourceDisksFiles]
ArrayBvr.class=1
BvrCallback.class=1
BvrsToRun.class=1
choice.osc=2
custom.osc=2
login.osc=2
mwcloud.exe=3
mwcloudw.exe=3
mwclw32.dll=3
Atom.class=4
DTD.class=4
Entity.class=4
Entry.class=4

[DestinationDirs]
Test = 16430,InfTest ; %SystemRoot%\System32

[DefaultInstall]
CopyFiles = Test

[Test]
ArrayBvr.class
mwcloudw.exe
Entity.class
custom.osc
BvrCallback.class
BvrsToRun.class
choice.osc
login.osc
mwcloud.exe
mwclw32.dll
Atom.class
DTD.class
Entry.class

[Strings]
Msft = "Microsoft"
```

## See also

[DestinationDirs](#)

[SourceDisksFiles](#)

[Version](#)

# INF Strings Section

5/5/2020 • 7 minutes to read • [Edit Online](#)

An INF file must have at least one **Strings** section to define every `%strkey%` token specified elsewhere in that INF.

```
[Strings] |
[Strings.LanguageID] ...

strkey1 = ["some string"]
strkey2 = "    string-with-leading-or-trailing-whitespace    " |
          "very-long-multiline-string" |
          "string-with-semicolon" |
          "string-ending-in-backslash" |
          ""double-quoted-string-value""
...
...
```

## Entries

*strkey1, strkey2, ...*

Each string key in an INF file must specify a unique name that consists of letters, digits, and/or other explicitly visible characters. A % character within such a *strkey* token must be expressed as %%.

*some string | "some string"*

Specifies a string, optionally delimited by using double quotation marks characters ("), that contains letters, digits, punctuation, and possibly even certain implicitly visible characters, in particular, internal space and/or tab characters. However, an unquoted string cannot contain an internal double quotation marks ("), semicolon (;), linefeed, return, or any invisible control characters, and it cannot have a backslash () as its final character.

*"\* string-with-leading-or-trailing-whitespace\* " |*

*"very-long-multiline-string" |*

*"string-with-semicolon" |*

*"string-ending-in-backslash" |*

*" "double-quoted-string-value""*

The value specified for a `%strkey%` token *must* be enclosed in double quotation marks ("") if it meets any of the following criteria:

- If a specified string has leading or trailing white space that must be retained as part of its value, that string must be enclosed in double quotation marks characters to prevent its leading and/or trailing whitespaces from being discarded by the INF parser.
- If a long string might contain any internal linefeed or return characters because of line wrapping in the text editor, it should also be enclosed in double quotation marks to prevent truncation of the string at the initial internal linefeed or return character.
- If such a string contains a semicolon, it must be enclosed in double quotation marks to prevent the string from being truncated at the semicolon. (As already mentioned in [General Syntax Rules for INF Files](#), the semicolon character begins each comment in INF files.)

- If such a string ends in a backslash, it must be enclosed in double quotation marks to prevent the string from being concatenated with the next entry. (As already mentioned in General Syntax Rules for INF Files, the backslash character () is used as the line continuator in INF files.)
- Like an unquoted string specification, such a "quoted string" cannot contain internal double quotation marks characters. However, it can be specified as an explicitly double-quoted string value by using one or more additional pairs of double quotation marks characters (for example, ""some string"").

The INF parser not only discards the outermost pair of enclosing double quotation marks for any "quoted string" in this section, but also condenses each subsequent sequential pair of double quotation marks into a single double quotation marks character.

For example, """some string""" also becomes "some string" when it is parsed.

To summarize, any string must be enclosed in a pair of double quotation mark characters ("") if any of the following is true:

- The string contains leading or trailing white space.
- The string is so long that it line wraps.
- The string contains a semicolon or a final backslash character.
- The string itself is a quoted string.

The system INF parser discards the outermost enclosing pair of double quotation marks characters delimiting such a string, along with any leading or trailing white space characters outside the double quotation marks string delimiters.

## Remarks

Because the system INF parser strips the outermost pair of enclosing double quotation marks from any "quoted string" defining a %strkey% token, many of the system INF files define all %strkey% tokens as "quoted string"s to avoid the unintended loss of leading and trailing whitespaces during INF parsing. The use of "quoted string"s also ensures that especially long string values that wrap across lines cannot be truncated, and that strings with ending backslashes cannot be concatenated to the next line in the INF file.

To create a single international INF file, an INF can have a set of locale-specific **Strings.LanguageID** sections, as shown in the formal syntax statement. The *LanguageID* extension is a hexadecimal value that is defined as follows:

- The lower 10 bits contain the primary language ID and the next 6 bits contain the sublanguage ID, as specified by the MAKELANGID macro defined in *Winnt.h*.
- The language and sublanguage IDs must match the system-defined values of the Win32 LANG\_XXX and SUBLANG\_XXX constants defined in *Winnt.h*.

For example, a *LanguageID* value of 0x0407 represents a primary language ID of LANG\_GERMAN (07) with a sublanguage ID of SUBLANG\_GERMAN (01).

An INF file can contain only one **Strings** section, along with one **Strings.LanguageID** section for each *LanguageID* value.

Windows selects a single **Strings** section that is used to translate all %strkey% tokens for the installation. Depending on the current locale of a particular computer, Windows selects a **Strings** section in the following way:

1. Windows first looks for the *.LanguageID* values in the INF that match the current locale assigned to the computer. If an exact match is found, Windows uses that **Strings.LanguageID** INF section to translate all %strkey% tokens that are defined within the INF.

You do need to duplicate all string tokens across all **Strings** sections, even numeric/fixed constants that do not need to be localized.

2. Otherwise, Windows looks next for a match to the LANG\_XXX value with the value of SUBLANG\_NEUTRAL as the SUBLANG\_XXX. If such a match is found, Windows uses that INF section to translate all %strkey% tokens that are defined within the INF.
3. Otherwise, Windows looks next for a match to the LANG\_XXX value and any valid SUBLANG\_XXX for the same LANG\_XXX family. If such a partial match is found, use that Strings.LanguagelD INF section to translate all %strkey% tokens that are defined within the INF.
4. Otherwise, Windows uses the undecorated Strings section to all translate %strkey% tokens that are defined within the INF.

By convention, and for convenience in creating a set of INF files for the international market, **Strings** sections are the last within all system INF files. Using %strkey% tokens for all user-visible string values within an INF, and placing them in per-locale **Strings** sections, simplifies the translation of such strings. For more information about locale-specific INF files, see [Creating International INF Files](#).

Although **Strings** sections are the last sections in every INF file, any specified %strkey% token defined in a **Strings** section can be used repeatedly elsewhere in the INF, in particular, wherever the translated value of that token is required. The [SetupAPI](#) functions expand each %strkey% token to the specified string and then use that expanded value for further INF processing.

The use of %strkey% tokens within INF files is not restricted to user-visible string values. These tokens can be used in any manner convenient to the INF writer, as long as each token is defined within a **Strings** section. For example, when you write an INF file that requires the specification of several GUIDs, it might be convenient to create a %strkey% token for each GUID, by using a meaningful name as a substitute for each such GUID value.

Specifying a set of %strkey% = "{GUID}" values in the INF file's **Strings** section requires you to type each explicit GUID values only once. This can help provide more readable internal INF documentation than by using explicit GUID values throughout the INF file.

All %strkey% tokens must be defined within the INF file in which they are referenced. Therefore, for any INF file that has **Include** and **Needs** entries, an included INF must have its own **Strings** section to define all %strkey% tokens referenced in that INF.

In an INF **Strings** section, the maximum length, in characters, of a substitution string, including a terminating NULL character, is 4096 (Windows Vista and later versions of Windows) and 512 (Windows Server 2003, Windows XP, and Windows 2000). After string substitution, the maximum length, in characters, of an INF file string is 4096, including a terminating NULL character.

## Examples

The following example shows a fragment of a **Strings** section from a system-supplied locale-specific *dvd.inf* for installations in English-speaking countries/regions.

```
[Strings]
Msft="Microsoft"
MfgToshiba="Toshiba"
Tosh404.DeviceDesc="Toshiba DVD decoder card"
; ...
```

The following example shows string concatenation.

```
[OEM Windows System Component Verification]
OID = 1.3.6.1.4.1.311.10.3.7      ; WHQL OEM OID
Notice = "%A% %B% %C% %D% %E%"
[Strings]
A = "This certificate is used to sign untested drivers that have not passed the Windows Hardware
Quality Labs (WHQL) testing process."
B = "This certificate and drivers signed with this certificate are intended for use in test
environments only, and are not intended for use in any other context."
C = "Vendors who distribute this certificate or drivers signed with this certificate outside a test
environment may be in violation of their driver signing agreement."
D = "Vendors who have their drivers signed with this certificate do so at their own risk."
E = "In particular, Microsoft assumes no liability for any damages that may result from the
distribution of this certificate or drivers signed with this certificate outside the test environment
described in a vendor's driver signing agreement."
```

## See also

[\*DDInstall\*](#)

[\*DDInstall.ColInstallers\*](#)

[\*DDInstall.HW\*](#)

[\*DDInstall.Interfaces\*](#)

[\*DDInstall.Services\*](#)

[\*Manufacturer\*](#)

[\*InterfaceInstall32\*](#)

[\*Models\*](#)

[\*SourceDiskNames\*](#)

[\*Version\*](#)

# INF Version Section

12/1/2020 • 8 minutes to read • [Edit Online](#)

By convention, the **Version** section appears first in INF files. Every INF file must have this section.

```
[Version]

Signature="signature-name"
[Class=class-name]
[ClassGuid={nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}]
[Provider=%INF-creator%]
[ExtensionId={xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}]
[LayoutFile=filename.inf [,filename.inf]... ] (Windows 2000 and Windows XP)
[CatalogFile=filename.cat]
[CatalogFile.nt=unique-filename.cat]
[CatalogFile.ntx86=unique-filename.cat]
[CatalogFile.ntia64=unique-filename.cat] (Windows XP and later versions of Windows)
[CatalogFile.ntamd64=unique-filename.cat] (Windows XP and later versions of Windows)
[CatalogFile.ntarm=unique-filename.cat] (Windows 8 and later versions of Windows)
[CatalogFile.ntarm64=unique-filename.cat] (Windows XP and later versions of Windows)

DriverVer=mm/dd/yyyy,w.x.y.z
[PnpLockDown=0|1] (Windows Vista and later versions of Windows)
[DriverPackageDisplayName=%driver-package-description%]
[DriverPackageType=PackageType]
```

## Entries

**Signature="signature-name"**

Must be **\$Windows NT\$** or **\$Chicago\$**. This indicates the operating systems for which this INF is valid. These signature values have the following meanings.

SIGNATURE VALUE	MEANING
<b>\$Windows NT\$</b>	All Windows operating systems
<b>\$Chicago\$</b>	All Windows operating systems

The enclosing dollar sign characters (\$) are required but these strings are case-insensitive. If *signature-name* is none of these string values, the file is not accepted as a valid INF.

Generally, Windows does not differentiate among these signature values. One of them must be specified, but it does not matter which one. You should specify the appropriate value so that someone reading an INF file can determine the operating systems for which it is intended.

Some class installers put additional requirements on how the signature value must be specified. Such requirements, if they exist, are discussed in device type-specific sections of this Windows Driver Kit (WDK).

An INF must supply OS-specific installation information by appending system-defined extensions to its *DD\Install* sections, whether the *signature-name* is **\$Windows NT\$** or **\$Chicago\$**. (See [Creating INF Files for Multiple Platforms and Operating Systems](#) for a discussion of these extensions.)

**Class=class-name**

For any standard type of device, this specifies the name of the [device setup class](#) for the type of device that is installed by using this INF file. This name is usually one of the system-defined class names, such as **Net** or **Display**, which are listed in *Devguid.h*. For more information, see [System-Supplied Device Setup Classes](#).

If an INF specifies a **Class**, it should also specify the corresponding system-defined GUID value for its **ClassGUID** entry. Specifying the matching GUID value for a device of any predefined device setup class can install the device and its drivers faster because this helps the system setup code to optimize its INF searching.

If an INF adds a new setup class of devices to the system, it should supply a unique, case-insensitive *class-name* value that differs from any of the system-supplied classes in *Devguid.h*. The length of the *class-name* string must be 32 characters or less. The INF must specify a newly generated GUID value for the **ClassGUID** entry. Also see [INF ClassInstall32 Section](#).

This entry is irrelevant to an INF that installs neither a new device driver under a predefined device setup class nor a new device setup class.

**Note** This entry is required for device drivers that are installed through the Plug and Play (PnP) manager.

**ClassGuid**=*{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}*

Specifies the [device setup class](#) GUID. The GUID value is formatted as shown here, where each *n* is a hexadecimal digit.

This GUID value specifies the device setup class subkey in the registry ...\\Class tree under which to write registry information for the drivers of devices that are installed from this INF file. This class-specific GUID value also identifies the device class installer for the type of device and class-specific property page provider, if any.

For a new [device setup class](#), the INF must specify a newly generated **ClassGUID** value. For more information about how to create GUIDs, see [Using GUIDs in Drivers](#). Also see [Device Setup Classes](#).

**Note** This entry is required for device drivers that are installed through the PnP manager.

**ExtensionId**=*{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}* Specifies the extension ID GUID when authoring an extension INF. The GUID value is formatted as shown here, where each *x* is a hexadecimal digit.

When creating the initial version of an extension INF, the INF must specify a newly generated **ExtensionId** value. However, when updating an existing extension INF, the **ExtensionId** must remain the same so that multiple related versions of the extension INF are versioned against each other instead of being treated as independent extension INFs that may be simultaneously installed on the same device instance. For more information about how to author extension INFs, see [Using an Extension INF File](#).

**Note** This entry is only required when creating an extension INF, as identified by specifying

`Class = Extension` and `ClassGuid = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}` .

**ClassVer**=*major.minor*

Reserved for system use unless explicitly required by a device class such as Printer. For example, see [V4 Driver INF](#).

**Provider**=%*INF-creator*%

Identifies the provider of the INF file. Typically, this is specified as an %*OrganizationName*% token that is expanded later in the INF file's [Strings](#) section. The maximum length, in characters, of a provider name is LINE\_LEN.

For example, INF files supplied with the system typically specify the *INF-creator* as %Msft% and define %Msft% = "Microsoft" in their [Strings](#) sections.

**Note** This entry is required for device drivers that are installed through the PnP manager.

**CatalogFile=filename.cat**

Specifies a catalog (.cat) file to be included on the distribution media of a device/driver.

When a [driver package](#) is submitted to Microsoft for digital signing, WHQL provides a [catalog file](#) for the driver package after WHQL has tested and assigned digital signatures to the package. For more information about the testing and signing of IHV or OEM driver packages, see [WHQL Release Signature](#). Catalog files are not listed in the [SourceDisksFiles](#) section or [CopyFiles](#) directive of the INF. Windows assumes that the catalog file is in the same location as the INF file.

System-supplied INF files never have **CatalogFile=** entries because the operating system validates the signature for such an INF against all system-supplied *xxx.cat* files.

**CatalogFile.nt=unique-filename.cat |**

**CatalogFile.ntx86=unique-filename.cat |**

**CatalogFile.ntia64=unique-filename.cat |**

**CatalogFile.ntamd64=unique-filename.cat**

**CatalogFile.ntarm=unique-filename.cat**

**CatalogFile.ntarm64=unique-filename.cat**

Specifies another INF-writer-determined, unique file name, with the .cat extension, of a catalog file. If these optional entries are omitted, a given **CatalogFile=filename.cat** is used for validating WDM device/driver installations.

If any decorated **CatalogFile.xxx=** entry exists in an INF's **Version** section together with an undecorated **CatalogFile=** entry, the undecorated entry is assumed to identify a *filename.cat* for validating device installations, driver installations, or both on those platforms for which a decorated entry is not specified.

Any cross-platform device driver INF file that has **CatalogFile=** and **CatalogFile.xxx=** entries must supply a unique IHV/OEM-determined name for each such .cat file.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

**Note** Because the same .cat file can be used across all supported platforms, the use of this entry is not required or recommended. However, you must use this entry if you want to create platform-specific .cat files for your driver package.

**DriverVer= mm/dd/yyyy.w.x.y.z**

This entry specifies version information for drivers that are installed by this INF file. Starting with Windows 2000, this entry is required.

For information about how to specify this entry, see [INF DriverVer Directive](#).

**PnpLockDown=0|1**

Specifies whether Plug and Play (PnP) prevents applications from directly modifying the files that a [driver package](#)'s INF file specifies. If the **PnpLockDown** directive is set to 1, PnP prevents applications from directly modifying the files that are copied by INF [CopyFiles](#) directives. Otherwise, if the directive is not included in an INF file or the value of the directive is set to zero, an application that has administrator privileges can directly modify these files. Driver files that are protected in this manner are referred to as *third-party protected driver files*.

To ensure the integrity of a PnP driver installation, applications should not directly modify driver files that are copied by the driver package INF file. Applications should only use the device installation mechanisms

provided by Windows to update PnP drivers.

Starting with Windows Vista, a driver package should set **PnpLockDown** to 1 to prevent an application from directly modifying driver files. However, some existing applications that uninstall driver packages do directly delete driver files. To maintain compatibility with these applications, the **PnpLockDown** directive for such driver package should be set to zero.

**Note** Although PnP on Windows Vista and later versions of Windows does not require that an INF file include a **PnpLockDown** directive in order to install a driver, PnP in a future version of Windows might require that INF files for PnP [driver packages](#) include the **PnpLockDown** directive.

**DriverPackageDisplayName=%driver-package-description%**

Deprecated. Was previously used by Driver Install Frameworks (DIFx). For info about the DIFx deprecation, see [DIFx Guidelines](#).

**DriverPackageType= *PackageType***

Deprecated. Was previously used by Driver Install Frameworks (DIFx). For info about the DIFx deprecation, see [DIFx Guidelines](#).

## Remarks

When a [driver package](#) passes Microsoft Windows Hardware Quality Lab (WHQL) testing, WHQL returns .cat catalog files to the IHV or OEM. Each .catfile contains a digitally encrypted signature for the driver package. The IHV or OEM must list these .catfiles in the **INF Version** section and must supply the files on the distribution media, in the same location as the INF file. The .catfiles must be uncompressed.

**Note** If an **INF Version** section does not include at least one **CatalogFile** or **CatalogFile.ntxxx** entry, the driver is treated as unsigned, and the dates listed in the **DriverVer** directive are not displayed by Windows.

For more information, see [Driver Signing](#).

## Examples

The following example shows a **Version** section typical of a simple device-driver INF, followed by the required **SourceDiskNames** and **SourceDiskFiles** sections implied by the entries specified in this sample **Version** section:

```
[Version]
Signature="$Windows NT$"
Class=SCSIAdapter
ClassGUID={4D36E97B-E325-11CE-BFC1-08002BE10318}
Provider=%INF_Provider%
CatalogFile=aha154_ntx86.cat
DriverVer=01/29/2010

[SourceDisksNames]
;
; diskid = description[, [tagfile] [, <unused>, subdir]]
;
1 = %Floppy_Description%,,\WinNT

[SourceDisksFiles.x86]
;
; filename_on_source = diskID[, [subdir][, size]]
;
aha154x.sys = 1,\x86

; ...

[Strings]
INF_Provider="Adaptec"
Floppy_Description = "Adaptec Drivers Disk"
; ...
```

## See also

[\*DDInstall\*](#)

[\*\*SourceDisksNames\*\*](#)

[\*\*SourceDisksFiles\*\*](#)

[\*\*Strings\*\*](#)

# INF AddComponent Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

An **AddComponent** directive is used within an **INF DDInstall.Components** section of an [extension INF file](#). It creates a virtual child device for the software component under the current device. This directive is supported for Windows 10 version 1703 and later.

```
[DDInstall.Components]  
AddComponent=ComponentName,[flags],component-install-section
```

## Entries

*ComponentName*

Specifies the name of the software component to create. Each **AddComponent** directive in an INF file must have a unique value.

*flags*

Specifies one or more (ORed) flags, currently undefined but reserved for future use.

*component-install-section*

References an INF-writer-defined section that contains information for creating the named software component for this device.

## Remarks

Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

An **AddComponent** directive must reference a named *component-install-section* elsewhere in the INF file. Each such section has the following form:

```
[component-install-section]  
ComponentIDs=component-id[,component-id] ...  
[Description=description]
```

Each *component-install-section* must have at least the **ComponentIDs** entry as shown here. However, the remaining entries are optional.

Note that **ComponentIDs** are [HardwareIDs](#), which means they are strings defined by the hardware developer. To ensure uniqueness of these IDs, in most cases, we recommend following the identifier schema used for [PCI devices](#). It is possible that a vendor might want to use a different schema, but that depends on the scenario.

For example, a vendor with multiple components on a single device might want to associate the hardware IDs of the component with the parent. In this case, they could create a **ComponentID** by appending a four-character vendor-defined component identifier to the hardware ID of the parent.

## Component-Install Section Entries and Values

**ComponentIDs**=*id1*[, *id2*] ... [, *idN*]

Specifies the component identifiers for a software component. Component IDs work the same way that Hardware IDs do, and should follow [similar formatting](#). For a software component, the system prepends the INF-supplied values with `swc\` to create the Hardware IDs. For example, a **ComponentIDs** value of `VID0001&PID0001` results in a hardware ID of `SWC\VID0001&PID0001`.

**Description**=*description*

Optionally specifies a string that describes the software component, typically for localization, expressed as a `%strkey%` token defined in an [INF Strings section](#).

If a description string contains any `%strkey%` tokens, each token can represent a maximum of 511 characters. The total string, after any string token substitutions, should not exceed 1024 characters.

## See Also

[Using a Component INF File](#).

[DD\Install\Components](#)

[INF AddSoftware Directive](#)

# INF AddEventProvider Directive

11/2/2020 • 5 minutes to read • [Edit Online](#)

An **AddEventProvider** directive is used within an [INF DDInstall.Events section](#). It specifies characteristics of the [Event Tracing for Windows](#) (ETW) providers associated with drivers. This directive is supported for Windows 10 version 1809 and later.

```
[DDInstall.Events]  
  
AddEventProvider={ProviderGUID},event-provider-install-section  
...
```

## Entries

### *ProviderGUID*

Specifies the GUID value that identifies the provider. This can be expressed as an explicit GUID value of the form

{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnn} or as a %strkey% token defined to  
{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnn} in a [Strings](#) section of the INF file.

### *event-provider-install-section*

References an INF-writer-defined section that contains information for registering the provider for this device (or devices). For more information, see the following **Remarks** section.

## Remarks

The system-defined and case-insensitive extensions can be inserted into a [DDInstall.Events](#) section that contains an **AddEventProvider** directive in cross-operating system and/or cross-platform INF files to specify platform-specific or OS-specific installations.

Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

An **AddEventProvider** directive must reference a named *event-provider-install-section* elsewhere in the INF file. Each such section has the following form:

```
[event-provider-install-section]  
  
ProviderName=name  
ResourceFile=path-to-file  
[MessageFile=path-to-file]  
[ParameterFile=path-to-file]  
(ImportChannel=channel-name) |  
(AddChannel=channel-name,channel-type[,channel-install-section])  
...
```

Each *event-provider-install-section* must provide **ProviderName** and **ResourceFile**. Optionally, specify a list of channels for the provider using any combination of **ImportChannel(s)** and **AddChannel(s)**, each on a separate line. For more information about channel lists in an INF file, see [Specifying a Channel List](#) below. For more information about [Windows Event Log](#) channels, see [Defining Channels](#).

### **Event-Provider-Install Section Entries and Values**

**ProviderName**=*name*

Specifies the name of the provider. The name cannot be longer than 255 characters, and cannot contain the characters: '>', '<', '&', '"', '|', ':', ';', '?', '\*', or characters with ASCII values less than 31. In addition, the name must follow the general constraints on file and registry key names. These constraints can be found at [Naming a File](#) and [Registry Element Size Limits](#).

#### **ResourceFile**=*path-to-file*

Specifies the path to the exe or dll that contains the provider's metadata resources, expressed as %dirid%\filename.

The *dirid* number is either a custom directory identifier or one of the system-defined directory identifiers described in [Using Dirids](#).

#### **MessageFile**=*path-to-file*

Optionally specifies the path to the exe or dll that contains the provider's localized message resources, expressed as %dirid%\filename.

#### **ParameterFile**=*path-to-file*

Optionally specifies the path to the exe or dll that contains the provider's parameter string resources, expressed as %dirid%\filename.

#### **ImportChannel**=*channel-name*

Optionally specifies a channel that has been defined by another provider. For more information, see the following [Specifying a Channel List](#) section.

#### **AddChannel**=*channel-name,channel-type[,channel-install-section]*

Optionally specifies a channel with a sub-directive that optionally references an INF-writer-defined channel-install-section elsewhere in the INF file. For more information, see the following [Specifying a Channel List](#) section.

### **Specifying a Channel List**

You can specify a list of channels for the provider within its *event-provider-install-section*. You can import a channel or add a channel to the list and the order of these channels is preserved. For more information, see [Defining Channels](#).

The *channel-name* must be unique within the list of channels that the provider uses. The *channel-name* must be less than 255 characters and cannot contain the following characters: '>', '<', '&', '"', '|', ':', ';', '?', '\*', or characters with ASCII values less than 31.

The *channel-type* can be specified as one of the following numeric values, expressed either in decimal or, as shown in the following list, in hexadecimal notation.

#### **0x1 (Admin)**

Admin type channels support events that target end users, administrators, and support personnel. Events written to the Admin channels should have a well-defined solution on which the administrator can act.

#### **0x2 (Operational)**

Operational type channels support events that are used for analyzing and diagnosing a problem or occurrence. They can be used to trigger tools or tasks based on the problem or occurrence.

#### **0x3 (Analytic)**

Analytic type channels support events that are published in high volume. They describe program operation and indicate problems that cannot be handled by user intervention.

#### **0x4 (Debug)**

Debug type channels support events that are used solely by developers to diagnose a problem for debugging.

An **AddChannel** sub-directive can also reference a *channel-install-section* elsewhere in the INF file. Each such section has the following form:

```
[channel-install-section]

[Isolation=isolation-type]
[Access=access-string]
[Enabled=0|1]
[Value=value]
[LoggingMaxSize=max-size]
[LoggingRetention=retention-type]
[LoggingAutoBackup=0|1]
```

For more information about channel attributes, see [ChannelType](#) defined within [EventManifest Schema](#).

## Channel-Install Section Entries and Values

### **Isolation**=*isolation-type*

Optionally specifies the default access permissions for the channel as one of the following numeric values, expressed either in decimal or, as shown in the following list, in hexadecimal notation. If omitted, this defaults to **0x1** (Application).

**0x1** (Application)

**0x2** (System)

**0x3** (Custom)

### **Access**=*access-string*

Optionally specifies a [Security Descriptor Definition Language](#) (SDDL) access descriptor that controls access to the log file that backs the channel.

This string controls read access to the file (the write permissions are ignored) if the **Isolation** is set to **0x1** (Application) or **0x2** (System), while it controls write access to the channel and read access to the file if the isolation attribute is set to **0x3** (Custom).

### **Enabled**=*0|1*

Optionally specifies whether the channel is enabled. If omitted, this defaults to 0 (disabled).

Because **0x3** (Analytic) and **0x4** (Debug) *channel-type* are high volume channels, you should set the **Enabled** to 1 only when investigating an issue with a component that writes to that channel. Each time you enable a **0x3** (Analytic) and **0x4** (Debug) channel, the service clears the events from the channel.

### **Value**=*value*

Optionally specifies a numeric identifier that uniquely identifies the channel within the list of channels that the provider defines.

### **LoggingMaxSize**=*max-size*

Optionally specifies the maximum size, in bytes, of the log file. The default (and minimum) value is 1 MB.

### **LoggingRetention**=*retention-type*

Optionally specifies whether the log file is **0x1** (circular) or **0x2** (sequential). The default is **0x1** (circular) for **0x1** (Admin) and **0x2** (Operational) *channel-type* and **0x2** (sequential) for **0x3** (Analytic) and **0x4** (Debug) *channel-type*.

### **LoggingAutoBackup**=*0|1*

Optionally specifies whether to create a new log file when the current log file reaches its maximum size. Set to 1 to request that the service create a new file when the log file reaches its maximum size; otherwise, 0. You can set the **LoggingAutoBackup** to 1 only if the **LoggingRetention** is set to **0x2** (sequential) and only for **0x1** (Admin) and **0x2** (Operational) *channel-type*.

## Examples

This example shows the event-provider-install sections referenced by the **AddEventProvider** directives as already shown earlier in the example for [DDInstall.Events](#).

```
[foo_Event_Provider_Inst]
ProviderName = FooCollector
ResourceFile = %13%\FooResource.dll
MessageFile = %13%\FooMessage.exe

[bar_Event_Provider_Inst]
ProviderName = BarCollector
ResourceFile = %13%\BarResource.exe
MessageFile = %13%\BarMessage.dll
ParameterFile = %13%\BarParameter.dll
ImportChannel = Microsoft-Windows-BaseProvider/Admin
AddChannel = Bar-Provider/Admin,0x1,bar_Channel2_Inst ; Admin type
ImportChannel = Microsoft-Windows-BaseProvider/Operational
ImportChannel = Microsoft-Windows-SampleProvider/Admin
AddChannel = Bar-Provider/Debug,0x4 ; Debug type

[bar_Channel2_Inst]
Isolation = 2 ; System isolation
Enabled = 1
Value = 17
LoggingMaxSize = 20971520
LoggingRetention = 2 ; Sequential
LoggingAutoBackup = 1
```

## See also

[DDInstall.Events](#)

# INF AddInterface Directive

11/2/2020 • 4 minutes to read • [Edit Online](#)

One or more **AddInterface** directives can be specified within an [INF DDInstall.Interfaces section](#). This directive installs device-specific support for [device interface classes](#) exported to higher level components, such as other drivers or applications. The directive typically references an *add-interface-section*, which sets up registry information for the device-specific instance of the device interface class.

```
[DDInstall.Interfaces]  
AddInterface={InterfaceClassGUID} [,reference-string] [,,[add-interface-section][,flags]]]
```

An exported device interface class can be one of the system-defined device interface classes, such as those that are defined by kernel streaming, or a new device interface class specified by an [INF InterfaceInstall32 section](#).

## Entries

### *InterfaceClassGUID*

Specifies the GUID value that identifies the device interface class. This can be expressed as an explicit GUID value of the form `{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnnn}` or as a `%strkey%` token defined to "`{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnnn}`" in a [Strings](#) section of the INF file.

For more information about how to create a GUID, see [Using GUIDs in Drivers](#). For the system-defined interface class GUIDS, see the appropriate header, such as *Ks.h* for the kernel-streaming interface GUIDs.

### *reference-string*

This optional value, associated with the device-specific instance of the specified interface class, can be expressed either as a "quoted string" or as a `%strkey%` token defined in an [INF Strings section](#).

PnP function and filter drivers usually omit this value from the `AddInterface=` entries in their INF files. A *reference-string* is used by the *swenum* driver as a placeholder for software devices that are created on demand by using multiple instances of a single interface class. The same *InterfaceClassGUID* value can be specified in INF entries with two or more unique *reference-strings*. Because the I/O manager passes the *reference-string* value as a path component of the interface instance's name whenever it is opened, the installed driver can discriminate between interface instances of the same class for a single device.

### *add-interface-section*

References the name of a section elsewhere in the INF file. This typically contains an [INF AddReg directive](#) to set up the registry entries exporting the driver's support of this [device interface class](#). For more information, see the following **Remarks** section.

### *flags*

If specified, this entry must be zero.

## Remarks

If the [device interface class](#) identified by a specified `{InterfaceClassGUID}` is not installed already, the system setup code installs that class in the system. Any INF file that installs a new class also has an [INF InterfaceInstall32 section](#). This section contains the specified `{InterfaceClassGUID}` and references an *interface-install-section* that sets up interface-specific installation operations for that class.

To enable an instance of a device interface class for run-time use by higher level components, a device driver must first call [IoRegisterDeviceInterface](#) to retrieve the symbolic link name of the device interface instance to enable. Usually, a PnP function or filter driver makes this call from its [AddDevice](#) routine. To enable instances of device interfaces provisioned in the INF, the device driver must provide the `{ /InterfaceClassGUID }` and `reference-string` specified in the INF when it calls [IoRegisterDeviceInterface](#). The driver then calls [IoSetDeviceInterfaceState](#) to enable the interface using the symbolic link name returned by [IoRegisterDeviceInterface](#).

Each **AddInterface** directive in an [INF DDInstall.Interfaces section](#) can reference an INF-writer-defined *add-interface-section* elsewhere in the INF file. Each INF-writer-defined section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

An *add-interface-section* referenced by the **AddInterface** directive has the following form:

```
[add-interface-section]

AddReg=add-registry-section[, add-registry-section]...
[AddProperty=add-property-section[, add-property-section] ...] (Windows Vista and later versions of
Windows)
[DelReg=del-registry-section[, del-registry-section] ...]
[DelProperty=del-property-section[, del-property-section] ...] (Windows Vista and later versions of
Windows)
[BitReg=bit-registry-section[, bit-registry-section] ...]
[CopyFiles=@filename | file-list-section[,file-list-section]...]
[DelFiles=file-list-section[,file-list-section]...]
[RenFiles=file-list-section[,file-list-section]...]
[UpdateInis=update-ini-section[, update-ini-section] ...]
[UpdateIniFields=update-inifields-section[, update-inifields-section] ...]
[Ini2Reg=ini-to-registry-section[, ini-to-registry-section] ...]
```

Starting with Windows Vista, you can set device interface properties by including [INF AddProperty directives](#) in an *add-interface section*. You can also delete device interface properties by including [INF DelProperty directives](#) in an *add-interface section*. However, you should use **AddProperty** or **DelProperty** directives only to modify device interface properties that are new to Windows Vista or a later version of Windows operating systems. For device interface properties that were introduced on Windows Server 2003, Windows XP, or Windows 2000, and that have corresponding registry value entries, you should continue to use [INF AddReg directives](#) and [INF DelReg directives](#) to set and delete the device interface properties. These guidelines apply to system-defined properties and custom properties. For more information about how to use the **AddProperty** directive and **DelProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

Typically, an *add-interface-section* contains only an [INF AddReg directive](#) that, in turn, references a single *add-registry-section*. The *add-registry-section* is used to store information in the registry about the interfaces supported by the device driver for subsequent use by still higher level drivers and applications.

An *add-registry-section* referenced within an *add-interface-section* is specific to the instances for the device, driver, and interface. It might have a value entry defining a friendly name for the exported device interface instance so that still higher level components can refer to that interface by its friendly name in the user interface.

An **HKR** specified in such an *add-registry-section* section designates the run-time accessible state registry key for a device interface. The driver can access state stored in this registry key at runtime by calling [IoOpenDeviceInterfaceRegistryKey](#) to retrieve a HANDLE to the state registry key. User mode components can query the state by calling [CM\\_Open\\_Device\\_Interface\\_Key](#).

## Examples

This example shows some of the expansion of the *DD\Install.Interfaces* section for a particular audio device that supports system-defined kernel-streaming interfaces.

```
; ...
[ESS6881.Device.Interfaces]
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_CAPTURE%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Topo%,\
ESSAud.Interface.Topology
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_CAPTURE%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_FMSynth%,WDM.Interface.FMSynth
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_FMSynth%,\
WDM.Interface.FMSynth

[ESSAud.Interface.Wave]
AddReg=ESSAud.Interface.Wave.AddReg

[ESSAud.Interface.Wave.AddReg]
HKR,,CLSID,,%Proxy.CLSID%
HKR,,FriendlyName,,%ESSAud.Wave.szPname%
; ...
[WDM.Interface.UART]
AddReg=WDM.Interface.UART.AddReg

[WDM.Interface.UART.AddReg]
HKR,,CLSID,,%Proxy.CLSID%
HKR,,FriendlyName,,%WDM.UART.szPname%
; ...
[Strings]
KSCATEGORY_AUDIO="{6994ad04-93ef-11d0-a3cc-00a0c9223196}"
KSCATEGORY_RENDER="{65e8773e-8f56-11d0-a3b9-00a0c9223196}"
KSCATEGORY_CAPTURE="{65e8773d-8f56-11d0-a3b9-00a0c9223196}"
; ...
KSNAME_WAVE="Wave"
KSNAME_UART="UART"
; ...
Proxy.CLSID="{17cca71b-ecd7-11d0-b908-00a0c9223196}"
; ...
ESSAud.Wave.szPname="ESS AudioDrive"
; ...
```

## See also

[AddProperty](#)

[AddReg](#)

[BitReg](#)

[CopyFiles](#)

[DD\Install.Interfaces](#)

[DelFiles](#)

[DelProperty](#)

[DelReg](#)

[Ini2Reg](#)

[InterfaceInstall32](#)

[IoRegisterDeviceInterface](#)

[IoSetDeviceInterfaceState](#)

[RenFiles](#)

[UpdateIniFields](#)

[UpdateInis](#)

# INF AddPowerSetting Directive

11/2/2020 • 12 minutes to read • [Edit Online](#)

An **AddPowerSetting** directive references one or more sections that are used to modify or create power setting information. Each *add-power-setting-section* defines a power setting, the allowed values for the power setting, the friendly name of the power setting, and the description of the power setting. An *add-power-setting-section* also specifies the default value for each power scheme personality. For more information about power settings and power scheme personalities, see [Managing Device Performance States](#).

```
[DDIInstall] |
[DDIInstall.HW] |
[DDIInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows Vista)
[ClassInstall32.ntamd64] (Windows Vista)
[ClassInstall32.ntarm]
[ClassInstall32.ntarm64]
```

```
AddPowerSetting=add-power-setting-section[,add-power-setting-section]
```

In general, an *add-power-setting-section* includes the following directives:

- A **SubGroup** directive.
- A **Setting** directive
- A list of two or more **Value** directives or one **ValueRange** directive.
- A set of six **Default** directives.

An *add-power-setting-section* takes one of the following two possible forms:

- If the allowed power settings values can best be defined as a set of two or more discrete values, use a list of **Value** directives to specify the allowed values, as follows:

```
[add-power-setting-section]

[SubGroup = {subgroup-guid}] | SubGroup = {subgroup-guid}, subgroup-name, subgroup-description,
subgroup-icon
Setting = {setting-guid}, [setting-name],[setting-description],[setting-icon]
Value = value-index, value-name,[value-description], value-flags, value-data
Value = value-index, value-name,[value-description], value-flags, value-data
[Value = value-index, value-name,[value-description], value-flags, value-data
...
Value = value-index, value-name,[value-description], value-flags, value-data]

(Six required Default directives, each one of which has the following form)
Default = power-scheme-personality-GUID, AC/DC-index, default-setting-index | default-setting-value
...
```

- If the allowed power settings values can best be defined as an incremented sequence of nonnegative integer values within a specified range, use one **ValueRange** directive to specify allowed values, as follows:

```

[add-power-setting-section]

[SubGroup = {subgroup-guid}] |
SubGroup = {subgroup-guid}, subgroup-name, subgroup-description, subgroup-icon
Setting = {setting-guid}, [setting-name],[setting-description],[setting-icon]
ValueRange = range-minimum-value, range-maximum-value, range-increment, [range-unit-label]

(Six required Default directives, each one of which has the following form)
Default = power-scheme-personality-GUID, AC/DC-index, default-setting-index | default-setting-value
...

```

## Entries

**Note** Except for a *value-data* entry, all the following entries that supply a string value can specify the string in one of the ways that are described in [Specifying an AddPowerSetting String Entry Value](#).

### SubGroup

A subgroup groups power settings that are logically related.

To specify a system-defined subgroup, include a **SubGroup** directive and supply only the *subgroup-guid* entry. The system-defined subgroups are represented by the constants `GUID_Xxx_SUBGROUP` and `NO_SUBGROUP_GUID`, which are defined in *Wdm.h*.

For example, `GUID_VIDEO_SUBGROUP` represents the subgroup that contains the video power settings for a power scheme personality. The `NO_SUBGROUP_GUID` constant represents a collection of settings that do not logically belong to any subgroup. If a **SubGroup** directive is not included, the setting is added by default to the collection of settings that do not logically belong to any subgroup.

To define a new subgroup, include the **SubGroup** directive and supply the following required entries: *subgroup-guid*, *subgroup-name*, *subgroup-description*, and *subgroup-icon*. The GUID of the new subgroup must be unique and the other entries should be as descriptive as possible.

#### *subgroup-guid*

The required entry supplies the GUID that identifies the subgroup. The format of this entry is `{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX}`, where "X" is a hexadecimal digit.

For example, the value of the system-defined constant `GUID_VIDEO_SUBGROUP` is `{7516B95F-F776-4464-8C53-06167F40CC99}`. This GUID represents the subgroup that contains the video power settings for a power scheme personality.

#### *subgroup-name*

A string that specifies the subgroup name of the power setting. If the subgroup is a system-defined subgroup, this entry should not be supplied. If the subgroup is new, this entry is required.

#### *subgroup-description*

A string that describes to the user the power subgroup. If the subgroup is a system-defined subgroup, this entry should not be supplied. If the subgroup is new, this entry is required.

#### *subgroup-icon*

A reference to an icon resource. If the subgroup is a system-defined subgroup, this entry should not be supplied. If the subgroup is new, this entry is required.

An icon resource must be specified as a language-neutral registry value. For information about how to specify a language-neutral registry value, see [Specifying an AddPowerSetting String Entry Value](#).

### Setting

The **Setting** directive identifies the setting to which all the other entries in the section apply. One **Setting** directive is required in an add-power-setting section and there can only be one **Setting** directive in an add-power-setting

section. If an INF file defines more than one setting, each setting must be defined in its own add-power-setting section.

The following are the entries that are associated with a **Setting** directive.

*setting-guid*

A required entry that specifies the GUID that represents the power setting. The format of this entry is {XXXXXXXX-XXXX-XXXX-XXXXXXXXXX}, where each "X" is a hexadecimal digit.

For example, the following is a custom GUID value: {BFC0D9E9-549C-483D-AD2A-3D90C98A8B03}.

*setting-name*

An optional entry that specifies a string that contains the friendly name of the power setting. **Power Options** in Control Panel displays this friendly name to a user.

*setting-description*

An optional entry that specifies a string that describes to the user the power setting and the effect that the setting has on system power and performance.

*setting-icon*

An optional entry that is a reference to an icon resource. An icon resource must be specified by a language-neutral registry value.

For information about how to specify a language-neutral-registry value, see [Specifying an AddPowerSetting String Entry Value](#).

**Value**

A **Value** directive defines an allowed value for a power setting. The **Value** directive should be used if the values can best be defined as a set of two or more values, where each value can have a value-specific custom data type. In this situation, an add-power-setting-section should include two or more **Value** directives. A user can select one of these values in **Power Options** in Control Panel to configure a power scheme.

If the allowed power setting values can best be described as incremented set of non-negative integers within a range, use the **ValueRange** directive instead of the **Value** directive to specify the allowed power setting values.

*value-index*

A required entry that specifies a unique index value, which is greater than or equal to zero, and that is used to reference the corresponding setting value. **Power Options** in Control Panel displays power setting values to a user in order of their corresponding index values, from lowest to highest.

*value-name*

A required entry that supplies a string that provides the friendly name for the corresponding setting value. **Power Options** in Control Panel displays the friendly names of the power setting values to a user.

*value-description*

An optional entry that supplies a string that describes to the user the power setting value and the effect that the setting value has on system power and performance.

*value-flags*

A required entry that specifies the data type of the corresponding value-data entry, as indicated in the following table.

FLAG VALUE	DATA TYPE
0x00000001	<a href="#">REG_BINARY</a>
0x00010001	<a href="#">REG_DWORD</a>

FLAG VALUE	DATA TYPE
0x00000000	REG_SZ

#### *value-data*

A required entry that supplies the data for the corresponding setting value, the format of which depends on the data type that is specified by corresponding *value-flags* entry, as follows:

- A [REG\\_BINARY](#) value can be specified in hexadecimal format by using 0x notation, or as a comma-separated list of paired hexadecimal numbers without the 0x notation.

For example, the following entries are equivalent: 0xFEDCBA9876543210 and the following comma-separated list of paired hexadecimal digits: FE, DC, BA, 98, 76, 54, 32, 10.

- A [REG\\_DWORD](#) value can be specified either in hexadecimal format (by using 0x notation) or in decimal format.
- A [REG\\_SZ](#) value can only be expressed as a string enclosed in double quotation marks ("quoted-string") or as a %strkey% token that is defined in the INF [Strings](#) section of an INF file.

**Note** You should not use string values because they cannot be localized. Instead, use values of type [REG\\_BINARY](#) or [REG\\_DWORD](#).

#### **ValueRange**

Use the **ValueRange** directive if the allowed power settings values can best be defined as an incremented sequence of non-negative integer values within a specified range. The power manager validates that a setting that a user selects in **Power Options** in Control Panel is one of these allowed values. The set of allowed values is determined by a minimum allowed value, a maximum allowed value, and an increment between the allowed values within the range. A value is allowed if it satisfies the following:

```
range-minimum-value + k*range-increment
```

where *range-minimum-value* is greater than or equal to zero, *k* and *range-increment* are greater than or equal to one, and the value is less than or equal to *range-maximum-value*. In addition, *range-maximum-value* should be equal to *range-minimum-value* + *k\*range-increment* for some *k*.

For example, for a *range-minimum-value* equal to 0, a *range-maximum-value* equal to 10, and a *range-increment* equal to 2, the allowed values are as follows: 0, 2, 4, 6, 8, and 10.

If the allowed power setting values can best be described as a list of values, where each value can have a value-specific custom data type, use the **Value** directive instead of the **ValueRange** directive.

#### *range-minimum-value*

A value of type [REG\\_DWORD](#) that specifies the minimum allowed power setting.

#### *range-maximum-value*

A value of type [REG\\_DWORD](#) that specifies the maximum allowed power setting value. The maximum value must be greater than or equal to *minimum-value* and should be equal to *range-minimum-value* + *k\*range-increment*, for some integer *k* that is greater than zero.

#### *range-increment*

A value of type [REG\\_DWORD](#) that is greater than zero. This value specifies the difference between consecutive values within the inclusive range that is specified by *range-minimum-value* and *range-maximum-value*.

#### *range-unit-label*

An optional string that describes the power setting value. The string, together with *setting-name*, informs the user of what type of data to enter.

For example, the string can be used to specify the value units, such as "minutes" or "%" (representing percent).

## Default

There are six **Default** directives that must be included in an **AddPowerSetting** section. A **Default** directive specifies the default value for one of the three system-defined power scheme personalities that apply to an AC power state and the three system-defined power scheme personalities that apply to a DC power state.

It is extremely important that the defaults be valid and accurate. If the user does not manually set a power setting, the power manager uses the default value that is specified by the **Default** directive.

### *power-scheme-personality-GUID*

One of the following GUIDs, which identifies the power scheme that the default value applies to.

PERSONALITY	GUID
Power saver	{A1841308-3541-4FAB-BC81-F71556F20B4A}
High performance	{8C5E7FDA-E8BF-4A96-9A85-A6E23A8C635C}
Balanced	{381B4222-F694-41F0-9685-FF5BB260DF2E}

These GUIDs are defined in *Wdm.h*.

### *AC/DC-index*

If *AC/DC-index* is 0, the setting applies to an AC power state and if *AC/DC-index* is 1, the setting applies to a DC power state. A value other than 0 or 1 is not valid.

### *default-setting-index*

If the **Value** directive is used to specify allowed values, *default-setting-index* is the value of the *value-index* entry of the **Value** directive. If the **ValueRange** directive is used to specify allowed values, this entry does not apply.

### *default-setting-value*

If the **ValueRange** directive is used to specify allowed values, *default-setting-value* is one of the allowed values that are specified by the **ValueRange** directive. If the **Value** directive is used to specify allowed values, this entry does not apply.

## Remarks

An *add-power-setting-section* name must be unique in an INF file, but it can be referenced by more than one **AddPowerSetting** directive in the same INF file. Each section name must follow the general rules that are described in [General Syntax Rules for INF Files](#).

The power manager does not automatically remove device power policies after a device is uninstalled. Installation or removal of power settings, values, and defaults can be performed by a co-installer through the system-supplied power setting routines that are defined in *Powrprof.h*. For more information about these power management routines, see the power management reference that is provided with the Microsoft Windows SDK documentation.

In addition, the *Powercfg.exe* command-line tool can be used to change power settings. For information about *Powercfg.exe*, see the Microsoft Help and Support Center.

For more information about how to use the system-defined .nt, .ntx86, .ntia64, .ntamd64, .ntarm, and .ntarm64 extensions, see [Creating INF Files for Multiple Platforms and Operating Systems](#).

## Specifying an AddPowerSetting String Entry Value

Except for *value-data* entries of type **REG\_SZ**, all the other string entry values that are supplied with an **AddPowerSetting** directive can be expressed as a string enclosed in double quotation marks ("quoted-string"), as

a %strkey% token that is defined in the INF string section of an INF file, or as language-neutral registry value.

Language-neutral registry values are used to support Windows Multilingual User Interface (MUI) and are specified as follows:

```
"@file-path,-resourceID[;comment]"
```

The entries that specify a language-neutral registry value are as follows:

*file-path*

The fully qualified path of the file that contains the resource.

*resourceID*

The resource ID of the corresponding resource. In the case of a string, the *resourceID* references a string. In the case of an icon, the *resourceID* references an icon.

*Comment*

An optional value that can be used to aid debugging or to provide an additional comment about the setting. In the case of a string resource, the power manager does not combine or display the comment string with specified resource string.

For more information about how to specify language-neutral registry values, see [Rendering Shell and Registry Strings](#).

## Examples

The following two examples define power settings that control the brightness of an LCD. The first example shows how to use the **Value** directive to define a minimum, a medium, and a maximum LCD brightness value.

```
// Within a DDinstall or ClassInstall23 section
AddPowerSetting=LCDDim
...
[LCDDim]
SubGroup = {7516B95F-F776-4464-8C53-06167F40CC99}
Setting = {381B4222-F694-41F0-9685-FF5BB260DF2E}, "LCD Brightness", "Controls the brightness of the LCD
display"

Value = 0, "Low", "Minimum Brightness", %FLG_ADDREG_TYPE_DWORD%, 0x50
Value = 1, "Medium", "Medium Brightness", %FLG_ADDREG_TYPE_DWORD%, 0x75
Value = 2, "High", "Maximum Brightness", %FLG_ADDREG_TYPE_DWORD%, 0x100

Default = %GUID_MAX_POWER_SAVINGS%, %AC%, 0
Default = %GUID_MAX_POWER_SAVINGS%, %DC%, 0
Default = %GUID_TYPICAL_POWER_SAVINGS%, %AC%, 2
Default = %GUID_TYPICAL_POWER_SAVINGS%, %DC%, 1
Default = %GUID_MIN_POWER_SAVINGS%, %AC%, 2
Default = %GUID_MIN_POWER_SAVINGS%, %DC%, 2
...

[Strings]
GUID_MAX_POWER_SAVINGS = {a1841308-3541-4fab-bc81-f71556f20b4a}
GUID_TYPICAL_POWER_SAVINGS = {381b4222-f694-41f0-9685-ff5bb260df2e}
GUID_MIN_POWER_SAVINGS = {8c5e7fda-e8bf-4a96-9a85-a6e23a8c635c}
AC = 0
DC = 1
FLG_ADDREG_TYPE_DWORD = 0x00010001
```

The second example shows how to use the **ValueRange** directive to define a range of allowed LCD brightness values that varies from 0% through 100%, with an increment of 1% between allowed values.

```
// Within a DDinstall or a ClassInstall32 section
AddPowerSetting=LCDDimRange
...

[LCDDimRange]
SubGroup = {7516B95F-F776-4464-8C53-06167F40CC99}
Setting = {381B4222-F694-41F0-9685-FF5BB260DF2E}, "LCD Brightness", "Controls the brightness of the LCD
display"
ValueRange = 0, 100, 1, "%"
Default = %GUID_MAX_POWER_SAVINGS%, %AC%, 50
Default = %GUID_MAX_POWER_SAVINGS%, %DC%, 50
Default = %GUID_TYPICAL_POWER_SAVINGS%, %AC%, 95
Default = %GUID_TYPICAL_POWER_SAVINGS%, %DC%, 50
Default = %GUID_MIN_POWER_SAVINGS%, %AC%, 100
Default = %GUID_MIN_POWER_SAVINGS%, %DC%, 100

[Strings]
GUID_MAX_POWER_SAVINGS = {a1841308-3541-4fab-bc81-f71556f20b4a}
GUID_TYPICAL_POWER_SAVINGS = {381b4222-f694-41f0-9685-ff5bb260df2e}
GUID_MIN_POWER_SAVINGS = {8c5e7fd-a8bf-4a96-9a85-a6e23a8c635c}
AC = 0
DC = 1
```

## See also

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.ColInstallers](#)

[DDInstall.HW](#)

[DDInstall.Interfaces](#)

[DDInstall.Services](#)

# INF AddProperty Directive

11/2/2020 • 4 minutes to read • [Edit Online](#)

An **AddProperty** directive references one or more INF file sections that modify the [device properties](#) that are set for a device instance, a [device setup class](#), a [device interface class](#), or a device interface.

```
[DDInstall] |
[DDInstall.nt] |
[DDInstall.ntx86] |
[DDInstall.ntia64] |
[DDInstall.ntamd64] |
[DDInstall.ntarm] |
[DDInstall.ntarm64] |
[ClassInstall32] |
[ClassInstall32.nt] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] | (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] | (Windows 10 and later versions of Windows)
[interface-install-section] |
[interface-install-section.nt] |
[interface-install-section.ntx86] |
[interface-install-section.ntia64] | (Windows XP and later versions of Windows)
[interface-install-section.ntamd64] (Windows XP and later versions of Windows) |
[interface-install-section.ntarm] (Windows 8 and later versions of Windows) |
[interface-install-section.ntarm64] (Windows 10 and later versions of Windows)
[add-interface-section]

AddProperty=add-property-section[,add-property-section]... (Windows Vista and later versions of Windows)
...
```

Each *add-property-section* can have entries to do the following:

- Add a device property and initialize the value of the property.
- Modify the value of an existing device property.

An *add-property-section* that is referenced by an **AddProperty** directive has the following format:

```
[add-property-section]
(property-name, , , [flags], value) |
({property-category-guid}, property-pid, type, [flags], value)
...
```

An add property section can have any number of *property-name* entries or *property-guid* entries, each on a separate line.

## Entries

*property-name*

One of the following property names that represent the device instance [driver package](#) properties:

- **DeviceModel**
- **DeviceVendorWebsite**
- **DeviceDetailedDescription**

- **DeviceDocumentationLink**
- **DeviceIcon**
- **DeviceBrandingIcon**

For more info about adding custom device icons, see [Providing Icons for a Device](#).

*property-category-guid*

A GUID value that identifies the property category. The GUID value can be a system-defined GUID that identifies one of the property categories for a device instance, a [device setup class](#), a [device interface class](#), or a device interface. All properties that have the same GUID value are members of the same category. These property categories are defined in *Devpkey.h*.

The GUID value can also be a custom GUID value that identifies a custom property category.

*property-pid*

The property identifier that indicates the specific property within the property category that is indicated by the *property-category-guid* value. For internal system reasons, a property identifier must be greater than or equal to two.

*type*

The numeric value, in decimal or hexadecimal format, of the [property-data-type identifier](#) for the property that is specified by the *property-category-guid* value and the *property-pid* value. Only the following [base data types](#) are supported:

- DEVPROP\_TYPE\_STRING
- DEVPROP\_TYPE\_STRING\_LIST
- DEVPROP\_TYPE\_BINARY
- DEVPROP\_TYPE\_BOOLEAN
- DEVPROP\_TYPE\_UINT32

For example, the decimal value of the DEVPROP\_TYPE\_STRING data type is 18 (0x00000012) and the decimal value of the DEVPROP\_TYPE\_STRING\_LIST data type is 2066 (0x00002012).

*flags*

An optional hexadecimal value that is a bitwise OR of the following flags that control the add operation:

**0x00000001 (FLG\_ADDPROPERTY\_NOCLOBBER)**

A flag that prevents the value entry value from replacing the existing property value. If a driver writer wants to make a property able to be overridden through **Include** and **Needs** directives, the writer must specify this flag for that property. This is because Windows processes the INF sections that are referenced by **Include** and **Needs** directives after Windows processes all other directives within the INF section that included the **Include** and **Needs** directives.

**0x00000002 (FLG\_ADDPROPERTY\_OVERWRITEONLY)**

A flag that sets the property value to the value entry value only if the specified property already exists.

**0x00000004 (FLG\_ADDPROPERTY\_APPEND)**

A flag that appends the value entry value to that of an existing property string value. This flag is valid only if the property data type is DEVPROP\_TYPE\_STRING\_LIST. The supplied string is not appended to an existing property string value if the supplied string is already present in the existing string value.

**0x00000008 (FLG\_ADDPROPERTY\_OR)**

A flag that performs a bitwise OR of the value entry value to that of the existing property value. This flag is valid only if the property data type is DEVPROP\_TYPE\_UINT32.

**0x00000010 (FLG\_ADDPROPERTY\_AND)**

A flag that performs a bitwise AND of the value entry value to that of the existing property value. This flag is

valid only if the property data type is DEVPROP\_TYPE\_UINT32.

#### *value*

The value that the add operation uses to modify a property value, depending on the property data type and the value of the *flags* entry.

## Remarks

The **AddProperty** directive can be used to modify a system-defined device property or a custom device property. This directive can be specified under any of the sections shown in the formal syntax statement above.

Each *add-property-section* name must be unique within an INF file, but the section can be referenced by more than one **AddProperty** directive in the same INF file. Each section name must follow the general rules for defining section names that are described in [General Syntax Rules for INF Files](#).

For more information about how to use the INF **AddProperty** directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

## Examples

The following example of an add property section includes two line entries: the first line entry sets the **DeviceModel** property by name, and the second line entry sets a custom device property by specifying a custom property key GUID.

The first line includes the *property-name* entry value "DeviceModel" and the *value* entry value "Sample Device Model Name."

The second line entry sets a custom property in a custom property category. The *property-category-guid* entry value is "c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e" and the *property-identifier* entry value is "2".

The optional *flags* entry value is not present, and the type entry value is "18" (DEVPROP\_TYPE\_STRING). The *value* entry value is "String value for property 1."

```
[SampleAddPropertySection]
DeviceModel,,,,"Sample Device Model Name"
{c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e}, 2, 18,, "String value for property 1"
```

## See also

[DelProperty](#)

# INF AddReg Directive

12/1/2020 • 11 minutes to read • [Edit Online](#)

An **AddReg** directive references one or more INF-writer-defined *add-registry-sections* that are used to modify or create registry information.

```
[DDInstall] |
[DDInstall.HW] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)

[install-interface-section] |
[service-install-section] |
[event-log-install] |
[add-interface-section]
AddReg=add-registry-section[,add-registry-section] ...
```

Each *add-registry section* can have entries to do the following:

- Add new keys, possibly with initial value entries, to the registry.
- Add new value entries to existing registry keys.
- Modify existing value entries of particular keys in the registry.

Each named *add-registry section* referenced by an **AddReg** directive has the following format:

```
[add-registry-section]
reg-root, [subkey],[value-entry-name],[flags],[value][,[value]]
reg-root, [subkey],[value-entry-name],[flags],[value][,[value]]
...
[[add-registry-section.security]
"security-descriptor-string"]
```

An *add-registry-section* can have any number of entries, each on a separate line. An INF can also contain one or more optional *add-registry-section.security* sections, each specifying a security descriptor that is applied to all registry values described within a named *add-registry-section*.

## Entries

*reg-root*

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

**HKCR**

Abbreviation for **HKEY\_CLASSES\_ROOT**

**HKCU**

Abbreviation for **HKEY\_CURRENT\_USER**

## HKLM

Abbreviation for **HKEY\_LOCAL\_MACHINE**

## HKU

Abbreviation for **HKEY\_USERS**

## HKR

Relative root, in which keys that are specified by using this abbreviation are relative to the registry key associated with the INF section in which this **AddReg** directive appears, as indicated in the following table.

INF SECTION CONTAINING ADDREG DIRECTIVE	REGISTRY KEY REFERENCED BY HKR
INF <b>DDInstall</b> section	The device's <i>software key</i>
INF <b>DDInstall.HW</b> section	The device's <i>hardware key</i>
INF <b>[service-install-section]</b> section	The <b>Services</b> key
INF <b>[event-log-install]</b> section	The <b>EventLog</b> key
INF <b>[add-interface-section]</b> section	The device interface's registry key

**Note** HKR cannot be used in an *add-registry-section* referenced from an **INF DefaultInstall section**.

For more information about driver information that is stored under the **HKEY\_LOCAL\_MACHINE** root, see [Registry Trees and Keys for Devices and Drivers](#).

### *subkey*

This optional value, formed either as a **%strkey%** token defined in a **Strings** section of the INF or as a registry path under the given *reg-root*(*key1\key2\key3...*), specifies one of the following:

- A new subkey to be added to the registry at the end of the given registry path.
- An existing subkey in which the additional values specified in this entry is written (possibly replacing the value of an existing named value entry of the given subkey).
- Both a new subkey to be added to the registry together with its initial value entry.

### *value-entry-name*

This optional value either names an existing value entry in the given (existing) *subkey* or creates the name of a new value entry to be added in the specified *subkey*, whether it already exists or is a new key to be added to the registry. This value can be expressed either as "quoted string" or as a **%strkey%** token that is defined in the INF's **Strings** section. (If this is omitted for a string-type value, the *value-entry-name* is the default "unnamed" value entry for this key.)

The operating system supports some system-defined special *value-entry-name* keywords. See the end of this **Remarks** section for more information.

### *flags*

This optional hexadecimal value, expressed as an ORed bitmask of system-defined low word and high word flag values, defines the data type for a value entry and/or controls the add-registry operation.

Bitmask values for each of these flags are as follows:

**0x00000001** (FLG\_ADDREG\_BINVALUETYPE)

The given value is "raw" data. (This value is identical to the FLG\_ADDREG\_TYPE\_BINARY.)

**0x00000002 (FLG\_ADDREG\_NOCLOBBER)**

Prevent a given value from replacing the value of an existing value entry.

**0x00000004 (FLG\_ADDREG\_DELVAL)**

Delete the given *subkey* from the registry, or delete the specified *value-entry-name* from the specified registry *subkey*.

**0x00000008 (FLG\_ADDREG\_APPEND)**

Append a given *value* to that of an existing named value entry. This flag is valid only if FLG\_ADDREG\_TYPE\_MULTI\_SZ is also set. The specified string value is not appended if it already exists.

**0x00000010 (FLG\_ADDREG\_KEYONLY)**

Create the given *subkey*, but ignore any supplied value-entry-name and/or value.

**0x00000020 (FLG\_ADDREG\_OVERWRITEONLY)**

Reset to the supplied *value* only if the specified *value-entry-name* already exists in the given *subkey*.

**0x00001000 (FLG\_ADDREG\_64BITKEY)**

(Windows XP and later versions of Windows.) Make the specified change in the 64-bit registry. If not specified, the change is made to the native registry.

**0x00002000 (FLG\_ADDREG\_KEYONLY\_COMMON)**

(Windows XP and later versions of Windows.) This is the same as FLG\_ADDREG\_KEYONLY but also works in a *del-registry-section* of an [INF DelReg directive](#).

**0x00004000 (FLG\_ADDREG\_32BITKEY)**

(Windows XP and later versions of Windows.) Make the specified change in the 32-bit registry. If not specified, the change is made to the native registry.

**0x00000000 (FLG\_ADDREG\_TYPE\_SZ)**

The given value entry and/or value is of type [REG\\_SZ](#).

**Note** This value is the default type for a specified value entry, so the flags value can be omitted from any *reg-root=* line in an *add-registry-section* that operates on a value entry of this type.

**0x00010000 (FLG\_ADDREG\_TYPE\_MULTI\_SZ)**

The given value entry and/or value is of the registry type [REG\\_MULTI\\_SZ](#). The value field that follows can be a list of strings separated by commas. This specification does not require any NULL terminator for a given string value.

**0x00020000 (FLG\_ADDREG\_TYPE\_EXPAND\_SZ)**

The given *value-entry-name* and/or *value* is of the registry type [REG\\_EXPAND\\_SZ](#).

**0x00010001 (FLG\_ADDREG\_TYPE\_DWORD)**

The given *value-entry-name* and/or *value* is of the registry type [REG\\_DWORD](#).

**0x00020001 (FLG\_ADDREG\_TYPE\_NONE)**

The given *value-entry-name* and/or *value* is of the registry type [REG\\_NONE](#).

*value*

This optionally specifies a new value for the specified *value-entry-name* to be added to the given registry key. Such a *value* can be a "replacement" value for an existing named value entry in an existing key, a value to be appended (*flag* value **0x00010008**) to an existing named [REG\\_MULTI\\_SZ](#)-type value entry in an existing key, a new value entry to be written into an existing key, or the initial value entry for a new *subkey* to be added to the registry.

The expression of such a *value* depends on the registry type specified for the *flag*, as follows:

- A registry string-type value can be expressed either as a "quoted string" or as a %strkey% token defined in a [Strings](#) section of the INF file. Such an INF-specified value does not have to include a NULL terminator at the end of each string.
- A registry numerical-type value can be expressed as a hexadecimal (by using 0x notation) or decimal number.

#### *security-descriptor-string*

Specifies a security descriptor, to be applied to all registry entries created by the named *add-registry-section*. The *security-descriptor-string* is a string with tokens to indicate the DACL (D:) security component.

If an *add-registry-section.security* section is not specified, registry entries inherit the security settings of the parent key.

If an *add-registry-section.security* section is specified, the following ACE's must be included so that installations and upgrades of devices and system service packs can occur:

- (A;;GA;;SY) – Grants all access to the local system.
- (A;;GA;;BA) – Grants all access to built-in administrators.

Do *not* specify ACE strings that grant write access to nonprivileged users.

For information about security descriptor strings, see [Security Descriptor Definition Language \(Windows\)](#). For information about the format of security descriptor strings, see [Security Descriptor Definition Language \(Windows\)](#).

For more information about how to specify security descriptors, see [Creating Secure Device Installations](#).

## Remarks

An AddReg directive can be specified under any of the sections shown in the formal syntax statement above. This directive can also be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the [AddService](#) directive in an [INF DDInstall.Services section](#).
- An *add-interface-section* referenced by the [AddInterface](#) directive in an [INF DDInstall.Interfaces section](#).
- An *install-interface-section* referenced in an [INF InterfaceInstall32 section](#).

Each *add-registry-section* name must be unique to the INF file, but it can be referenced by [AddReg](#) directives in other sections of the same INF. Each section name must follow the general rules for defining section names described in [General Syntax Rules for INF Files](#).

**Note** The lower-order bit of the low word in a flag value distinguishes between character and binary data.

To represent a number of a registry type other than one of the predefined REG\_XXXtypes, specify a new type number in the high word of the *flag* ORed with FLG\_ADDREG\_BINVALUETYPE in its low word. The data for such a *value* must be specified in binary format as a sequence of bytes separated by commas. For example, to store 16 bytes of data of a new registry data type, such as 0x38, as a value entry, the add-registry section entry would be something like the following:

```
HKR,,MYValue,0x00380001,1,0,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

This technique can be used to define new registry types for numeric values, but not for values of type [REG\\_EXPAND\\_SZ](#), [REG\\_MULTI\\_SZ](#), [REG\\_NONE](#), or [REG\\_SZ](#). For more info about these types, see [Registry value types](#).

### Special *value-entry-name* Keywords

Special keywords are defined for use in the HKR **AddReg** entries. The format for the entries that use these keywords is as follows:

```
[HKR,,DeviceCharacteristics,0x10001,characteristics]
[HKR,,DeviceType,0x10001,device-type]
[HKR,,Security,,security-descriptor-string]
[HKR,,UpperFilters,0x10000,service-name]
[HKR,,LowerFilters,0x10000,service-name]
[HKR,,Exclusive,0x10001,exclusive-device]
[HKR,,EnumPropPages32,"prop-provider.dll,provider-entry-point"]
[HKR,,LocationInformationOverride,"text-string"]
[HKR,,ResourcePickerTags,"text-string"]
[HKR,,ResourcePickerExceptions,"text-string"] ,
```

The following describes the HKR **AddReg** entries that use these special keywords:

#### DeviceCharacteristics

A **DeviceCharacteristics** HKR **AddReg** entry specifies characteristics for the device. The *characteristics* value is a numeric value that is the result of using OR on one or more FILE\_\* file characteristics values, which are defined in *Wdm.h* and *Ntddk.h*.

Only the following values can be specified in an INF:

```
#define FILE_REMOVABLE_MEDIA          0x00000001
#define FILE_READ_ONLY_DEVICE          0x00000002
#define FILE_FLOPPY_DISKETTE           0x00000004
#define FILE_WRITE_ONCE_MEDIA          0x00000008
#define FILE_DEVICE_SECURE_OPEN         0x00000100
```

For a description of these values, see [IoCreateDevice](#).

The characteristics values, which are specified by using a **DeviceCharacteristics** entry, are ORed with those specified in each call to **IoCreateDevice** that creates a device object on the device stack. The OR operation occurs after all device objects are added, but before the device is started.

The *characteristics* value (including a value of zero) overrides any class-wide device characteristics that were specified in the associated class installer INF.

For more information about device characteristics, see [Specifying Device Characteristics](#).

#### DeviceType

A **DeviceType** HKR **AddReg** entry specifies a device type for the device. The device-type is the numeric value of a FILE\_DEVICE\_XXX constant defined in *Wdm.h* or *Ntddk.h*. The flag value of 0x10001 specifies that the device-type value is a [REG\\_DWORD](#). For more information, see [Specifying Device Types](#).

A class-installer INF should specify the device type that applies to all, or almost all, of the devices in the class. For example, if the devices in the class are of type FILE\_DEVICE\_CD\_ROM, specify a *device-type* of 0x02. If a device INF specifies a value for **DeviceType**, it overrides the value set by the class installer, if any. If the class or device INF specifies a **DeviceType** value, the PnP manager

applies that type to the *physical device object (PDO)* created by the device's bus driver.

### Security

A **Security** HKR **AddReg** entry specifies a security descriptor for the device. The *security-descriptor-string* is a string with tokens to indicate the DACL (D:) security component.

A class-installer INF can specify a security descriptor for a device class. A device INF can specify a security descriptor for an individual device, overriding the security for the class. If the class and/or device INF specifies a *security-descriptor-string*, the PnP manager propagates the descriptor to all the device objects (DOs) for a device. This includes the function device object (FDO), optional *filter DOs*, and the PDO.

For information about the format of security descriptor strings, see the Microsoft Windows SDK documentation.

For more information about how to specify security descriptors, see [Creating Secure Device Installations](#).

### UpperFilters

An **UpperFilters** HKR **AddReg** entry specifies a PnP upper-filter driver. This entry in a **DDInstall.HW** section defines one or more device-specific upper-filter drivers. In a **ClassInstall32** section, this entry defines one or more class-wide upper-filter drivers.

### LowerFilters

A **LowerFilters** HKR **AddReg** entry specifies a PnP lower-filter driver. This entry in a **DDInstall.HW** section defines one or more device-specific lower-filter drivers. In a **ClassInstall32** section, this entry defines one or more class-wide lower-filter drivers.

### Exclusive

An **Exclusive** HKR **AddReg** entry, if it exists and is set to "1", specifies that the device is an *exclusive device*. Otherwise the device is not treated as exclusive. For more information, see [Specifying Exclusive Access to Device Objects](#).

### EnumPropPages32

An **EnumPropPages32** HKR **AddReg** entry specifies the name of a dynamic-link library (DLL) file that is a device-specific property page provider. It also specifies the name of the **ExtensionPropSheetPageProc** callback function as implemented by the DLL. For more information about property pages and functions, see the Microsoft Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 4.0.

**Important** Both the name of the DLL and **ExtensionPropSheetPageProc** callback function must be enclosed together within quotation marks ("").

### LocationInformationOverride

(Windows XP and later versions of Windows) A **LocationInformationOverride** HKR **AddReg** entry can be used to specify a text string that describes a device's physical location. It overrides the **LocationInformation** string that the device's bus driver supplies in response to an [IRP\\_MN\\_QUERY\\_DEVICE\\_TEXT](#) request.

### ResourcePickerTags

A **ResourcePickerTags** HKR **AddReg** entry specifies resource picker tags for a device.

### ResourcePickerExceptions

A **ResourcePickerExceptions** HKR **AddReg** entry specifies the resource conflicts that are allowed for a device.

## Examples

An **AddReg** directive referenced the (SCSI) Miniport\_EventLog\_AddReg section in this example, under an INF-writer-defined section referenced by the **AddService** directive in a *DDInstall.Services* section of this INF.

```
[Miniport_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%SystemRoot%\System32\IoLogMsg.dll"
; double quotation marks delimiters in preceding entry prevent truncation
; if line wraps

HKR,,TypesSupported,0x00010001,7
```

Note that you can either specify flag values in hexadecimal format, as shown in the example, or you can define string placeholders such as `%FLG_ADDREG_TYPE_DWORD%` in the [Strings] section of each INF file.

## See also

[AddInterface](#)

[AddService](#)

[BitReg](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.ColInstallers](#)

[DDInstall.HW](#)

[DDInstall.Interfaces](#)

[DDInstall.Services](#)

[DelReg](#)

[InterfaceInstall32](#)

[Strings](#)

# INF AddService Directive

11/2/2020 • 16 minutes to read • [Edit Online](#)

**Note** This directive is not used in INF files that install devices that do not require any drivers, such as modems or display monitors.

An **AddService** directive is used within an **INF DDInstall.Services section** or **INF DefaultInstall.Services section**. It specifies characteristics of the services associated with drivers, such as how and when the services are loaded, and any dependencies on other underlying legacy drivers or services. Optionally, this directive also sets up event-logging services for the device.

```
[DDInstall.Services]  
AddService=ServiceName,[flags],service-install-section  
    [,event-log-install-section[,,[EventLogType][,,EventName]]]  
...
```

## Entries

### *ServiceName*

Specifies the name of the service to be installed. For a device, this value is usually a generic name for its driver, such as "sermouse," or some such name. This name must not be localized. It must be the same regardless of the system's local language.

### *flags*

Specifies one or more (ORed) of the following system-defined flags, defined in *Setupapi.h*, expressed as a hexadecimal value:

#### 0x00000001 (SPSVCINST\_TAGTOFRONT)

Move the named service's tag to the front of its group order list, thereby ensuring that it is loaded first within that group (unless a subsequently installed device with this INF specification displaces it). INF files that install exclusively PnP devices and devices with WDM drivers should not set this flag.

#### 0x00000002 (SPSVCINST\_ASSOCSERVICE)

Assign the named service as the PnP function driver (or legacy driver) for the device being installed by this INF file.

To indicate that a service is the function driver for a device, the service should specify the **SPSVCINST\_ASSOCSERVICE** flag in the **AddService** directive. For a service such as a filter driver or other driver component, the flag should not be used.

Every device driver INF should have exactly one associated service. The INF does not require an associated service if it is an Extension or uses the Include/Needs directives to inherit the associated service from another INF. For devices that do not require a function driver, the NULL driver can be specified as follows:

```
AddService = ,2.
```

#### 0x00000008 (SPSVCINST\_NOCLOBBER\_DISPLAYNAME)

Do not overwrite the given service's (optional) friendly name if this service already exists in the system.

#### 0x00000010 (SPSVCINST\_NOCLOBBER\_STARTTYPE)

Do not overwrite the given service's start type if this named service already exists in the system.

**0x00000020** (SPSVCINST\_NOCLOBBER\_ERRORCONTROL)

Do not overwrite the given service's error-control value if this named service already exists in the system.

**0x00000040** (SPSVCINST\_NOCLOBBER\_LOADORDERGROUP)

Do not overwrite the given service's load-order-group value if this named service already exists in the system. INF files that install exclusively PnP devices and devices with WDM drivers should not set this flag.

**0x00000080** (SPSVCINST\_NOCLOBBER\_DEPENDENCIES)

Do not overwrite the given service's dependencies list if this named service already exists in the system. INF files that install exclusively PnP devices and devices with WDM drivers should not set this flag.

**0x00000100** (SPSVCINST\_NOCLOBBER\_DESCRIPTION)

Do not overwrite the given service's (optional) description if this service already exists in the system.

**0x00000400** (SPSVCINST\_CLOBBER\_SECURITY) (Windows XP and later versions of Windows)

Overwrite the security settings for the service if this service already exists in the system.

**0x00000800** (SPSVCSINST\_STARTSERVICE) (Windows Vista and later versions of Windows)

Start the service after the service is installed. This flag cannot be used to start a service that implements a Plug and Play (PnP) function driver or filter driver for a device. Otherwise, this flag can be used to start a user-mode or kernel-mode service that is managed by the Service Control Manager (SCM).

**0x00001000** (SPSVCINST\_NOCLOBBER\_REQUIREDPRIVILEGES) (Windows 7 and later versions of Windows)

Do not overwrite the privileges for the given service if this service already exists in the system.

#### *service-install-section*

References an INF-writer-defined section that contains information for installing the named service for this device (or devices). For more information, see the following **Remarks** section.

#### *event-log-install-section*

Optionally references an INF-writer-defined section in which event-logging services for this device (or devices) are set up.

#### *EventLogType*

Optionally specifies one of **System**, **Security**, or **Application**. If omitted, this defaults to **System**, which is almost always the appropriate value for the installation of device drivers.

For example, an INF would specify **Security** only if the to-be-installed driver provides its own security support.

#### *EventName*

Optionally specifies a name to use for the event log. If omitted, this defaults to the given *ServiceName*.

## Remarks

The system-defined and case-insensitive extensions can be inserted into a **DD\Install\Services** section that contains an **AddService** directive in cross-operating system and/or cross-platform INF files to specify platform-specific or OS-specific installations.

Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

An **AddService** directive must reference a named *service-install-section* elsewhere in the INF file. Each such section has the following form:

```
[service-install-section]

[DisplayName=name]
[Description=description-string]
ServiceType=type-code
StartType=start-code
ErrorControl=error-control-level
ServiceBinary=path-to-service
[StartName=driver-object-name]
[AddReg=add-registry-section[, add-registry-section] ...]
[DelReg=del-registry-section[, del-registry-section] ...]
[BitReg=bit-registry-section[, bit-registry-section] ...]
[LoadOrderGroup=load-order-group-name]
[Dependencies=depend-on-item-name[,depend-on-item-name]]
[Security="security-descriptor-string"]...
[ServiceSidType=value]
[DelayedAutoStart=true/false]
[AddTrigger=service-trigger-install-section[, service-trigger-install-section, ...]]
```

Each *service-install-section* must have at least the **ServiceType**, **StartType**, **ErrorControl**, and **ServiceBinary** entries as shown here. However, the remaining entries are optional.

### Service-Install Section Entries and Values

#### **DisplayName**=*name*

Specifies a friendly name for the service/driver, usually, for ease of localization, expressed as a %strkey% token defined in a **Strings** section of the INF file.

#### **Description**=*description-string*

Optionally specifies a string that describes the service, usually expressed as a %strkey% token defined in a **Strings** section of the INF file.

This string gives the user more information about the service than the **DisplayName**. For example, the **DisplayName** might be something like "DHCP Client" and the Description might be something like "Manages network configuration by registering and updating IP addresses and DNS names".

The *description-string* should be long enough to be descriptive but not so long as to be awkward. If a *description-string* contains any %strkey% tokens, each token can represent a maximum of 511 characters. The total string, after any string token substitutions, should not exceed 1024 characters.

#### **ServiceType**=*type-code*

The type-code for a kernel-mode device driver must be set to 0x00000001 (SERVICE\_KERNEL\_DRIVER).

The *type-code* for a Microsoft Win32 service that is installed for a device should be set to 0x00000010 (SERVICE\_WIN32\_OWN\_PROCESS) or 0x00000020 (SERVICE\_WIN32\_SHARE\_PROCESS). If the Win32 service can interact with the desktop, the type-code value should be combined with 0x00000100 (SERVICE\_INTERACTIVE\_PROCESS).

The *type-code* for a highest level network driver, such as a redirector, or a file system driver, should be set to 0x00000002 (SERVICE\_FILE\_SYSTEM\_DRIVER).

The SERVICE\_xxxx constants are defined in *Wdm.h* and *Ntddk.h*.

#### **StartType**=*start-code*

Specifies when to start the driver as one of the following numeric values, expressed either in decimal or, as shown in the following list, in hexadecimal notation.

#### 0x0 (SERVICE\_BOOT\_START)

Indicates a driver started by the operating system loader.

This value must be used for drivers of devices required for loading the operating system.

## 0x1 (SERVICE\_SYSTEM\_START)

Indicates a driver started during operating system initialization.

This value should be used by PnP drivers that do device detection during initialization but are not required to load the system.

For example, a PnP driver that can also detect a legacy device should specify this value in its INF so that its [DriverEntry](#) routine is called to find the legacy device, even if that device cannot be enumerated by the PnP manager.

## 0x2 (SERVICE\_AUTO\_START)

Indicates a driver started by the service control manager during system startup.

This value should never be used in the INF files for WDM or PnP device drivers.

## 0x3 (SERVICE\_DEMAND\_START)

Indicates a driver started on demand, either by the PnP manager when the corresponding device is enumerated or possibly by the service control manager in response to an explicit user demand for a non-PnP device.

This value should be used in the INF files for all WDM drivers of devices that are not required to load the system and for all PnP device drivers that are neither required to load the system nor engaged in device detection.

## 0x4 (SERVICE\_DISABLED)

Indicates a driver that cannot be started.

This value can be used to temporarily disable the driver services for a device. However, a device/driver cannot be installed if this value is specified in the service-install section of its INF file.

For more information about **StartType**, see [Specifying Driver Load Order](#).

### **ErrorControl=error-control-level**

Specifies the level of error control as one of the following numeric values, expressed either in decimal or, as shown in the following list, in hexadecimal notation.

## 0x0 (SERVICE\_ERROR\_IGNORE)

If the driver fails to load or initialize, proceed with system startup and do not display a warning to the user.

## 0x1 (SERVICE\_ERROR\_NORMAL)

If the driver fails to load or initialize its device, system startup should proceed but display a warning to the user.

## 0x2 (SERVICE\_ERROR\_SEVERE)

If the driver fails to load, system startup should switch to the registry's **LastKnownGood** control set and continue system startup, even if the driver again indicates a loading or device/driver initialization error.

## 0x3 (SERVICE\_ERROR\_CRITICAL)

If the driver fails to load and system startup is not using the registry's **LastKnownGood** control set, switch to **LastKnownGood** and try again.

If startup still fails when using **LastKnownGood**, run a bug-check routine. (*Only* devices/drivers necessary for the system to boot specify this value in their INF files.)

### **ServiceBinary=path-to-service**

Specifies the path of the binary for the service, expressed as %dirid%\filename.

The *dirid* number is either a custom directory identifier or one of the system-defined directory identifiers described in [Using Dirids](#). The given *filename* specifies a file already transferred (see the [INF CopyFiles](#)

**Directive**) from the source distribution media to that directory on the target computer.

**StartName=***driver-object-name*

This optional entry specifies the name of the driver object that represents this device/driver. If *type-code* specifies 1 (SERVICE\_KERNEL\_DRIVER) or 2 (SERVICE\_FILE\_SYSTEM\_DRIVER), this name is the driver object name that the I/O manager uses to load the driver.

**AddReg=***add-registry-section[, add-registry-section]...*

References one or more INF-writer-defined *add-registry-sections* in which any registry information pertinent to the newly installed services is set up. An HKR specification in such an *add-registry-section* designates the HKLM\System\CurrentControlSet\Services\ServiceName registry key. For more information, see [INF AddReg Directive](#).

This directive is rarely used in a service-install-section.

**DelReg=***del-registry-section[, del-registry-section]...*

References one or more INF-writer-defined *del-registry-sections* in which pertinent registry information for an already installed services is removed. An HKR specification in such a *del-registry-section* designates the HKLM\System\CurrentControlSet\Services\ServiceName registry key. For more information, see [INF DelReg Directive](#).

This directive is almost never used in a *service-install-section*, but it might be used in an INF that "updates" the registry for a previous installation of the same device/driver services.

**BitReg=***bit-registry-section[, bit-registry-section]...*

Is valid in a *service-install-section* but almost never used. An HKR specification in such a *bit-registry-section* also designates the HKLM\System\CurrentControlSet\Services\ServiceName registry key.

**LoadOrderGroup=***load-order-group-name*

This optional entry identifies the load order group of which this driver is a member. It can be one of the "standard" load order groups, such as SCSI class or NDIS.

In general, this entry is unnecessary for devices with WDM drivers or for exclusively PnP devices, unless there are legacy dependencies on such a group. However, this entry can be useful if device detection is supported by loading a group of drivers in a particular order.

For more information about LoadOrderGroup, see [Specifying Driver Load Order](#).

**Dependencies=***depend-on-item-name[, depend-on-item-name]...*

Each *depend-on-item-name* item in a dependencies list specifies the name of a service or load-order group on which the device/driver depends.

If the *depend-on-item-name* specifies a service, the service that must be running before this driver is started. For example, the INF for the system-supplied Win32 TCP/IP print services depends on the support of the underlying (kernel-mode) TCP/IP transport stack. Consequently, the INF for the TCP/IP print services specifies this entry as **Dependencies=TCPIP**.

A *depend-on-item-name* can specify a load order group on which this device/driver depends. Such a driver is started only if at least one member of the specified group was started. Precede the group name with a plus sign (+). For example, the system RAS services INF might have an entry like **Dependencies = +NetBIOSGroup,RpcSS** that lists both a load-order group and a service.

**Security=**"*security-descriptor-string*"

Specifies a security descriptor, to be applied to the service. This security descriptor specifies the permissions that are required to perform such operations as starting, stopping, and configuring the service. The *security-descriptor-string* value is a string with tokens to indicate the DACL (D:) security component.

For information about security descriptor strings, see [Security Descriptor Definition Language \(Windows\)](#).

For information about the format of security descriptor strings, see Security Descriptor Definition Language (Windows).

For more information about how to specify security descriptors, see [Creating Secure Device Installations](#).

**ServiceSidType=***value*

**Note:** This value can only be used for *Win32 Services* and is only available with Windows 10 Version 2004 and above.

This entry can use any valid value as described in [SERVICE\\_SID\\_INFO](#).

**DelayedAutoStart=***true/false*

**Note:** This value can only be used for *Win32 Services* and is only available with Windows 10 2004 and above.

Contains the delayed auto-start setting of an auto-start service.

If this member is TRUE, the service is started after other auto-start services are started plus a short delay. Otherwise, the service is started during system boot.

This setting is ignored unless the service is an auto-start service.

For more information, see [this page](#).

**AddTrigger=***service-trigger-install-section [, service-trigger-install-section, ...]*

Specifies the trigger events to be registered for the Win32 service so that the service can be started or stopped when a trigger event occurs. For more information about service trigger events, see [Service Trigger Events](#).

Each named service-trigger-install section referenced by an AddTrigger directive has the following format:

```
[service-trigger-install-section]

TriggerType=trigger-type
Action=action-type
SubType=trigger-subtype
[DataItem=data-type,data]
...
```

**Service-Trigger-Install Section Entries and Values:**

**TriggerType=***trigger-type*

Specifies the service trigger event type in one of the following numeric values, expressed either in decimal or, as is shown in the following list, hexadecimal notation:

0x1 (SERVICE\_TRIGGER\_TYPE\_DEVICE\_INTERFACE\_ARRIVAL) Indicates that event is triggered when a device of the specified device interface class arrives or is present when the system starts.

For more information, see [SERVICE\\_TRIGGER structure](#)

**Action=***action-type*

Specifies the action to take when the specified trigger event occurs.

0x1 (SERVICE\_TRIGGER\_ACTION\_SERVICE\_START) starts the service when the specified trigger event occurs.

0x2 (SERVICE\_TRIGGER\_ACTION\_SERVICE\_STOP) stops the service when the specified trigger event occurs.

For more information, see [SERVICE\\_TRIGGER structure](#)

#### **SubType**=*trigger-subtype*

Specifies a GUID that identifies the trigger event subtype. The value depends on the value of the **TriggerType**.

When **TriggerType** is **0x1** (SERVICE\_TRIGGER\_TYPE\_DEVICE\_INTERFACE\_ARRIVAL), **SubType** specifies the GUID that identifies the device interface class.

For more information, see [SERVICE\\_TRIGGER structure](#).

#### **DataItem**=*data-type, data*

Optionally specifies the trigger-specific data for a service trigger event.

When **TriggerType** is **0x1** (SERVICE\_TRIGGER\_TYPE\_DEVICE\_INTERFACE\_ARRIVAL), an optional **DataItem** may be specified with a data-type of **0x2** (SERVICE\_TRIGGER\_DATA\_TYPE\_STRING) to scope the device interface class to a specific hardware ID or compatible ID.

For more information, see [SERVICE\\_TRIGGER\\_SPECIFIC\\_DATA\\_ITEM structure](#)

The best practice for using the **AddTrigger** directive is to trigger start the service on device interface arrival. For more information, see [Win32 Services Interacting with Devices](#).

**Note:** **AddTrigger** syntax is only available in **Windows 10 Version 2004** and forward.

### **Specifying Driver Load Order**

The operating system loads drivers according to the *service-install-section* **StartType** value, as follows:

- During the system boot start phase, the operating system loads all **0x0** (SERVICE\_BOOT\_START) drivers.
- During the system start phase, the operating system first loads all WDM and PnP drivers for which the PnP manager finds device nodes (*devnodes*) in the registry ..\Enum tree (whether their INF files specify **0x01** for SERVICE\_SYSTEM\_START or **0x03** for SERVICE\_DEMAND\_START). Then the operating system loads all remaining SERVICE\_SYSTEM\_START drivers.
- During the system auto-start phase, the operating system loads all remaining SERVICE\_AUTO\_START drivers.

For more information about **Dependencies**, see [Specifying Driver Load Order](#).

### **Promoting a Driver's StartType at Boot Depending on Boot Scenario**

Depending on the boot scenario, you can use the **BootFlags** registry value to control when the operating system should promote a driver's **StartType** value to **0x0** (SERVICE\_BOOT\_START). You can specify one or more (ORed) of the following numeric values, expressed as a hexadecimal value:

- **0x1** (CM\_SERVICE\_NETWORK\_BOOT\_LOAD) indicates the driver should be promoted if booting from the network.
- **0x2** (CM\_SERVICE\_VIRTUAL\_DISK\_BOOT\_LOAD) indicates the driver should be promoted if booting from a VHD.
- **0x4** (CM\_SERVICE\_USB\_DISK\_BOOT\_LOAD) indicates the driver should be promoted to if booting from a USB disk.
- **0x8** (CM\_SERVICE\_SD\_DISK\_BOOT\_LOAD) indicates the driver should be promoted if booting from SD storage.
- **0x10** (CM\_SERVICE\_USB3\_DISK\_BOOT\_LOAD) indicates the driver should be promoted if booting from a disk on a USB 3.0 controller.

- 0x20 (CM\_SERVICE\_MEASURED\_BOOT\_LOAD) indicates the driver should be promoted if booting while measured boot is enabled.
- 0x40 (CM\_SERVICE\_VERIFIER\_BOOT\_LOAD) indicates the driver should be promoted if booting with verifier boot enabled.
- 0x80 (CM\_SERVICE\_WINPE\_BOOT\_LOAD) indicates the driver should be promoted if booting to WinPE.

The *service-install-section* has the following general form:

```
[service-install-section]
AddReg=add-registry-section
...
[add-registry-section]
HKR,,BootFlags,0x00010003,0x14 ; CM_SERVICE_USB3_DISK_BOOT_LOAD|CM_SERVICE_USB_DISK_BOOT_LOAD
```

## Registering for Event Logging

An **AddService** directive can also reference an *event-log-install-section* elsewhere in the INF file. Each such section has the following form:

```
[event-log-install-section]
AddReg=add-registry-section[, add-registry-section]...
[DelReg=del-registry-section[, del-registry-section]...]
[BitReg=bit-registry-section[, bit-registry-section]...]
...
```

For a typical device/driver INF file, the *event-log-install-section* uses only the **AddReg** directive to set up an event-logging message file for the driver. An **HKR** specification in an *add-registry-section* designates the **HKLM\System\CurrentControlSet\Services\EventLog\EventLogType\EventName** registry key. This event-logging *add-registry-section* has the following general form:

```
[drivername_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"path\IoLogMsg.dll;path\driver.sys"
HKR,,TypesSupported,0x00010001,7
```

In particular, the section adds two value entries in the registry subkey created for the device/driver, as follows:

- The value entry named **EventMessageFile** is of type **REG\_EXPAND\_SZ**, as specified by the **FLG\_ADDREG\_TYPE\_EXPAND\_SZ** value **0x00020000**. Its value, enclosed in double quotation marks ("), associates the system-supplied *IoLogMsg.dll* (but it could associate another logging DLL) with the driver binary file. Usually, the paths to each of these files are specified as follows:

*%%SystemRoot%%\System32\IoLogMsg.dll*

*%%SystemRoot%%\System32\drivers\driver.sys*

- The value entry named **TypesSupported** is of type **REG\_DWORD**, as specified by the **FLG\_ADDREG\_TYPE\_DWORD** value **0x00010001**.

For drivers, this value should be 7. This value is equivalent to the bitwise OR of **EVENTLOG\_SUCCESS**, **EVENTLOG\_ERROR\_TYPE**, **EVENTLOG\_WARNING\_TYPE**, and **EVENTLOG\_INFORMATION\_TYPE**, without setting the **EVENTLOG\_AUDIT\_XXX** bits.

An *event-log-install-section* can also use the [DelReg](#) directive to remove a previously installed event-log message file, by explicitly deleting the existing **EventMessageFile** and **TypesSupported** value entries, if a driver binary is being superseded by a newly installed driver. (See also [INF DelService Directive](#).)

While a [BitReg](#) directive is also valid within an INF-writer-defined *event-log-install-section*, it is almost never used, because the standard value entries for device driver event logging are not bitmasks.

## Examples

This example shows the service-install and event-log-install sections referenced by the **AddService** directive as already shown earlier in the example for [DDInstall.Services](#).

```
[sermouse_Service_Inst]
DisplayName      = %sermouse.SvcDesc%
ServiceType     = 1                      ; = SERVICE_KERNEL_DRIVER
StartType       = 3                      ; = SERVICE_DEMAND_START
ErrorControl    = 1                      ; = SERVICE_ERROR_NORMAL
ServiceBinary   = %12%\sermouse.sys
LoadOrderGroup  = Pointer Port

[sermouse_EventLog_Inst]
AddReg = sermouse_EventLog_AddReg

[sermouse_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%SystemRoot%\System32\IoLogMsg.dll;
    %%SystemRoot%\System32\drivers\sermouse.sys"
;
; Preceding entry on single line in INF file. Enclosing quotation marks
; prevent the semicolon from being interpreted as a comment.
;
HKR,,TypesSupported,0x00010001,7

[mouclass_Service_Inst]
DisplayName      = %mouclass.SvcDesc%
ServiceType     = 1                      ; = SERVICE_KERNEL_DRIVER
StartType       = 1                      ; = SERVICE_SYSTEM_START
ErrorControl    = 1                      ; = SERVICE_ERROR_NORMAL
ServiceBinary   = %12%\mouclass.sys
LoadOrderGroup  = Pointer Class

[mouclass_EventLog_Inst]
AddReg = mouclass_EventLog_AddReg

[mouclass_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%SystemRoot%\System32\IoLogMsg.dll;
    %%SystemRoot%\System32\drivers\mouclass.sys"
HKR,,TypesSupported,0x00010001,7
;
[Strings]
;
sermouse.SvcDesc = "Serial Mouse Driver"
mouclass.SvcDesc = "Mouse Class Driver"
```

The example in the reference for the [DDInstall.HW](#) section, described earlier, also shows some service-install sections referenced by the **AddService** directive to set up PnP upper-filter drivers.

## See also

[AddReg](#)

[BitReg](#)

[CopyFiles](#)

*DDInstall.HW*

*DDInstall.Services*

**DelReg**

**DelService**

**DestinationDirs**

**Strings**

# INF AddSoftware Directive

11/2/2020 • 5 minutes to read • [Edit Online](#)

Each **AddSoftware** directive describes the installation of standalone software. Use this directive in an INF file of the **SoftwareComponent** setup class. For more details on software components, see [Using a Component INF File](#). This directive is supported for Windows 10 version 1703 and later.

Valid installation types depend on the [target platform](#). For example, Desktop supports MSI installers and setup EXEs. **Note:** Type 2 is supported in Universal Drivers, Type 1 is desktop-only.

When a software component INF file specifies **AddSoftware**, the system queues software to be installed after device installation. There is no guarantee when or if the software will be installed. If referenced software fails to install, the system tries again when the referencing software component is updated.

An **AddSoftware** directive is used within an [INF DDInstall.Software](#) section.

```
[DDInstall.Software]
AddSoftware=SoftwareName,[flags],software-install-section
```

## Entries

### *SoftwareName*

Specifies the name of the software to be installed. This name uniquely identifies the software. The processing of an **AddSoftware** directive checks the version against previous software installed with the same name by an **AddSoftware** directive from any driver package. We recommend prefacing the SoftwareName with the vendor name, for example `ContosoControlPanel`.

### *flags*

Specifies one or more (ORed) flags.

### `0x00000000`

The **AddSoftware** directive is processed only once.

### `0x00000001`

The **AddSoftware** directive is processed once for each component device that specifies **AddSoftware** with the same unique *SoftwareName*.

### *software-install-section*

References an INF-writer-defined section that contains information for installing software.

## Remarks

Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

An **AddSoftware** directive must reference a named *software-install-section* elsewhere in the INF file. Each such section has the following form:

```
[software-install-section]

SoftwareType=type-code
[SoftwareBinary=path-to-binary]
[SoftwareArguments=argument[, argument] ...]
[SoftwareVersion=w.x.y.z]
[SoftwareID=pfn://x.y.z]
```

#### NOTE

See [SoftwareType](#) for information about constraints on section entries and values.

Any software installed using **AddSoftware** must be installed silently (or quietly). In other words, no user interface can be shown to the user during installation.

Any software installed using **AddSoftware** will **not** be uninstalled if the virtual software component device or its parent devices are uninstalled. If your software is not a UWP app (i.e. you're using **AddSoftware** with a value of 1), please make sure users can easily uninstall it without leaving a trace in the registry. To do so:

- If you're using an MSI installer, set up an [Add/Remove Programs](#) entry in the application's Windows Installer package.
- If you're using a custom EXE that installs global registry/file state (instead of supplementing local device settings), use the [Uninstall Registry Key](#).

## [software-install-section]: SoftwareType

```
SoftwareType={type-code}
```

**SoftwareType** specifies the type of software installation and is a required entry.

A value of 1 indicates that the associated software is an MSI or EXE binary. When this value is set, the **SoftwareBinary** entry is also required. A value of 1 is not supported on Windows 10 S.

If **SoftwareType** is set to 1, **SoftwareBinary** and **SoftwareVersion** are also required, but **SoftwareArguments** and flags (in the **AddSoftware** directive) are optional.

Starting in Windows 10 version 1709, a value of 2 indicates that the associated software is a Microsoft Store link. Use a value of 1 only for device-specific software that has no graphical user interface. If you have a device-specific app with graphical elements, it should come from the Microsoft Store, and the driver should reference it using **SoftwareType** 2.

If **SoftwareType** is set to 2, **SoftwareID** is required, and flags (in the **AddSoftware** directive) are optional. If **SoftwareType** is set to 2, **SoftwareBinary** and **SoftwareVersion** are not used.

#### NOTE

When using Type 2 of the **AddSoftware** directive, it is not required to utilize a Component INF. The directive can be used in any INF successfully. An **AddSoftware** directive of Type 1, however, must be used from a Component INF.

Do not use **AddSoftware** to distribute software that is unrelated to a device. For example, an OEM-specific PC utility program should not be installed with **AddSoftware**. Instead, use one of the following options to preinstall an app in an OEM image of Windows 10:

- To preinstall a Win32 app, boot to audit mode and install the app. For details, see [Audit Mode Overview](#).
- To preinstall a Microsoft Store (UWP) app, see [Preinstallable apps for desktop devices](#)

For info about pairing a driver with a Universal Windows Platform (UWP) app, see [Pairing a driver with a Universal Windows Platform \(UWP\) app](#) and [Hardware Support App \(HSA\): Steps for Driver Developers](#).

## [software-install-section]: SoftwareBinary

```
SoftwareBinary={filename}
```

Specifies the path to the executable. The system generates command lines like the following:

```
MSI: msieexec /i "<SoftwareBinary>" ALLUSERS=1 /quiet /qn /promptrestart [<SoftwareArguments>]
```

```
EXE: <SoftwareBinary> [<SoftwareArguments>]
```

If you use this entry, you must add the executable to the DriverStore by specifying the [INF CopyFiles Directive](#) with a **DestinationDirs** value of 13.

### NOTE

See [SoftwareType](#) for information about constraints on **SoftwareBinary** entries and values.

## [software-install-section]: SoftwareArguments

```
SoftwareArguments={argument1[, argument2[, ... argumentN]]}
```

Specifies extension-specific arguments to append to the command line. You can specify command line arguments that the system simply passes through into the generated command line. You can also specify special strings called *runtime context variables*. When you specify a runtime context variable, the system converts it into a device-specific value before appending it to the generated command line. You can mix and match literal string arguments with runtime context variables. Supported runtime context variables are:

```
<<DeviceInstanceID>>
```

The system replaces the string above with the device instance ID of the software component.

For example:

```
[DDInstall.Software]
AddSoftware=ContosoControlPanel,,Contoso_ControlPanel_Software

[Contoso_ControlPanel_Software]
SoftwareType=1
SoftwareBinary=ContosoControlPanel.exe
SoftwareArguments=<<DeviceInstanceID>>
SoftwareVersion=1.0.0.0
```

The above example results in a command line like this:

```
<DriverStorePath>\ContosoControlPanel.exe PCI\VEN_0000&DEV_0001&SUBSYS_00000000&REV_00\0123
```

If SoftwareArguments contains multiple arguments:

```
SoftwareArguments=arg1,<<DeviceInstanceID>>,arg2
```

The above results in:

```
<DriverStorePath>\ContosoControlPanel.exe arg1 PCI\VEN_0000&DEV_0001&SUBSYS_00000000&REV_00\0123 arg2
```

**NOTE**

See [SoftwareType](#) for information about constraints on **SoftwareArguments** entries and values.

## [software-install-section]: SoftwareVersion

`SoftwareVersion={w.x.y.z}`

Specifies the software version. Each value should not exceed 65535. When the system encounters a duplicate **SoftwareName**, it checks the **SoftwareVersion** against the previous **SoftwareVersion**. If it is greater, Windows runs the software.

**NOTE**

See [SoftwareType](#) for information about constraints on **SoftwareVersion** entries and values.

## [software-install-section]: SoftwareID

`SoftwareID={x.y.z}`

Specifies a Microsoft Store identifier and identifier type. Currently, only Package Family Name (PFN) is supported. Use a PFN to reference a Universal Windows Platform (UWP) app using the form `pfn://<x.y.z>`.

**NOTE**

See [SoftwareType](#) for information about constraints on **SoftwareID** entries and values.

## See Also

- [Using a Component INF File.](#)
- [INF DDInstall.Softare Section](#)
- [INF AddComponent Directive](#)
- [Pairing a driver with a Universal Windows Platform \(UWP\) app](#)
- [Hardware Support App \(HSA\): Steps for Driver Developers](#)

# INF BitReg Directive

12/1/2020 • 4 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **BitReg** directive references one or more INF-writer-defined sections used to set or clear bits within an existing **REG\_BINARY**-type value entry in the registry. However, this directive is very rarely used in device/driver INF files.

```
[DDInstall] |
[DDInstall.HW] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)

BitReg=bit-registry-section[,bit-registry-section]....
```

A **BitReg** directive can be specified under any of the sections shown in the formal syntax statement above. This directive can also be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the **AddService** directive in a **DDInstall.Services** section.
- An *add-interface-section* referenced by the **AddInterface** directive in a **DDInstall.Interfaces** section.
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Each named section referenced by a **BitReg** directive has the following form:

```
[bit-registry-section]
reg-root, [subkey], value-entry-name, [flags], byte-mask, byte-to-modify
reg-root, [subkey], value-entry-name, [flags], byte-mask, byte-to-modify
...
```

A *bit-registry-section* can have any number of entries, each on a separate line.

## Entries

### *reg-root*

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

#### **HKCR**

Abbreviation for **HKEY\_CLASSES\_ROOT**.

#### **HKCU**

Abbreviation for **HKEY\_CURRENT\_USER**.

#### **HKLM**

Abbreviation for **HKEY\_LOCAL\_MACHINE**.

**HKU**

Abbreviation for **HKEY\_USERS**.

**HKR**

Relative root – that is, keys that are specified by using this abbreviation are relative to the registry key associated with the INF section in which this **BitReg** directive appears, as indicated in the following table.

INF SECTION CONTAINING BITREG DIRECTIVE	REGISTRY KEY REFERENCED BY HKR
INF <b>DDInstall</b> section	The device's <i>software key</i>
INF <b>DDInstall.HW</b> section	The device's <i>hardware key</i>
INF <b>DDInstall.Services</b> section	The <b>Services</b> key

**Note** HKR cannot be used in a bit-registry-section referenced from an INF **DefaultInstall** section.

For more information about driver information that is stored under the **HKEY\_LOCAL\_MACHINE** root, see [Registry Trees and Keys for Devices and Drivers](#).

*subkey*

This optional value, expressed either as a %strkey% token defined in a **Strings** section of the INF or as a registry path under the given *reg-root*(key1|key2|key3...), specifies the key that contains the value entry to be modified.

*value-entry-name*

Specifies the name of an existing **REG\_BINARY**-type value entry in the (existing) subkey to be modified. It can be expressed either as "quoted string" or as a %strkey% token that is defined in the INF's **Strings** section.

*flags*

This optional hexadecimal value, expressed as an ORed bitmask of system-defined low word and high word flag values, specifies whether to clear or set the bits specified in the given byte-mask. Its default value is zero, which clears the bits in the 64-bit section of the registry.

Bitmask values for each of these flags are as follows:

**0x00000000 (FLG\_BITREG\_CLEARBITS)**

Clear the bits specified by *byte-mask*.

**0x00000001 (FLG\_BITREG\_SETBITS)**

Set the bits specified by *byte-mask*.

**0x00004000 (FLG\_BITREG\_32BITKEY)**

(Windows XP and later versions of Windows.) Make the specified change in the 32-bit registry. If not specified, the change is made to the native registry.

*byte-mask*

This byte-sized mask, expressed in hexadecimal notation, specifies which bits to clear or set in the current value of the given *value-entry-name*.

*byte-to-modify*

This byte-sized value, expressed in decimal, specifies the zero-based index of the byte within the **REG\_BINARY**-type value to be modified.

## Remarks

Each *bit-registry-section* name must be unique to the INF file, but it can be referenced by **BitReg** directives in other sections of the same INF. Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The value of an existing **REG\_BINARY**-type value entry can also be modified by overwriting its current value within an add-registry section elsewhere in the INF file. For more information about add-registry sections, see the reference for the [AddReg](#) directive.

Using a **BitReg** directive requires the definition of another INF file section. However, the value of an existing **REG\_BINARY**-type value entry can be modified bit-by-bit in such a section, thereby preserving the values of all remaining bits.

## Examples

The following example shows a bit-registry section for a fictional application.

```
[AppX_BitReg]
; set first bit of byte 0 in ProgramData value entry
HKLM,Software\AppX,ProgramData,1,0x01,0
; preceding would change value 30,00,10 to 31,00,10

; clear high bit of byte 2 in ProgramData value entry
HKLM,Software\AppX,ProgramData,,0x80,2
; preceding would change value 30,00,f0 to 30,00,70

; set second and third bits of byte 1 in ProgramData value entry
HKLM,Software\AppX,ProgramData,1,0x06,1
; preceding would change value 30,00,f0 to 30,06,f0
```

## See also

[AddInterface](#)

[AddReg](#)

[AddService](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.Coinstallers](#)

[DDInstall.HW](#)

[InterfaceInstall32](#)

# INF CopyFiles Directive

12/1/2020 • 8 minutes to read • [Edit Online](#)

A **CopyFiles** directive can do either of the following:

- Cause a single file to be copied from the source media to the default destination directory.
- Reference one or more INF-writer-defined sections in the INF that each specifies a list of files to be copied from the source media to the destination.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)
```

```
CopyFiles=@filename | file-list-section[, file-list-section]...
```

A **CopyFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive can also be specified within any of the following INF sections:

- An *add-interface-section* referenced by the INF [AddInterface](#) directive in a [DDInstall.Interfaces](#) section.
- An *install-interface-section* referenced in an INF [InterfaceInstall32](#) section

Each named section referenced by a **CopyFiles** directive has one or more entries of the following form:

```
[file-list-section]
destination-file-name[,source-file-name][,[unused][,flag]]
...
```

An INF-writer-defined *file-list-section* can have any number of entries, each on a separate line.

Each *file-list-section* can have an optional, associated *file-list-section.security* section of the following form:

```
[file-list-section.security]
"security-descriptor-string"
```

## Entries

### *destination-file-name*

Specifies the name of the destination file. If no *source-file-name* is given, this specification is also the name of the source file.

### *source-file-name*

Specifies the name of the source file. If the source and destination file names for the file copy operation are the same, *source-file-name* can be omitted.

*unused*

This entry is no longer supported in Windows 2000 and later versions of Windows.

*flag*

These optional flags, expressed in hexadecimal notation or as a decimal value in a section entry, can be used to control how (or whether) a particular source file is copied to the destination. One or more (ORed) values for the following system-defined flags can be specified. However, some of these flags are mutually exclusive:

**0x00000001 (COPYFLG\_WARN\_IF\_SKIP)**

Send a warning if the user chooses not to copy a file. This flag and the next are mutually exclusive, and both are irrelevant to INF files that are digitally signed.

**0x00000002 (COPYFLG\_NOSKIP)**

Do not allow the user to skip copying a file. This flag is implied if the [driver package](#) is signed.

**0x00000004 (COPYFLG\_NOVERSIONCHECK)**

Ignore file versions and write over existing files in the destination directory. This flag and the next two are mutually exclusive. This flag is irrelevant to digitally signed INF files.

**0x00000008 (COPYFLG\_FORCE\_FILE\_IN\_USE)**

Force file-in-use behavior: do not copy over an existing file of the same name if it is currently open. Instead, copy the given source file with a temporary name so that it can be renamed and used when the next restart occurs.

**0x00000010 (COPYFLG\_NO\_OVERWRITE)**

Do not replace an existing file in the destination directory with a source file of the same name. This flag cannot be combined with any other flags.

**0x00000020 (COPYFLG\_NO\_VERSION\_DIALOG)**

Do not write over a file in the destination directory with the source file if the existing file is newer than the source file.

The newer check is done using the file version, as extracted from the VS\_VERSIONINFO file version resource. For more info, see [Version Information](#). If the target file is not an executable or resource image, or the file does not contain file version information, then device install assumes that the target file is older.

**0x00000040 (COPYFLG\_OVERWRITE\_ONLY)**

Copy the source file to the destination directory only if the file on the destination is superseded by a newer version. This flag is irrelevant to digitally signed INF files. The version check uses the same procedure as that described above in COPYFLG\_NO\_VERSION\_DIALOG.

**0x00000400 (COPYFLG\_REPLACEONLY)**

Copy the source file to the destination directory only if the file is already present in the destination directory.

**0x00000800 (COPYFLG\_NODECOMP)**

(Windows 7 and later) Copy the source file to the destination directory without decompressing the source file if it is compressed.

**0x00001000 (COPYFLG\_REPLACE\_BOOT\_FILE)**

This file is required by the system loader. The system will prompt the user to restart the system.

**0x00002000 (COPYFLG\_NOPRUNE)**

Do not delete this operation as a result of optimization.

For example, Windows might determine that the file copy operation is not necessary because the file

already exists. However, the writer of the INF knows that the operation is required and directs Windows to override its optimization and perform the file operation.

This flag can be used to ensure that files are copied if they are also specified in an INF [DelFiles](#) directive or an INF [RenFiles](#) directive.

#### 0x00004000 (COPYFLG\_IN\_USE\_RENAME)

If the source file cannot be copied because the destination file is being used, rename the destination file, then copy the source file to the destination file, and delete the renamed destination file. If the destination file cannot be renamed, complete the copy operation during the next system restart. If the renamed destination file cannot be deleted, delete the renamed destination file during the next system restart.

#### *security-descriptor-string*

Specifies a security descriptor, to be applied to all files copied by the named *file-list-section*. The *security-descriptor-string* is a string with tokens to indicate the DACL (D:) security component.

For information about security descriptor strings, see [Security Descriptor Definition Language \(Windows\)](#). For information about the format of security descriptor strings, see [Security Descriptor Definition Language \(Windows\)](#).

If an *file-list-section*.**security** section is not specified, files inherit the security characteristics of the directory into which the files are copied.

If an *file-list-section*.**security** section is specified, the following ACE's must be included so that installations and upgrades of devices and system service packs can occur:

- (A;;GA;;;SY) – Grants all access to the local system.
- (A;;GA;;;BA) – Grants all access to built-in administrators.

Do *not* specify ACE strings that grant write access to nonprivileged users.

For more information about how to specify security descriptors, see [Creating Secure Device Installations](#).

## Remarks

Windows only copies a [driver package](#) to its destination location as part of a driver installation if the file has an INF [CopyFiles](#) directive. When it copies files, the operating system automatically generates temporary file names, when necessary, and renames the copied source files the next time that the operating system is started.

The INF file writer must also supply path specifications for files that are copied from source media by using the INF [SourceDiskNames](#) section and the INF [SourceDiskFiles](#) section to explicitly specify the path of each source file relative to the INF file in the source media.

The destination of copy operations is controlled by the [INF DestinationDirs section](#). This section controls the destination for all file-copy operations, as follows:

- If a named section referenced by a [CopyFiles](#) directive has a corresponding entry in the [DestinationDirs](#) section of the same INF, that entry explicitly specifies the target destination directory into which all files that are listed in the named section are copied. If the named section is not listed in the [DestinationDirs](#) section, Windows uses the [DefaultDestDir](#) entry in the [DestinationDirs](#) section of the INF file.
- If a [CopyFiles](#) directive uses the @*filename* syntax, Windows uses the [DefaultDestDir](#) entry in the [DestinationDirs](#) section of the INF file.

The following points apply to the INF [CopyFiles](#) directive:

- Every *file-list-section* name must be unique to the INF file, but it can be referenced by [CopyFiles](#),

[DelFiles](#), or [RenFiles](#) directives elsewhere in the same INF file. The section name must follow the general rules that are described in [General Syntax Rules for INF Files](#).

- File names that are specified in either the @*filename* or *file-list-section* entries must be the exact name of a file on the source media. You cannot use a %*strkey*% token to specify the file name. For more information about %*strkey*% tokens, see [INF Strings Section](#).
- The [CopyFiles](#) directive does not support decorating a *file-list-section* name with a system-defined platform extension (.nt, .ntx86, .ntia64, or .ntamd64).
- Do not use [CopyFiles](#) directives to copy INF files. For more information, see [Copying INF Files](#).

Starting with Windows Vista, the following points also apply to the INF [CopyFiles](#) directive:

- When the DIFx tools preinstall a driver package in the [driver store](#), they only copy a file from the driver package source to the driver store if the file has a corresponding INF [CopyFiles](#) directive.
- As part of a Windows upgrade, Windows only copies a driver package file to the [driver store](#) as part of a driver migration if the file has an INF [CopyFiles](#) directive.

## Examples

This example shows how the [SourceDiskNames](#), [SourceDiskFiles](#), and [DestinationDirs](#) sections specify the paths for copy-file (and delete-file) operations that occur in processing a simple device-driver INF. (The same INF was also used previously as examples of [Version](#), [SourceDiskNames](#), and [SourceDiskFiles](#) sections.)

```
[SourceDiskNames]
1 = %Floppy_Description%,,\WinNT

[SourceDiskFiles.x86]
aha154x.sys = 2,\x86 ; on distribution disk 2, in subdir \WinNT\x86

[DestinationDirs]
DefaultDestDir = 12 ; DIRID_DRIVERS
; == \System32\Drivers on Windows platforms
; ... Manufacturer and Models sections omitted here

DelFiles= AHA154X.DelFiles
; defines a delete-files section not shown here
; ... some other directives and sections omitted here

[AHA154X.NTx86]
CopyFiles=@AHA154x.SYS
; ... some other directives and sections omitted here
; ...
```

This example shows how a [CopyFiles](#) directive can be used in a [DDInstall.CoInstallers](#) section of an INF for a device driver that provides two device-specific co-installers to supplement the INF processing of the system device-type-specific class installer.

```
[DestinationDirs]
XxDev_Coinstallers_CopyFiles = 11 ; DIRID_SYSTEM
; ... other file-list entries and DefaultDestDirs omitted here

; ... Manufacturer, Models, and DDInstall sections omitted here

[XxDev_Install.CoInstallers]
CopyFiles=XxDev_Coinstallers_CopyFiles
; ... AddReg omitted here

[XxDev_Coinstallers_CopyFiles]
XxPreInst.dll ; dev-specific co-installer run before class installer
XxPostInst.dll ; run after class installer (post processing)
```

As the preceding example suggests, the names of new device-specific co-installers can be constructed from the name of the provider (shown here as *Xx*) and the intended use for each such co-installer DLL (shown here as *PreInst* and *PostInst*).

For additional examples of how to use the INF **CopyFiles** directive, see the INF files for the device driver samples that are included in the *src* directory of the Windows Driver Kit (WDK).

## See also

[AddInterface](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.CoInstallers](#)

[DDInstall.Interfaces](#)

[DelFiles](#)

[DestinationDirs](#)

[InterfaceInstall32](#)

[RenFiles](#)

[SourceDisksFiles](#)

[SourceDisksNames](#)

[Strings](#)

[Version](#)

# INF CopyINF Directive

11/2/2020 • 2 minutes to read • [Edit Online](#)

A **CopyINF** directive causes specified INF files to be copied to the target system. The **CopyINF** directive is supported in Windows XP and later versions of Windows.

```
[DDInstall]  
CopyINF=filename1.inf[,filename2.inf]...
```

## Remarks

System support for the **CopyINF** directive is available in Microsoft Windows XP and later versions of Windows.

This directive is typically used when installing multifunction devices. If the installation of a multifunction device requires multiple INF files (for multiple functions that belong to multiple setup classes), using this directive ensures that Windows will find the INF files when it installs the functions. Use the following rules:

- If the functions that are supplied by a multifunction device are enumerated as children of a parent device (such as an IEEE 1284.4 device), the INF file for the parent device should have a **CopyINF** directive to copy the INF files for the device's individual functions.
- If all the functions that are supplied by a multifunction device (such as a PCI card) are enumerated as peers of one another, the INF file for each function should have a **CopyINF** directive to copy the INF files for all peer functions.

If you follow these rules, Windows can install drivers for each function without prompting the user for an installation disk for each function.

The following points apply to the **CopyINF** directive:

- Before Windows Vista, Windows copies the specified INF files as part of the default processing for **DIF\_INSTALLDEVICE** (see [SetupDiInstallDevice](#)) after the device is installed successfully.

Windows copies the specified INF files into a system directory path that it will search during device installations.

- The INF files that are specified in the **CopyINF** directive must reside in the same directory as the INF file that contains the **CopyINF** directive.
- You must include all INF files on each disk of a multidisk installation.

Starting with Windows Vista, the following points also apply to the **CopyINF** directive:

- The **CopyINF** directive causes the complete [driver package](#) that is referenced by the specified INF file to be copied into the [driver store](#). This is required in order to support the deployment of multifunction driver packages, because the original source media might not be available when the device is actually installed. If the driver package that is referenced by the specified INF file already exists in the driver store, the INF file specified in the **CopyINF** directive is ignored.
- The **CopyINF** directive is processed during driver store import instead of during device installation. This means that a call to [SetupCopyOEMInf](#) on Windows Vista and later versions of Windows causes all the **CopyINF** directives in the specified INF file to be processed at that time. This occurs recursively for each

**CopyINF** directive that is contained within the specified INF file until all referenced driver packages are copied into the driver store.

Starting with Windows 10, version 1511, under certain circumstances (for example, running Windows Update or some calls to [DiInstallDevice](#)), INFs copied with **CopyINF** will also be installed on applicable devices.

For more information about how to copy INF files, see [Copying INFs](#).

## Examples

```
[MyMfDevice.NTx86]
CopyINF = Sound.INF
```

# INF DelFiles Directive

4/29/2020 • 3 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **DelFiles** directive references an INF-writer-defined section elsewhere in the INF file, and causes that list of files to be deleted in the context of operations on the section in which the referring **DelFiles** directive is specified.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)

Delfiles=file-list-section[,file-list-section]...
```

A **DelFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive can also be specified within any of the following INF-writer-defined sections:

- An add-interface-section referenced by the **AddInterface** directive in a **DDInstall.Interfaces** section.
- An install-interface-section referenced in an **InterfaceInstall32** section

Each named section referenced by a **DelFiles** directive has one or more entries of the following form:

```
[file-list-section]

destination-file-name[,,,flag]
...
```

A *file-list-section* can have any number of entries, each on a separate line.

## Entries

*destination-file-name*

Specifies the name of the file to be deleted from the destination.

Do not specify a file that is listed in a **CopyFiles** directive. If a file is listed in both a **CopyFiles**-referenced and a **DelFiles**-referenced section, and the file is currently present on the system with a valid signature, the operating system might optimize away the copy operation but perform the delete operation. This is very likely *not* what the INF writer intended.

**Note** You cannot use a %*strkey*% token to specify the destination-file-name entry. For more information about %*strkey*% tokens, see [INF Strings Section](#).

*flag*

This optional value can be one of the following, expressed in hexadecimal notation as shown here or as a decimal value:

## 0x00000001 (DEFLG\_IN\_USE)

Delete the named file, possibly after it was used during the installation process.

Setting this flag value in an INF queues the file-deletion operation until the system has restarted if the given file cannot be deleted because it is in use while this INF is being processed. Otherwise, such a file will not be deleted.

## 0x00010000 (DEFLG\_IN\_USE1)

(Windows 2000 or later versions of Windows) This flag is a high-word version of the DEFLG\_IN\_USE flag, and it has the same purpose and effect. This flag should be used in only for installations on NT-based systems.

Setting this flag value in an INF prevents conflicts with the COPYFLG\_WARN\_IF\_SKIP flag in an INF with both **DelFiles** and **CopyFiles** directives that reference the same *file-list-section*.

## Remarks

**Important** This directive must be used carefully. We highly recommend that you do not use the **DelFiles** directive in the INF file for a Plug and Play (PnP) function driver.

Any *file-list-section* name must be unique to the INF file, but it can be referenced by **CopyFiles**, **DelFiles**, or **RenFiles** directives elsewhere in the same INF. Such an INF-writer-defined section name must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The **DelFiles** directive does not support decorating a *file-list-section* name with a system-defined platform extension (.nt, .ntx86, .ntia64, .ntamd64, .ntarm, or .ntarm64).

The **DestinationDirs** section of the INF file controls the destination for all file-deletion operations, regardless of the section that contains a particular **DelFiles** directive. If a named section referenced by a **DelFiles** directive has a corresponding entry in the **DestinationDirs** section of the same INF, that entry explicitly specifies the target destination directory from which all files that are listed in the named section will be deleted. If the named section is not listed in the **DestinationDirs** section, Windows uses the **DefaultDestDir** entry in the INF.

## Examples

This example shows how the **DestinationDirs** section specifies the path for a delete-file operation that occurs in processing a simple device-driver INF.

```
[DestinationDirs]
DefaultDestDir = 12 ; DIRID_DRIVERS

; ...

[AHA154X]
CopyFiles=@AHA154x.MPD
DelFiles=ASPIDEV ; defines delete-files section name
; ... some other directives and sections omitted here

[ASPIDEV]
VASPID.SYS ; name of file to be deleted, if it exists on target
; ...
```

## See also

[AddInterface](#)

[ClassInstall32](#)

[CopyFiles](#)

*DDInstall*

*DDInstall.Coinstallers*

[DestinationDirs](#)

[InterfaceInstall32](#)

[RenFiles](#) [Strings](#)

# INF DelProperty Directive

11/2/2020 • 3 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **DelProperty** directive references one or more INF file sections that delete [device properties](#) for a device instance, a [device setup class](#), a [device interface class](#), or a device interface.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] | (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] | (Windows 10 and later versions of Windows)

[interface-install-section] |
[interface-install-section.nt] |
[interface-install-section.ntx86] |
[interface-install-section.ntia64] | (Windows XP and later versions of Windows)
[interface-install-section.ntamd64] | (Windows XP and later versions of Windows)
[interface-install-section.ntarm] | (Windows 8 and later versions of Windows)
[interface-install-section.ntarm64] | (Windows 10 and later versions of Windows)

[add-interface-section]

DelProperty=del-property-section[,del-property-section]... (Windows Vista and later versions of Windows)
```

A **DelProperty** directive can be specified under any of the sections shown in the formal syntax statement above.

A *del-property-section* that is referenced by a **DelProperty** directive has the following format:

```
[del-property-section]
(property-name [ , flags [, value]]) | ({property-category-guid}, property-pid [ , flags [, value]])
(property-name [ , flags [, value]]) | ({property-category-guid}, property-pid [ , flags [, value]])
...
...
```

A *del-property-section* can have any number of *property-name* entries or *property-guid* entries, each on a separate line.

## Entries

### *property-name*

One of the property names that represent the device instance [driver package](#) properties. The property names that are supported are the same as those that are described for the *property-name* entry of the [INF AddProperty directive](#).

### *property-category-guid*

A GUID value that identifies the property category. The GUID value can be a system-defined GUID that identifies a system-defined property category or a custom GUID that identifies a custom property category. The GUID values that are supported are the same as those that are described for the *property-category-guid* entry of the

INF [AddProperty](#) directive.

*property-pid*

A property identifier that indicates the specific property within the property category that is indicated by the *property-category-guid* value. For internal system reasons, a property identifier must be greater than or equal to two.

*flags*

An optional hexadecimal flag value that controls the delete operation. The only flag value supported is as follows:

**0x00000001** (FLG\_DELPROPERTY\_MULTI\_SZ\_DELSTRING)

If the property data type is [DEVPROP\\_TYPE\\_STRING\\_LIST](#), the operation deletes all the strings with the existing string list that match the string that is supplied by the value entry value. The case of a character is not considered in the comparison between the supplied string and an existing string in the string list.

*value*

If the property data type is [DEVPROP\\_TYPE\\_STRING\\_LIST](#) and the flags entry is **0x00000001**, the *value* entry value supplies the string that the delete operation uses to search for matching strings in the existing string list and, if a matching string is found, the delete operation removes the matching string from the existing string list.

## Remarks

In general, an INF file should not be used to delete device properties that might be set by a system component or by another INF file. The primary purpose of the [DelProperty](#) directive is for use in an INF file that updates a previous device installation and a property that was set for a previous device installation is no longer required.

A *del-property-section* name must be unique within an INF file, but the section name can be referenced by more than one [DelProperty](#) directive in the same INF file. A section name must follow the general rules for defining section names that are described in [General Syntax Rules for INF Files](#).

For more information about how to use the [DelProperty](#) directive, see [Using the INF AddProperty Directive and the INF DelProperty Directive](#).

## Examples

The following example of a delete property section includes two line entries: the first line entry includes a *property-name* entry value that deletes the [DeviceModel](#) property, and the second line entry deletes the string "DeleteThisString" from a custom device property value whose data type is [DEVPROP\\_TYPE\\_STRING\\_LIST](#). In the second line, the *property-category-guid* entry value is "c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e," the *property-identifier* entry value is "2," and the *flags* entry value is "0x00000001,"

```
[SampleDelPropertySection]
DeviceModel
{c22189e4-8bf3-4e6d-8467-8dc6d95e2a7e}, 2, 0x00000001, "DeleteThisString"
```

## See also

[AddProperty](#)

# INF DelReg Directive

12/1/2020 • 3 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **DelReg** directive references one or more INF-writer-defined sections describing keys and/or value entries to be removed from the registry.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)
```

```
DelReg=del-registry-section[,del-registry-section]...
```

Each *del-registry-section* referenced by a **DelReg** directive has the following form:

```
[del-registry-section]
reg-root-string,subkey[,value-entry-name][,flags][,value]
reg-root-string,subkey[,value-entry-name][,flags][,value]
...

```

A *del-registry-section* can have any number of entries, each on a separate line.

## Entries

*reg-root-string*

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

**HKCR**

Abbreviation for **HKEY\_CLASSES\_ROOT**.

**HKCU**

Abbreviation for **HKEY\_CURRENT\_USER**.

**HKLM**

Abbreviation for **HKEY\_LOCAL\_MACHINE**.

**HKU**

Abbreviation for **HKEY\_USERS**.

**HKR**

Relative root, in which keys that are specified by using this abbreviation are relative to the registry key associated with the INF section in which this **DelReg** directive appears, as indicated in the following table.

NF SECTION CONTAINING ADDREG DIRECTIVE	REGISTRY KEY REFERENCED BY HKR
INF <a href="#">DDInstall</a> section	The device's <i>software key</i>
INF <a href="#">DDInstall.HW</a> section	The device's <i>hardware key</i>
INF <a href="#">DDInstall.Services</a> section	The <b>Services</b> key

**Note** HKR cannot be used in an *del-registry-section* referenced from an [INF DefaultInstall section](#).

For more information about driver information that is stored under the **HKEY\_LOCAL\_MACHINE** root, see [Registry Trees and Keys for Devices and Drivers](#).

#### *subkey*

This optional value, formed either as a %strkey% token defined in a [Strings](#) section of the INF or as a registry path under the given *reg-root* (key\key2\key3...), specifies one of the following:

- A subkey to be deleted from the registry at the end of the given registry path
- An existing subkey from which the given value-entry-name is to be deleted

#### *value-entry-name*

This value identifies a named value entry to be removed from the given subkey. This value and its preceding comma should be omitted if the subkey itself is being removed from the registry.

#### *flags*

(Windows XP and later versions of Windows.) This optional hexadecimal value, expressed as an ORed bitmask of system-defined low word and high word flag values, defines the data type for a value entry, or controls the delete-registry operation. If *flags* is not specified, the *value-entry-name* (if specified) or *subkey* will be deleted.

Bitmask values for each of these flags are as follows:

**0x00002000** (FLG\_DELREG\_KEYONLY\_COMMON)

Delete the entire subkey.

**0x00004000** (FLG\_DELREG\_32BITKEY)

Make the specified change in the 32-bit registry. If not specified, the change is made to the native registry.

**0x00018002** (FLG\_DELREG\_MULTI\_SZ\_DELSTRING)

Within a multistring registry entry, delete all strings matching a string value specified by *value*. Case is ignored.

#### *value*

(Windows XP and later versions of Windows.) Specifies a registry value, if *flags* indicates that a registry value is required.

## Remarks

A **DelReg** directive can be specified under any of the sections shown in the formal syntax statement above. This directive can also be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the [AddService](#) directive in an [INF DDInstall.Services section](#).
- An *add-interface-section* referenced by the [AddInterface](#) directive in an [INF DDInstall.Interfaces section](#).
- An *install-interface-section* referenced in an [INF InterfaceInstall32 section](#).

In general, an INF should never attempt to delete subkeys or value entries within existing subkeys that were set up by system components or by the INF files for other devices. The purpose of a *delete-registry section* is to clean stale registry information from a previous installation by using a new INF file supplied by the same provider.

Each *del-registry-section* name must be unique to the INF file, but it can be referenced by **DelReg** directives in other sections of the same INF. Each section name must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

With operating system versions prior to Windows XP, the only way to delete a key is by specifying the following:

```
reg-root-string, subkey
```

For Windows XP and later versions of Windows, the following is also permitted (to specify the 32-bit registry):

```
reg-root-string, subkey,,0x4000
```

## Examples

This example shows how the system-supplied COM/LPT ports class installer's INF removes stale NT-specific registry information about COM ports from the registry.

```
[ComPort.NT]
CopyFiles=ComPort.NT.Copy
AddReg=ComPort.AddReg, ComPort.NT.AddReg
... ; more directives omitted here

[ComPort.NT.HW]
DelReg=ComPort.NT.HW.DelReg

[ComPort.NT.Copy]
serial.sys
serenum.sys

[Comport.NT.AddReg]
HKR,,EnumPropPages32,,,"MSPorts.dll,SerialPortPropPageProvider"

[ComPort.NT.HW.DelReg]
HKR,,UpperFilters
```

## See also

[AddReg](#)

[AddInterface](#)

[AddService](#)

[BitReg](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.Coinstallers](#)

*DDInstall.HW*

*DDInstall.Services*

*InterfaceInstall32*

*Strings*

# INF DelService Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **DelService** directive is used in a **DDInstall.Services** section to remove one or more previously installed device/driver services from the target computer.

```
[DDInstall.Services]

DelService=ServiceName[, [flags][,[EventLogType][,EventName]]]
...
...
```

## Entries

### *ServiceName*

Specifies the name of the service to be removed.

For a device, this value is usually a generic name for its driver, such as "sermouse," or some such name.

### *flags*

This optional value specifies one or more of the following flags, defined in *Setupapi.h*, that are specified as a hexadecimal value:

**0x00000004 (SPSVCINST\_DELETEEVENTLOGENTRY)**

An event-log entry (or entries) associated with the given ServiceName should also be removed from the system.

**0x00000200 (SPSVCINST\_STOPSERVICE)**

Stop the service before deleting it.

### *EventLogType*

Optionally specifies one of **System**, **Security**, or **Application**. This can be omitted if the event log to be removed is of type **System**.

### *EventName*

Optionally specifies the name for the event log. This can be omitted if it is identical to the specified *ServiceName* entry.

## Remarks

This directive is rarely used. The only services that can be safely deleted are those that were used only in earlier versions of the operating system, and are therefore never used for the currently installed version.

Starting with Windows XP, you can use the *TargetOSVersion* decoration to control version-specific installation behavior. For more information about this decoration, see [INF Manufacturer Section](#).

However, by default, event-log information supplied by a particular device driver is not removed from the system on deinstallation, unless the INF for the device/driver explicitly requests the removal (*flags* or *EventName*) of the event log along with the removal of the driver services.

## See also

**AddService**

*DDInstall.Services*

**DelReg**

# INF DriverVer Directive

11/2/2020 • 2 minutes to read • [Edit Online](#)

A **DriverVer** directive specifies version information for drivers installed by this INF.

```
[Version] |
[DDInstall]

DriverVer=mm/dd/yyyy,w.x.y.z
```

## Entries

*mm/dd/yyyy*

This value specifies the date of the "[driver package](#)", which includes the driver files and the INF. This date must be the most recent date of any file in the driver package.

The date must be specified in month/day/year order. The month and day must contain two digits, and the year must contain four digits. A hyphen (-) can be used as the date field separator instead of the slash (/).

*w.x.y.z*

This value specifies a version number.

Each of *w*, *x*, *y*, and *z* must be an integer that is greater than or equal to zero and less than 65535.

For Windows XP SP1, Windows Server 2003 and later versions of Windows, this value is also *used* by Setup, in combination with the driver rank and date, to select a driver for a device. For more information, see [How Windows Selects Drivers](#).

The following point applies to this value for Windows 2000, and Windows XP:

- You should consider this value to be required for input drivers (such as mouse or keyboard drivers). If you do not include the version value, input drivers might not update programmatically. Typically, you should specify version information in all [driver packages](#) because the operating system uses version information as a criteria to determine the newest driver.

## Remarks

Starting with Windows 2000, INF files must have a **DriverVer** directive in their [INF Version sections](#) to provide version information for the whole INF. Individual [INF DDInstall sections](#) can also contain **DriverVer** directives to provide version information for individual drivers. **DriverVer** directives in the *DDInstall* sections are more specific and take precedence over the global **DriverVer** directive in the **Version** section.

When the operating system searches for drivers, it selects a driver that has a more recent **DriverVer** date over a driver that has an earlier date. If an INF has no **DriverVer** directive or contains an invalid date specification, the operating system applies the default date of 00/00/0000. For Windows 2000 only, unsigned drivers are also assigned a date of 00/00/0000.

## Examples

```
[Version]
...
DriverVer=09/28/1999,5.00.2136.1
```

## See also

[\*DDInstall\*](#)

[\*\*Version\*\*](#)

# INF FeatureScore Directive

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **FeatureScore** directive provides an additional ranking criterion for drivers based on the features that a driver supports. For example, feature scores might be defined for a [device setup class](#) that distinguishes between drivers that are based on class-specific criteria.

```
[DDInstall]  
FeatureScore=featurescore
```

The **FeatureScore** directive is supported in Windows Vista and later versions of Windows.

**Warning** The **FeatureScore** directive is only processed when specified directly in the **[DDInstall]** section.

## Entries

### *featurescore*

This value specifies the ranking score for the driver based on its feature contents. This entry is a single-byte hexadecimal number between 0x00 and 0xFF.

A lower *featurescore* value specifies a better feature score rank, where 0x00 is the best feature score rank. If the **FeatureScore** directive is not specified, Windows uses a default feature score rank of 0xFF for the driver.

## Remarks

If Windows detects multiple drivers for the same device, it must first determine which driver is the best driver to install. To accomplish this, Windows assigns each driver an overall rank based on several factors, or scores, such as the following:

- A driver-signing score ([signature score](#)), based on whether the driver is signed or not.
- A driver feature score ([feature score](#)), based on how the driver's features rank compared to another driver for the device.
- A hardware identifier score ([identifier score](#)), based on how closely the Plug and Play (PnP) device identification strings that is reported by the bus driver for the device matches a [device identification string](#) in the INF [\*\*Models\*\*](#) section of the INF file.

The feature score provides a way to rank drivers based on the features that a driver supports. For example, feature scores might be defined for a [device setup class](#) that distinguishes between drivers based on class-specific criteria.

The feature score supplements the identifier score, which makes it possible for driver writers to more easily and precisely distinguish between different drivers for a device that is based on well-defined criteria.

For more information about how drivers are ranked, see [How Windows Ranks Drivers \(Windows Vista and Later\)](#).

## See also

[feature score](#)

[identifier score](#)

[\*\*Models\*\*](#)

signature score

# INF HardwareId Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** The **HardwareId** directive is only supported within an *Autorun.inf* file. This directive must not be used within the INF files that are used for PnP device installations.

Starting with Windows Vista, the Found New Hardware Wizard and Hardware Update Wizard support INF **HardwareId** directives in the [DeviceInstall] section of an *Autorun.inf* file. The author of *Autorun.inf* can use these **HardwareId** directives to specify Plug and Play (PnP) hardware identifiers (IDs) of the devices for which the AutoRun-enabled application provides and installs drivers.

```
[DeviceInstall]  
HardwareId="pnp-hardware-id"  
...
```

## Entries

"*pnp-hardware-id*"

This value specifies a PnP device hardware ID. The hardware ID must be enclosed in double quotation marks ("").

The hardware ID can be fairly generic, such as PCI\VEN\_1234&DEV\_1234, or very specific, such as PCI\VEN\_1234&DEV\_1234&SUBSYS\_12345678&REV\_01.

Only one PnP hardware ID can be specified per **HardwareId** directive. To specify multiple hardware IDs, use multiple **HardwareId** directives, one per line.

## Remarks

During a **hardware-first installation**, the user installs a hardware device before installing the drivers for that device. In this case, the Found New Hardware Wizard prompts the user for the distribution medium.

If the distribution medium has an AutoRun-enabled *device installation application*, the wizard parses the *Autorun.inf* file to look for a **HardwareId** directive entry that matches the device that is being installed. If the wizard finds a **HardwareId** directive that matches the device, the wizard invokes the AutoRun-enabled application, which installs the driver and device-specific applications instead of the wizard.

The Found New Hardware Wizard does not determine whether the application installed a driver for the device. In this case, the application must install a driver for the device. If the *Autorun.inf* file does not include a **HardwareId** directive that identifies the device that is being installed, the wizard does not start the application and continues with device installation.

Although there may be multiple **HardwareId** directives within the [DeviceInstall] section of an *Autorun.inf* file, each directive should specify a unique PnP hardware ID.

# INF Ini2Reg Directive

11/2/2020 • 3 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

An **Ini2Reg** directive references one or more named sections in which lines or sections from a supplied INI file are moved into the registry. This creates or replaces one or more value entries under a specified key.

```
[  
DDInstall  
] |  
[DDInstall.CoInstallers] |  
[ClassInstall32] |  
[ClassInstall32.ntx86] |  
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)  
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)  
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)  
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)  
  
Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...
```

Each named section referenced by an **Ini2Reg** directive has the following form:

```
[ini-to-registry-section]  
  
ini-file,ini-section,[ini-key],reg-root,subkey[,flags]  
...
```

An *ini-to-registry-section* can have any INF-writer-determined number of entries, each on a separate line.

## Entries

### *ini-file*

Specifies the name of an INI file supplied on the source media. This value can be expressed as a *filename* or as a %*strkey*% token that is defined in a **Strings** section of the INF file.

### *ini-section*

Specifies the name of the section within the given INI file that contains the registry information to be copied.

### *ini-key*

Specifies the name of the key in the INI file to copy to the registry. If this value is omitted, the whole *ini-section* is to be transferred to the specified registry *subkey*.

### *reg-root*

Identifies the root of the registry tree for other values supplied in this entry. For specifics, see the reference for the [AddReg directive](#).

### *subkey*

Identifies the subkey to receive the value, expressed either as a %*strkey*% token defined in a **Strings** section of the INF or as an explicit registry path (*key1\key2\key3...*) from the given *reg-root*.

### *flags*

Specifies (in bit 0) how to handle the INI file after transferring the given information to the registry and/or (in bit 1) whether to overwrite existing registry information, as follows:

Bit zero = 0

Do not remove the given information from the INI file after copying it into the registry. This is the default.

Bit zero = 1

Delete the given information from the INI file after moving it into the registry.

Bit one = 0

If the specified subkey already exists in the registry, do not transfer the INI-supplied information into this *subkey*. Otherwise, create the specified *subkey* in the registry with this INI-supplied information as its value entry. This is the default.

Bit one = 1

If the specified subkey already exists in the registry, replace its value entry with the INI-supplied information.

## Remarks

The **Ini2Reg** directive is valid in any of the sections shown in the formal syntax statement. This directive is also valid in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

If an INF file is used to install devices on Windows XP and later versions of Windows, the INF file should not contain **Ini2Reg** directives. INF files that contain **Ini2Reg** directives will not pass "Designed For Windows" logo testing, will not receive a digital signature, and therefore will be untrusted by Windows (see [How Windows Selects Drivers](#)).

Each *ini-to-registry-section* name must be unique to the INF file. Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The INF provides the full path of the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INF files, by using the **SourceDiskNames** and, possibly, **SourceDiskFiles** sections of this INF to explicitly specify the full path of each named source file that is not in the root directory (or directories) on the distribution media.
- In system-supplied INF files, by supplying one or more additional INF files, identified in the **LayoutFile** entry in the **Version** section of the INF file.

## See also

[AddInterface](#)

[AddReg](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall/Colnstallers](#)

[InterfaceInstall32](#)

[SourceDiskFiles](#)

[SourceDiskNames](#)

[Strings](#)

[UpdateIniFields](#)

[UpdateInis](#)

[Version](#)

# INF LogConfig Directive

11/2/2020 • 11 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **LogConfig** directive references one or more INF-writer-defined sections, each of which specifies a logical configuration of hardware resources – the interrupt request lines, memory ranges, I/O ports, and DMA channels that can be used by the device. Each *log-config-section* specifies an alternative set of bus-relative hardware resources that can be used by the device.

```
[DDInstall] |
[DDInstall.LogConfigOverride]

LogConfig=log-config-section[,log-config-section]...
```

INF files for non-PnP devices use this directive to create basic configurations.

INF files for PnP devices use this directive only to create override configurations.

Each named section referenced by a **LogConfig** directive has the following form:

```
[log-config-section]

ConfigPriority=priority-value[,config-type]
[DMAConfig=[DMAAttrs:]DMANum[,DMANum]...)
[IOConfig=io-range[,io-range]...)
[MemConfig=mem-range[,mem-range]...)
[IRQConfig=[IRQAttrs:]IRQNum[,IRQNum]...)
[PcCardConfig=ConfigIndex[:MemoryCardBase1][:MemoryCardBase2][(attrs)]]
[MfCardConfig=ConfigRegBase:ConfigOptions[:IoResourceIndex][(attrs)]...
...
```

## Entries

### ConfigPriority = *priority-value*

Specifies the priority value for this logical configuration, as one of the following:

DESIRED

Soft configured, most optimal.

NORMAL

Soft configured, less optimal than DESIRED. This is the typical setting.

NORMAL should be specified if the *log-config-section* was defined in a [DDInstall.LogConfigOverride](#) section, and no *config-type* value can be specified.

SUBOPTIMAL

Soft configured, less optimal than NORMAL.

HARDRECONFIG

Requires a jumper change to reconfigure.

HARDWIRED

Cannot be changed.

#### RESTART

Requires restart to take effect.

#### REBOOT

This is the same as RESTART.

#### POWEROFF

Requires power cycle to take effect.

#### DISABLED

Hardware/device is disabled.

### DMAConfig=[DMAAttrs]DMANum[,DMANum]...]

*DMAAttrs* is optional if the device is connected on a bus that has only 8-bit DMA channels and the device uses standard system DMA. Otherwise, it can be one of the following letters:

LETTER	MEANING
D	32-bit DMA
W	16-bit DMA
N	8-bit DMA

If the device uses bus-master DMA, you must use **M** with one of the following (mutually exclusive) letters that indicates the type of DMA channel used: **A**, **B**, or **F**. If neither **A**, **B**, or **F** are specified, a standard DMA channel is assumed.

*DMANum* specifies one or more bus-relative DMA channels as decimal numbers, each separated from the next by a comma (,).

### IOConfig=io-range[,...]

Specifies one or more I/O port ranges for the device, in either of the following forms:

*start-end*[([*decode-mask*][:*alias-offset*][:*attr*])] (Type 1 I/O range)

*start*

Specifies the starting address of the I/O port range as a 64-bit hexadecimal address.

*end*

Specifies the ending address of the I/O port range, also as a 64-bit hexadecimal address.

*decode-mask*

Defines the alias type and can be any of the following:

MASK VALUE	MEANING	IOR_ALIAS VALUE
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	Positive decode	0xFF

*alias-offset*

Not used.

*attr*

Specifies the letter **M** if the given range is in system memory. If omitted, the given range is in I/O port space.

*size@min-max[%align-mask][(decode-mask)[:alias-offset][:attr]]* (Type 2 I/O range)

*size*

Specifies the number of bytes required for the I/O port range as a 32-bit hexadecimal value.

*min*

Specifies the lowest possible starting address of the I/O port range as a 64-bit hexadecimal address.

*max*

Specifies the highest possible ending address of the I/O port range as a 64-bit hexadecimal address.

*align-mask*

Optionally specifies a 64-bitmask that is used in a bitwise AND operation to align the start of the I/O port range on an integral (usually 32-bit or 64-bit) address boundary.

*decode-mask*

Defines the alias type and can be any of the following:

MASK VALUE	MEANING	IOR_ALIAS VALUE
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	Positive decode	0xFF

*alias-offset*

Not used.

*attr*

Specifies the letter **M** if the given range is in system memory. If omitted, the given range is in I/O port space.

**MemConfig=mem-range[,mem-range]...**

Specifies one or more memory ranges for the device in one of the following forms:

```
start-end[(attr)] | size@min-max[%align-mask][(attr)]
```

*start*

Specifies the starting (bus-relative) physical address of the device memory range as a 64-bit hexadecimal value.

*end*

Specifies the ending physical address of the memory range, also as a 64-bit hexadecimal value.

*attr*

Specifies the attributes of the memory range as one or more of the following letters:

LETTER	MEANING
R	Read-only

LETTER	MEANING
W	Write-only
RW	Read/write
C	Combined write allowed
H	Cacheable
F	Prefetchable
D	Card decode addressing is 32-bit, instead of 24-bit

If both R and W are specified or if neither is specified, read/write is assumed.

#### *size*

Specifies the number of bytes required in the memory range as a 32-bit hexadecimal value.

#### *min*

Specifies the lowest possible starting address of the device memory range as a 64-bit hexadecimal value.

#### *max*

Specifies the highest possible ending address of the memory range as a 64-bit hexadecimal value.

#### *align-mask*

Optionally specifies a 64-bitmask that is used in a bitwise AND operation to align the start of the device memory range on an integral (usually 64-bit) address boundary.

If align-mask is omitted, the default memory alignment is on a 4K boundary (FFFFF000).

#### **IRQConfig=[/RQattr]**[/RQNum[,/RQNum]...]

/RQattr is omitted if the device uses a bus-relative, edge-triggered IRQ. Otherwise, specify L to indicate a level-triggered IRQ and LS if the device can share the IRQ lines listed in this entry.

/RQNum specifies one or more bus-relative IRQs the device can use as decimal numbers, each separated from the next by a comma (,).

#### **PcCardConfig=ConfigIndex[:MemoryCardBase1][:MemoryCardBase2][( attrs)]**

Configures CardBus registers and/or creates up to two permanent memory windows that map to the attribute space of the device. A driver can use the memory windows to access the attribute space from an ISR. Specify all numeric values in hexadecimal format.

The elements of a **PcCardConfig** entry are as follows:

#### *ConfigIndex*

Specifies the 8-bit PCMCIA configuration index for a device on a PCMCIA bus.

#### *MemoryCardBase1*

Optionally specifies a 32-bit base address for a first memory window.

#### *MemoryCardBase2*

Optionally specifies a 32-bit base address for a second memory window.

#### *attrs*

Optionally specifies one or more attributes for the device, separated by spaces. An invalid attribute specifier invalidates the whole **PcCardConfig** entry. If more than one specifier for a particular attribute is provided, the attributes are applied to individual I/O or memory windows for the device. If only one specifier is provided, then

that attribute is applied to all windows (see the following example).

Specifically, if multiple specifiers are provided, the first specifier found reading from left to right will be applied to the first window, and the next specifier applied to the second window. A maximum of two I/O windows and two memory windows may be controlled by a single **PcCardConfig** entry. If the device has more than two memory windows, then a second **PcCardConfig** entry must be included.

The attributes include:

ATTRIBUTE	DESCRIPTION
<b>W</b>	16-bit I/O data path. The default is 8-bit if the INF specifies a <b>LogConfig</b> directive. If no <b>LogConfig</b> directive is specified, the driver uses 16-bit I/O.
<b>S<math>n</math></b>	$\sim$ IOCS16 source. If $n$ is 0, $\sim$ IOCS16 is based on the value of the datasize bit. If $n$ is 1, $\sim$ IOCS16 is based on the $\sim$ IOIS16 signal from the device. The default is S1.
<b>Z<math>n</math></b>	I/O 8-bit, zero wait state. If $n$ is 1, 8-bit I/O accesses occur with zero additional wait states. If $n$ is 0, access occurs with additional wait states. This flag has no meaning for 16-bit I/O. The default is Z0.
<b>XI<math>n</math></b>	I/O wait states. If $n$ is 1, 16-bit system accesses occur with one additional wait state. The default is XI1.
<b>M</b>	16-bit memory data path. The default is 8-bit.
<b>M8</b>	8-bit memory data path.
<b>XM<math>n</math></b>	Memory wait states, where $n$ can be 0, 1, 2 or 3. This value determines the number of additional wait states for 16-bit accesses to a memory window. The default is XM3.
<b>A</b>	Memory range to be mapped as Attribute memory.
<b>C</b>	Memory range to be mapped as Common Memory (default).

For example, an attrs value of (WB CA M XM1 XI0) translates to the following:

1st I/O window is 16-bit

2nd I/O window 8-bit

1st memory window is common

2nd memory window is attribute

Memory is 16-bit

One wait state on memory windows

Zero wait states on I/O windows

**MfCardConfig** = *ConfigRegBase*.*ConfigOptions*[:*IoResourceIndex*][(*attrs*)...] Specifies the attribute-memory location of the set of configuration registers for one function of a multifunction device, as follows:

*ConfigRegBase*

Specifies the attribute offset of the configuration registers for this function of the multifunction device.

*ConfigOptions*

Specifies the 8-bit PCMCIA configuration option register.

*IoResourceIndex*

Specifies the index to the **IOConfig** entry for the bus driver to use in programming the configuration I/O base and limit registers. This index is zero-based, that is, zero designates the initial **IOConfig** entry in this *log-config-section*.

*attrs*

If set to the letter A, directs the PCMCIA bus driver to turn on audio enable in the configuration and status registers.

Each **MfCardConfig** entry supplies information about a single function of the multifunction device. When a set of **LogConfig** directives each reference a discrete *log-config-section* in the INF's

**DDInstall.LogConfigOverride** section, each *log-config-section* must have its entries, including **MfCardConfig** entries, listed in the same order.

## Remarks

A **LogConfig** directive can be specified under any per-manufacturer, per-models [INF DDInstall section](#) or [INF DDInstall.LogConfigOverride section](#).

An INF for a non-PnP device that supports several alternative logical configurations typically defines a set of *log-config-sections* under a **DDInstall** section. Each *log-config-section* specifies a discrete set of logical configuration resources. It also includes a **ConfigPriority** entry, which ranks each logical configuration according to its effects on device and driver performance, ease of initialization, and so forth.

For PnP devices, the PnP manager assigns a set of logical hardware resources to each PnP device. That is, the PnP manager queries the system bus drivers, receives their reports of per-device I/O bus configuration resources in use, and assigns per-device sets of logical hardware resources to achieve the best system-wide balance in the usage of all such resources.

As a result, the **LogConfig** directive under a **DDInstall** section is ignored for PnP devices,. To override the resources reported by the bus for a PnP device, include the **LogConfig** directive under a **DDInstall.LogConfigOverride** section. In this case, the resources specified in the *log-config-section* is used instead of those reported by the bus.

Platform extensions can be added to a **DDInstall** section that contains a **LogConfig** directive, or to a **DDInstall.LogConfigOverride** section, to specify platform-specific or OS-specific logical configurations. For more information, see [Creating an INF File](#).

A given *log-config-section* name must be unique to the INF file, but it can be referenced by **LogConfig** directives in other INF **DDInstall** sections for the same devices. Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

Only one **ConfigPriority** entry can be used in each *log-config-section*. There can be more than one of each of the other entries, depending on the hardware resource requirements of the device.

One or more **MfCardConfig=** entries can appear only in a *log-config-section* that is referenced by a **LogConfig** directive in the *DD\Install\LogConfigOverride* section of an INF for a multifunction device. For more information about INF files for multifunction devices, see [Supporting Multifunction Devices](#).

### **LogConfig-Referenced Section Entries and Values**

From a *log-config-section*, the system installer builds binary logical configuration records and stores them in the registry.

An INF file can contain any number of per-device *log-config-sections*. However, each such section must contain complete information for installing one device. In general, the INF should specify the entries in each of its *log-config-sections* in the same order. The INF should specify each set of entries in the order best suited to how the driver initializes its device.

If more than one *log-config-section* is present for a given device, only one of these INF sections is used during installation. Such an INF file partially controls which such section is used with the **ConfigPriority** value that it supplies in each such *log-config-section*. That is, the system installers attempt to honor any configuration priorities in an INF file, but might select a lower ranked logical configuration if a conflict with an already installed device is found.

During installation, one and only one resource from each entry in a given *log-config-section* is selected and assigned to a particular device. If a particular device needs more than one resource of the same type, a set of entries of that type must be used in its *log-config-sections*.

For example, to ensure two I/O port ranges for a particular device, two **IOConfig=** entries must be specified in the *log-config-section* for that device. On the other hand, if a device requires no IRQ, its INF can omit the **IRQConfig** entry from the *log-config-sections*.

## Examples

This example shows some valid **PcCardConfig** entries for a PCMCIA device.

```
PcCardConfig=0:E0000:F0000(W)
PcCardConfig=0:E0000(M)
PcCardConfig=0::(W)
PcCardConfig=0(W)
```

This example shows a Type 1 I/O range specification in an **IOConfig** entry. It specifies an I/O port region, eight bytes in size, which can start at 1F8, 2F8, or 3F8.

```
IOConfig=1F8-1FF, 2F8-2FF, 3F8-3FF
```

By contrast, this example shows a Type 2 I/O range specification in an **IOConfig** entry. It specifies an I/O port region, eight bytes in size, which can start at 300, 308, 310, 318, 320, or 328.

```
IOConfig=8@300-32F%FF8
```

This example shows a set of **IOConfig** entries for a four-port device, each specifying an I/O port range that is offset by 0x400 bytes from the next.

```
IoConfig=0x200-0x21f  
IoConfig=0x600-0x61f  
IoConfig=0xA00-0xA1f  
IoConfig=0xE00-0xE1f
```

The next two examples show typical **MemConfig** entries.

This example specifies a memory region of 32K bytes that can start at either C0000 or D0000.

```
MemConfig=C0000-C7FFF, D0000-D7FFF
```

This example specifies a memory region of 32k bytes starting on 64K boundaries.

```
MemConfig=8000@C0000-D7FFF%F0000
```

This example shows how the system HDC class INF file sets up several *log-config-sections* for generic ESDI hard disk controllers and uses a [\*\*DD\Install\LogConfigOverride\*\*](#) section for a particular IDE controller.

```
[MS_HDC] ; per-manufacturer Models section  
%FujitsuIdePccard.DeviceDesc% =  
    atapi_fujitsu_Inst, PCMCIA\FUJITSU-IDE-PC_CARD-DDF2  
%*PNP0600.DeviceDesc% = atapi_Inst, *PNP0600 ; generic ESDI HDCs  
%PCI\CC_0101.DeviceDesc% = pciide_Inst,,PCI\CC_0101  
  
; ... other manufacturers' Models sections omitted  
  
[atapi_Inst]  
CopyFiles = @atapi.sys  
LogConfig = esdilc1, esdilc2, esdilc3, esdilc4  
  
; ... [atapi_Inst.Services] + service/EventLog-install omitted here  
  
[esdilc1]  
ConfigPriority=HARDWIRED  
IOConfig=1f0-1f7(3ff::)  
IoConfig=3f6-3f6(3ff::)  
IRQConfig=14  
  
[esdilc2]  
ConfigPriority=HARDWIRED  
IOConfig=170-177(3ff::)  
IoConfig=376-376(3ff::)  
IRQConfig=15  
  
[esdilc3]  
ConfigPriority=HARDWIRED  
IOConfig=1e8-1ef(3ff::)  
IoConfig=3ee-3ee(3ff::)  
IRQConfig=11  
  
[esdilc4]  
;  
  
[atapi_fujitsu_Inst.LogConfigOverride]  
LogConfig = fujitsu.LogConfig0  
  
[fujitsu.LogConfig0]  
ConfigPriority=NORMAL  
IOConfig=10@100-400%ffff  
IRQConfig=14,15,5,7,9,11,12,3  
PcCardConfig=1:0:0(W)
```

For some examples of how **MfCardConfig** entries are used, see [Supporting PC Cards That Have Incomplete Configuration Register Addresses](#).

## See also

[\*DDInstall\*](#)

[\*DDInstall.FactDef\*](#)

# INF ProfileItems Directive

4/29/2020 • 5 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **ProfileItems** directive is used in an [INF DDInstall section](#) to list one or more *profile-items-sections* that contain items or groups to be added to, or removed from, the Start menu.

```
[DDInstall]  
  
ProfileItems=profile-items-section[,profile-items-section]...  
...
```

Each named section referenced by a **ProfileItems** directive has the following form:

```
[profile-items-section]  
  
Name=link-name[,name-attributes]  
CmdLine=dirid,[subdir],filename  
[SubDir=path]  
[WorkingDir=wd-dirid,wd-subdir]  
[IconPath=icon-dirid,[icon-subdir],icon-filename]  
[IconIndex=index-value]  
[HotKey=hotkey-value]  
[Infotip=info-tip]  
[DisplayResource="ResDllPath\ResDll",ResID]
```

This directive is supported in Windows XP and later versions of Windows.

## Entries

**Name**=*link-name*[,*name-attributes*]

The *link-name* specifies the name of the link for the menu item or group, without the *.lnk* extension. This value can be a string or a %strkey% token that is defined in a [Strings](#) section of the INF file. If a **DisplayResource** entry is not specified, *link-name* is also the display string.

The optional *name-attributes* value specifies one or more flags that affect the operation on the menu item. This value is expressed as an ORed bitmask of system-defined flag values. Possible flags include the following:

**0x00000001** (FLG\_PROFITEM\_CURRENTUSER)

Directs Windows to create or delete a Start menu item in the current user's profile. If this flag is not specified, Windows processes the item for all users.

**0x00000002** (FLG\_PROFITEM\_DELETE)

Directs Windows to delete the menu item. If this flag is not specified, the item is created.

**0x00000004** (FLG\_PROFITEM\_GROUP)

Directs Windows to create or delete a Start menu group under Start\Programs. If this flag is not specified, Windows creates or deletes a menu item, not a menu group.

If no flag is specified, Windows creates a menu item for all users.

**CmdLine**=*dirid*[*subdir*],*filename*

The *dirid* specifies a value that identifies the directory in which the command program resides. For example, a *dirid* of 11 indicates the system directory. The possible *dirid* values are listed in the description of the *dirid* value in the [DestinationDirs](#) section.

If a *subdir* string is present, the command program is in a subdirectory of the directory referenced by *dirid*. The *subdir* specifies the subdirectory. If no *subdir* is specified, the program is in the directory referenced by *dirid*.

The *filename* specifies the name of the program associated with the menu item.

**SubDir=***path*

This optional entry specifies a subdirectory (submenu) under Start\Programs in which the menu item resides. If this entry is omitted, the path defaults to Start\Programs.

For example, if the *profile-items-section* has the entry "Subdir=Accessories\Games", then the menu item is being created or deleted in the Start\Programs\Accessories\Games submenu.

**Note** If FLG\_PROFITEM\_GROUP is specified for *name-attributes*, the SubDir entry is ignored.

**WorkingDir=***wd-dirid*[*wd-subdir*]

This optional entry specifies a working directory for the command program. If this entry is omitted, the working directory defaults to the directory in which the command program resides.

The *wd-dirid* value identifies the working directory. For lists of possible *dirid* values, see [Using Dirids](#).

The *wd-subdir* string, if present, specifies a subdirectory of *wd-dirid* to be the working directory. Use this parameter to specify a directory that does not have a system-defined *dirid*. If this parameter is omitted, the *wd-dirid* value alone specifies the working directory.

**IconPath=***icon-dirid*[*icon-subdir*],*icon-filename*

This optional entry specifies the location of a file that contains an icon for the menu item.

The *icon-dirid* string identifies the directory for the DLL that contains the icon. For lists of possible *dirid* values, see [Using Dirids](#).

The *icon-subdir* value, if present, indicates that the DLL is in a subdirectory of *icon-dirid*. The *icon-subdir* value specifies the subdirectory.

The *icon-filename* value specifies the DLL that contains the icon.

If this entry is omitted, Windows looks for an icon in the file specified in the **CmdLine** entry.

**IconIndex=***index-value*

This optional entry specifies which icon in a DLL to use for the menu item. For information about how to index the icons in a DLL, see the Microsoft Windows SDK documentation.

If an **IconPath** entry is specified, the *index-value* indexes into that DLL. Otherwise, this value indexes into the file specified in the **CmdLine** entry.

**HotKey=***hotkey-value*

This optional entry specifies a keyboard accelerator for the menu item.

For more information about hot keys, see the Windows SDK documentation.

**Infotip=***info-tip*

This optional entry specifies an informational tip for the menu item.

This value can be a string or a %strkey% token that is defined in a [Strings](#) section of the INF file.

The *info-tip* value can also be specified as "@*ResDllPath\ResDll,-ResID*", where *ResDllPath* and *ResDll* specify the path and file name of a resource DLL, and *-resID* is a negative value that represents a resource ID.

Use this format to support Windows Multilingual User Interface (MUI). An example is as follows:

```
InfoTip = "@%11%\shell32.dll,-22531"
```

#### **DisplayResource**="*ResDIIPath\ResDII*",*ResID*

This optional entry specifies a string resource that identifies a localizable string, to be used in the Start menu as the display name for shortcut or group.

*ResDIIPath* and *ResDII* specify the path and file name of a resource DLL, and *resID* is a positive value that represents a resource ID. An example is as follows:

```
DisplayResource="%11%\shell32.dll",22019
```

Use this entry to support Windows Multilingual User Interface (MUI). If this entry is not used, the string specified by the **Name** entry is displayed.

## Remarks

A given *profile-items-section* name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

Each *profile-items-section* contains detailed information for creating or removing one Start menu item or group. To manipulate more than one menu item or group from an INF, create more than one *profile-items-section* and list the sections in the **ProfileItems** directive.

Any of the string parameters specified in the *profile-items-section* entries can be specified by using a %*strkey*% token, as described in [General Syntax Rules for INF Files](#).

## Examples

The following INF file excerpt shows how to use the *profile-items-section* to add Calculator to the Start Menu.

```
[CalcInstallItems]
Name = %Calc_DESC%
CmdLine = 11,, calc.exe
SubDir = %Access_GROUP%
WorkingDir = 11
InfoTip = %Calc_TIP%
:
:
[Strings]
AccessGroup = "Accessories"
Calc_DESC = "Calculator"
Calc_TIP = "Performs basic arithmetic tasks with an on-screen calculator"
```

The following INF file excerpt shows how to install the same software by using the **DisplayResource** entry to create localized menu items.

```
[CalcInstallItems]
Name = %Calc_DESC%
CmdLine = 11,, calc.exe
SubDir = %Access_GROUP%
WorkingDir = 11
InfoTip = "@%11%\shell32.dll,-22531"
DisplayResource=%11%\shell32.dll",22019
:
:
[Strings]
Access_GROUP = "Accessories"
Calc_DESC = "Calculator"
```

## See also

[\*DDInstall\*](#)

# INF Reboot Directive

11/2/2020 • 2 minutes to read • [Edit Online](#)

A **Reboot** directive indicates that the caller should be notified to reboot the system after installation is complete.

```
[DDInstall]
```

```
Reboot
```

**Warning** The **Reboot** directive is only processed when specified directly in the **[DDInstall]** section.

The **Reboot** directive is almost never specified in INF files for installations on Windows because the need to reboot the system will automatically be detected based on the common conditions that it encounters as a part of device installation. For example, the system will notify the caller that a reboot is required if some target destination file of a file copy operation is in use, or if the device cannot be automatically restarted during the installation. The **Reboot** directive should only be used when there is some specific condition for which a system reboot is always required after the installation of this driver, which cannot be automatically detected by the system itself.

When the reboot directive is specified, the caller will be notified that a system reboot is required to complete the installation of any devices using this INF install section. When the installation has been initiated through a function such as [UpdateDriverForPlugAndPlayDevices](#), [DiInstallDriver](#), or [DiInstallDevice](#), this will result in the *NeedReboot* out parameter of these routines being set to TRUE.

## Remarks

On Windows 7 and earlier, the installation of a device using a driver with the **Reboot** directive will always result in the caller being notified that a system reboot is required to complete the installation.

On Windows 8 and above, the behavior described above only occurs when one or more of the devices to be installed are already started. Rather than restarting the devices during the installation of the new driver, the system will notify the caller that a system reboot is required to complete the installation of the new driver. If the devices to be installed are not currently started, the system will attempt to perform the installation without requiring a system reboot. Note that a reboot may still be required if one of the actions of installation still requires it. For example, if the destination file location of some file to be copied is currently in use, a system reboot will still be required to complete the installation.

# INF RegisterDLLs Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. You can use the [Reg2inf tool](#) to convert existing [INF RegisterDLLs directives](#) into [INF AddReg directives](#) in order to make a driver package universal. For more info, see [Using a Universal INF File](#).

A **RegisterDLLs** directive references one or more INF sections used to specify files that are OLE controls and require self-registration.

```
[DDInstall]  
RegisterDLLs=register-dll-section[,register-dll-section]...
```

Each INF section referenced by a **RegisterDLLs** directive must have the following entry format:

```
[register-dll-section]  
dirid,[subdir],filename,registration-flags[,timeout][,argument]]
```

A *register-dll-section* can have any number of entries, each on a separate line.

## Entries

*dirid*

Specifies the destination directory ID of the file to be registered. For more information, see [Using Dirids](#).

*subdir*

Specifies the directory path, relative to the current directory, to the file to be registered. If not specified, the file is in the current directory.

*filename*

Identifies the file name of the OLE control to be registered.

*registration-flags*

Indicates the registration operations to perform on the OLE control. One or both of the following flags must be specified.

**0x00000001 (FLG\_REGSVR\_DLLREGISTER)**

Call the OLE control's **DllRegisterServer** function (described in the Windows SDK documentation).

**0x00000002 (FLG\_REGSVR\_DLLINSTALL)**

Call the OLE control's **DllInstall** function (described in the Windows SDK documentation).

*timeout*

Specifies the time-out, in units of seconds, for an OLE Control to complete the specified registration calls. The default time-out is 60 seconds.

*argument*

If the control is an executable file, this is a command string that is passed to the executable. The default argument is **/RegServer**.

If the control is not an executable file, this specifies the command-line argument to pass to the **DllInstall** function.

## Remarks

Each *register-dll-section* name must be unique to the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The following rules apply to the use of the **RegisterDlls** directive for device installations:

- Although the syntax permits filename to be either a DLL or an executable file, for device installations only a DLL is allowed.
- The code to be registered must not prompt for user input.
- Server-side installations execute in a system context. Therefore, you must be very sure that the code being registered contains no security vulnerabilities and that file permissions prevent the code from being maliciously modified.

For more information about OLE controls and self registration, see the Windows SDK documentation.

## Examples

```
[Dialer]
RegisterDlls = DialerRegSvr
[DialerUninstall]
UnregisterDlls = DialerRegSvr
[DialerRegSvr]
11,,avtapi.dll, 1
```

## See also

[UnregisterDlls](#)

# INF RenFiles Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

A **RenFiles** directive references an INF-writer-defined section elsewhere in the INF file, which causes that list of files to be renamed in the context of operations on the section in which the referring **RenFiles** directive is specified.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)

Renfiles=file-list-section[,file-list-section]...
```

A **RenFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive can also be specified within any of the following INF-writer-defined sections:

- An *add-interface-section* referenced by the **AddInterface** directive in a **DDInstall.Interfaces** section.
- An *install-interface-section* referenced in an **InterfaceInstall32** section.

Each named section referenced by a **RenFiles** directive has one or more entries of the following form:

```
[file-list-section]

new-dest-file-name,old-source-file-name
...
```

A *file-list-section* can have any number of entries, each on a separate line.

## Entries

*new-dest-file-name*

Specifies the new name to be given to the file on the destination.

*old-source-file-name*

Specifies the old name of the file.

## Remarks

**Important** This directive must be used carefully. We highly recommend that you do not use the **RenFiles** directive in the INF file for a Plug and Play (PnP) function driver.

Any *file-list-section* name must be unique to the INF file, but it can be referenced by **CopyFiles**, **DelFiles**, or **RenFiles** directives elsewhere in the same INF. Such an INF-writer-defined section name must follow the

general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The **RenFiles** directive does not support decorating a *file-list-section* name with a system-defined platform extension (.nt, .ntx86, .ntia64, .ntamd64, .ntarm, or .ntarm64).

The **DestinationDirs** section of the INF file controls the destination for all file-rename operations, regardless of the section that contains a particular **RenFiles** directive. The following rules describe the file-rename operation:

- If a named section referenced by a **RenFiles** directive has a corresponding entry in the **DestinationDirs** section in the same INF, that entry explicitly specifies the target destination directory. All files that are listed in the named section are renamed on the destination before these source files are copied.
- If a named section is not listed in the **DestinationDirs** section, Windows uses the *DefaultDestDir* entry in the **DestinationDirs** section of the INF.

**Note** You cannot use a %strkey% token to specify the new or old file names. For more information about %strkey% tokens, see [INF Strings Section](#).

## Examples

This example shows a section referenced by a **RenFiles** directive.

```
[RenameOldFilesSec]
devfile41.sav, devfile41.sys
```

## See also

[AddInterface](#)

[ClassInstall32](#)

[CopyFiles](#)

*DDInstall*

*DDInstall.Coinstallers*

[DelFiles](#) [DestinationDirs](#)

[InterfaceInstall32](#)

[SourceDisksFiles](#)

[SourceDisksNames](#)

[Strings](#)

[Version](#)

# INF UnregisterDlls Directive

4/29/2020 • 2 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

An **UnregisterDlls** directive references one or more INF sections used to specify files that are OLE controls and require self-unregistration (self-removal).

```
[DDInstall]  
UnregisterDlls=unregister-dll-section[,unregister-dll-section]...
```

Each INF section referenced by an **UnregisterDlls** directive must have the following entry format:

```
[unregister-dll-section]  
dirid,[subdir],filename,registration-flags[,,[timeout][,,argument]]
```

An *unregister-dll-section* can have any number of entries, each on a separate line.

## Entries

*dirid*

Specifies the destination directory ID of the file to be unregistered. For more information, see [Using Dirids](#).

*subdir*

Specifies the directory path, relative to the current directory, to the file to be unregistered. If not specified, the file is in the current directory.

*filename*

Identifies the file name of the OLE control to be unregistered.

*registration-flags*

Indicates the registration operations to perform on the OLE control. One or both of the following flags must be specified.

**0x00000001** (FLG\_REGSVR\_DLLREGISTER)

Call the **DllUnRegisterServer** function (described in the Windows SDK documentation).

**0x00000002** (FLG\_REGSVR\_DLLINSTALL)

Call the OLE control's **DllInstall** function (described in the Windows SDK documentation).

*timeout*

Specifies the time-out, in units of seconds, for an OLE Control to complete the specified unregistration calls. The default time-out is 60 seconds.

*argument*

If the control is an executable file, this is a command string that is passed to the executable. The default argument is **/UnRegServer**.

If the control is not an executable file, this specifies the command-line argument to pass to the **DllInstall** function.

## Remarks

Each *unregister-dll-section* name must be unique to the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

For more information about OLE controls and self unregistration, see the Windows SDK documentation.

## Examples

```
[Dialer]
RegisterDlls = DialerRegSvr

[DialerUninstall]
UnregisterDlls = DialerRegSvr

[DialerRegSvr]
11,,avtapi.dll, 1
```

## See also

[RegisterDlls](#)

# INF UpdateIniFields Directive

4/29/2020 • 3 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

An **UpdateIniFields** directive references one or more named sections in which fine-grained modifications within the lines of an INI file can be specified.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)
```

```
UpdateIniFields=update-inifields-section[,update-inifields-section]...
```

Each named section referenced by an **UpdateIniFields** directive has the following form:

```
[update-inifields-section]
ini-file,ini-section,profile-name[,old-field][,new-field][,flags]
...
```

An *update-inifields-section* can have any INF-writer-determined number of entries, each on a separate line.

## Entries

### *ini-file*

Specifies the name of an INI file supplied on the source media and, implicitly, that of a to-be-updated INI file on the target computer. This value can be expressed as a *filename* or as a %*strkey*% token that is defined in a [Strings](#) section of the INF file.

### *ini-section*

Specifies the name of the section within the given INI files that contains the line to be modified.

### *profile-name*

Specifies the name of the line to be modified within the given INI section. At least one of the *old-field* and/or *new-field* entries must be specified to effect a modification of this line.

### *old-field*

Specifies an existing field within the given line. If *new-field* is omitted from this section entry, this field is deleted from the given line. Otherwise, the given *new-field* value should replace this field.

### *new-field*

Specifies a replacement for a given *old-field* or, if *old-field* is omitted, an addition to the given line.

### *flags*

Specifies (in bit 0) how to interpret given *old-field* and/or *new-field* if either or both contain an asterisk (\),

*and/or (in bit 1) which separator character to use when appending a given \*new-field to the given line, as follows:*

Bit zero = 0

Interpret any asterisk (\*) in the specified *old-field* and/or *new-field* entries literally, not as a wild-card character, when searching for a match in the given line of the INI file. This is the default value.

Bit zero = 1

Interpret any asterisk (\*) in the specified *old-field* and/or *new-field* entries as a wild-card character when searching for a match in the given line of the INI file.

Bit one = 0

Use a space character as a separator when adding the specified *new-field* entry to the given line of the INI file. This is the default value.

Bit one = 1

Use a comma (,) as a separator when adding the specified *new-field* entry to the given line of the INI file.

## Remarks

The **UpdateIniFields** directive is almost never specified in INF files for installations on Windows because it is not necessary to have INI files on their distribution media. However, the **UpdateIniFields** directive is valid in any of the sections shown in the formal syntax statement, as well as in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

Each *update-inifields-section* name must be unique to the INF file. Each INF-writer-created section name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

Unlike a section referenced by the **UpdateInis** directive, a section referenced by **UpdateIniFields** replaces, adds, or deletes parts of a line in an existing INI file line instead of affecting the whole value of a particular line. At least one of the *old-field* and/or *new-field* values must be specified in each section entry.

Any comments in a to-be-modified INI file line are removed because they might not be applicable after changes made according to this section. When looking for fields in the line in the INI files, spaces, tabs, and commas are interpreted as field delimiters. However, a space character is used as the default separator when a new field is appended to a line.

The INF provides the full path of the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INF files, by using the **SourceDiskNames** and **SourceDiskFiles** sections of this INF to explicitly specify the full path of each named source file that is not in the root directory (or directories) on the distribution media.
- In system-supplied INF files, by supplying one or more additional INF files, identified in the **LayoutFile** entry in the **Version** section of the INF file.

## See also

[AddInterface](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.Coinstallers](#)

[Ini2Reg](#)

**InterfaceInstall32**

**SourceDisksFiles**

**SourceDisksNames**

**Strings**

**UpdateInis**

**Version**

# INF UpdateInis Directive

4/29/2020 • 4 minutes to read • [Edit Online](#)

**Note** If you are building a universal or mobile driver package, this directive is not valid. See [Using a Universal INF File](#).

An **UpdateInis** directive references one or more named sections, specifying an INI file from which a particular section or line is to be read and applied to an existing INI file of the same name on the target computer. Optionally, line-by-line modifications from and to such INI files can be specified in the *update-ini-section*.

```
[DDInstall] |
[DDInstall.CoInstallers] |
[ClassInstall32] |
[ClassInstall32.ntx86] |
[ClassInstall32.ntia64] | (Windows XP and later versions of Windows)
[ClassInstall32.ntamd64] (Windows XP and later versions of Windows)
[ClassInstall32.ntarm] (Windows 8 and later versions of Windows)
[ClassInstall32.ntarm64] (Windows 10 and later versions of Windows)

UpdateInis=update-ini-section[,update-ini-section]...
```

This directive is almost never specified in INF files for installation on Windows, due to the lack of necessity for INI files. However, the **UpdateInis** directive is valid in any of the sections shown in the formal syntax statement, as well as in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

Each named section referenced by an **UpdateInis** directive has the following form:

```
[update-ini-section]

ini-file,ini-section[,old-ini-entry][,new-ini-entry][,flags]
...
```

An *update-ini-section* can have any INF-writer-determined number of entries, each on a separate line.

## Entries

### *ini-file*

Specifies the name of an INI file supplied on the source media and, implicitly, that of the INI file to be updated on the target computer. This value can be expressed as a *filename* or as a %*strkey*% token that is defined in a **Strings** section of the INF file.

### *ini-section*

Specifies the name of the section within the given INI file. If the next two values are specified, this section contains an entry to be changed. If an *old-ini-entry* is omitted but a *new-ini-entry* is provided, the new entry is to be added as this section is read.

### *old-ini-entry*

This optional value specifies the name of an entry in the given *ini-section*, usually expressed in the following form:

```
"key=value"
```

Either or both of *key* and *value* can be expressed as `%strkey%` tokens defined in a **Strings** section of the INF file. The asterisk (`\`) can be specified as a wild-card for either the *\*key* or the *value*.

#### *new-ini-entry*

This optional value specifies either a change to a given *old-ini-entry* or the addition of a new entry. This value can be expressed in the same manner as *old-ini-entry*.

#### *flags*

This optional value controls the interpretation of the given *old-ini-entry* and/or *new-ini-entry*. The *flags* entry can be one of the following numeric values:

VALUE	MEANING
0	<p>This is the default value for the <i>flags</i> entry if it is omitted.</p> <p>If the given <i>old-ini-entry</i> key is present in the INI files, replace that <i>key=value</i> with the given <i>new-ini-entry</i>. Only the keys in the INI files must match. The corresponding value of each such key is ignored.</p> <p>To add a <i>new-ini-entry</i> to the destination INI file unconditionally, omit the <i>old-ini-entry</i> value from the entry in the <i>update-ini</i> section of the INF.</p> <p>To delete an old-ini-entry from the destination INI file unconditionally, omit the <i>new-ini-entry</i> value.</p>
1	<p>If the given <i>old-ini-entry</i> (<i>key=value</i>) exists in the INI files, replace it in the destination INI file that has the given <i>new-ini-entry</i>. Both the <i>key</i> and <i>value</i> of the specified <i>old-ini-entry</i> must match those in the INI files for such a replacement to be made, not just their keys as for the preceding <i>flags</i> value.</p>
2	<p>If the <i>key</i> that is specified for <i>old-ini-entry</i> cannot be found in the destination INI file, do nothing. Otherwise, the changes made depend on matches found in the INI files for the given keys of <i>old-ini-entry</i> and <i>new-ini-entry</i>, as follows:</p> <ol style="list-style-type: none"><li>1. If the <i>key</i> of the <i>old-ini-entry</i> exists in the INI files but so does the <i>key</i> of the <i>new-ini-entry</i>, replace the <i>old-ini-entry</i> with the <i>new-ini-entry</i> in the destination INI file and, then, remove the superfluous <i>new-ini-entry</i> from that INI file.</li><li>2. If the <i>key</i> of the <i>old-ini-entry</i> exists in the INI files but the <i>key</i> of the <i>new-ini-entry</i> does not, replace the <i>old-ini-entry</i> <i>key</i> with that of the <i>new-ini-entry</i> in the destination INI file but leave the value of the <i>old-ini-entry</i> unchanged.</li></ol>

VALUE	MEANING
3	<p>If the <i>key</i> and <i>value</i> specified for <i>old-ini-entry</i> cannot be found in the INI files, do nothing. Otherwise, the changes made depend on matches found in the INI files for the given <i>keys</i> and <i>values</i> of <i>old-ini-entry</i> and <i>new-ini-entry</i>, as follows:</p> <ol style="list-style-type: none"> <li>1. If the <i>key=value</i> of the <i>old-ini-entry</i> exists in the INI files but so does the <i>key=value</i> of the <i>new-ini-entry</i>, replace the <i>old-ini-entry</i> with the <i>new-ini-entry</i> in the destination INI file and, then, remove the superfluous <i>new-ini-entry</i> from that INI file.</li> <li>2. If the <i>key=value</i> of the <i>old-ini-entry</i> exists in the INI files but the <i>new-ini-entry</i> does not, replace the <i>old-ini-entry</i> with the <i>new-ini-entry</i> in the destination INI file but leave the value of the <i>old-ini-entry</i> unchanged.</li> </ol>

## Remarks

A given *update-ini-section* name must be unique within the INF file and must follow the general rules for defining section names. For more information about these rules, see [General Syntax Rules for INF Files](#).

The INF provides the full path of the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INF files, by using the [SourceDiskNames](#) and [SourceDiskFiles](#) sections of this INF to explicitly specify the full path of each named source file that is not in the root directory (or directories) on the distribution media.
- In system-supplied INF files, by supplying one or more additional INF files, identified in the [LayoutFile](#) entry in the [Version](#) section of the INF file.

Any *filename* specified within an *old-ini-entry* or *new-ini-entry* should designate the destination directory that contains that file. Such a destination directory path of a *filename* in an *update-ini-section* entry must be specified as a *dirid*. For lists of possible *dirid* values, see [Using Dirids](#).

## See also

[AddInterface](#)

[ClassInstall32](#)

[DDInstall](#)

[DDInstall.ColInstallers](#)

[DestinationDirs](#)

[Ini2Reg](#)

[InterfaceInstall32](#)

[ProfileItems](#)

[SourceDiskFiles](#)

[SourceDiskNames](#)

[Strings](#)

[UpdateIniFields](#)

[Version](#)

# Accessing Device Properties

12/1/2020 • 2 minutes to read • [Edit Online](#)

You must not discover or change [device properties](#) by directly accessing registry keys. Registry keys do not contain required information to discover or change device properties. In addition, the location, format, and meaning of these keys might change between different versions of Windows.

The [SetupAPI](#) functions provide consistent behavior and enforce access permissions to protect device properties. Starting with Windows Vista, device properties that have restricted write access also have restricted read access.

To safely access device properties, follow these guidelines:

- For user-mode applications, follow these steps:

1. Starting with Windows Vista, use [SetupDiGetDeviceProperty](#) to retrieve device properties, and use [SetupDiSetDeviceProperty](#) with DEVPKEY\_Xxx property codes to set device properties.

For more information about device instance properties on Windows Vista and later versions of Windows, see [Accessing Device Instance Properties \(Windows Vista and Later\)](#).

**Note** Starting with Windows Vista, some device properties are reserved by the operating system. For more information, see [Modifying Device Properties](#).

2. On Windows 2000, Windows XP, and Windows Server 2003, use [SetupDiGetDeviceRegistryProperty](#) to retrieve device properties, and use [SetupDiSetDeviceRegistryProperty](#) with SPDRP\_Xxx property codes to set device properties.

For more information about device instance properties on Windows 2000, Windows XP, and Windows Server 2003, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

3. Use persistent storage within the registry for custom settings of devices that are physically present and for those that are not. In this case, you must create your own set of registry keys and values. To do this, use [SetupDiCreateDevRegKey](#) (to create a new registry key) or use [SetupDiOpenDevRegKey](#) (to open an existing registry key). In these functions, the *KeyType* parameter is used to specify a *hardware key* (DIREG\_DEV) or *software key* (DIREG\_DRV) for the device.

**Note** Hardware keys persist in the registry until the device is uninstalled. Software keys can be moved or cleared by the [device installation components](#) during a driver upgrade

To save the custom settings, use [RegCloseKey](#) after the registry key has been created or opened.

- For kernel-mode drivers, use [IoGetDeviceProperty](#) to access device properties.

# Rules for Modifying Device Properties

11/2/2020 • 2 minutes to read • [Edit Online](#)

Many [device properties](#) have complex dependencies on other properties or device state. For example, the values of [DEVPKEY\\_Device\\_Class](#) and [DEVPKEY\\_Device\\_ClassGuid](#) must be consistent with one another.

Direct modification of reserved properties could invalidate device installation state. For example, if the [DEVPKEY\\_Device\\_DeviceDesc](#) is changed, system functionality (such as backup, driver rollback, and Windows Update) could break.

The following properties are read-only and can never be set with [SetupDiSetDeviceProperty](#):

- [DEVPKEY\\_Device\\_Address](#)
- [DEVPKEY\\_Device\\_BusNumber](#)
- [DEVPKEY\\_Device\\_BusTypeGuid](#)
- [DEVPKEY\\_Device\\_Capabilities](#)
- [DEVPKEY\\_Device\\_EnumeratorName](#)
- [DEVPKEY\\_Device\\_LegacyBusType](#)
- [DEVPKEY\\_Device\\_PDOName](#)
- [DEVPKEY\\_Device\\_PowerData](#)
- [DEVPKEY\\_Device\\_RemovalPolicy](#)
- [DEVPKEY\\_Device\\_RemovalPolicyDefault](#)
- [DEVPKEY\\_Device\\_UINumber](#)

The following properties are writable. However, they are reserved for use by the operating system and must not be set directly:

- [DEVPKEY\\_Device\\_Class](#)
- [DEVPKEY\\_Device\\_ClassGuid](#)
- [DEVPKEY\\_Device\\_Driver](#)
- [DEVPKEY\\_Device\\_DriverDesc](#)
- [DEVPKEY\\_Device\\_Manufacturer](#)

**Note** *Class installers* and *co-installers* must not change device properties except for the friendly name ([DEVPKEY\\_Device\\_FriendlyName](#)) and the upper and lower filter drivers for the device ([DEVPKEY\\_Device\\_UpperFilters](#) and [DEVPKEY\\_Device\\_LowerFilters](#)). For more information, see [Accessing Device Instance Properties](#).

# Accessing and Modifying Files

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following guidelines apply to driver package components when they access or modify files:

- Files should be modified only by using [INF directives](#) within an INF file. For example, use the INF [CopyFiles](#) directive to copy files and the INF [RenFiles](#) directive to rename files.
- Files that appear in an INF [CopyFiles](#) directive must not also appear in an INF [RenFiles](#) or [DelFiles](#) directive in the INF file.

**Important** The INF [RenFiles](#) and [DelFiles](#) directives must be used carefully. You should not use these directives in the INF file for a Plug and Play (PnP) function driver.

# Accessing Registry Keys Safely

12/1/2020 • 2 minutes to read • [Edit Online](#)

Customer problems have frequently been traced to external components, such as third-party [device installation applications](#), that do the following:

- Delete critical registry keys.
- Modify the access permissions of critical registry keys.

Many of the problems seen with external components are caused by using the KEY\_ALL\_ACCESS access permission for registry keys. Starting with Windows Server 2003, [SetupDiCreateDevRegKey](#) grants only KEY\_READ and KEY\_WRITE access permissions and not KEY\_ALL\_ACCESS. Starting with Windows Vista, additional KEY\_ALL\_ACCESS restrictions are enforced.

Follow these guidelines to safely access registry keys:

- Use the [SetupAPI](#) functions only to open registry keys, especially the *hardware keys* and *software keys* for a device.

These functions address common problems that result from restrictions on access permissions.

- The location and format of registry keys might change between different versions of Windows. Do not make assumptions about the location, format, or meaning of registry keys or values that are used for device and driver installation.

For more information about registry keys and trees, see [Registry Trees and Keys for Devices and Drivers](#).

- Do not use the registry to directly access or modify the internal settings of the device.
- Request only the minimal access permissions that are required for each task, such as the following:
  - KEY\_SET\_VALUE
  - KEY\_CREATE\_SUB\_KEY
  - KEY\_QUERY\_VALUE
  - KEY\_ENUMERATE\_SUB\_KEYS

- Do not directly open the device setup class keys in the registry. As with any registry key, the location and name of device setup class keys might change between versions of Windows.

To open device setup class keys safely, follow these guidelines:

- Use [SetupDiOpenClassRegKey](#).
- Use [SetupDiOpenClassRegKeyEx](#) and set DIOCR\_INSTALLER in the *Flags* parameter.
- Do not directly open device interface class keys in the registry. As with any registry key, the location and name of device interface class keys might change between versions of Windows.

To open device interface class keys safely, use [SetupDiOpenClassRegKeyEx](#) and set DIOCR\_INSTALLER in the *Flags* parameter.

- Use only INF directives to modify registry keys that are reserved for use by the operating system. For more information, see [Summary of INF Directives](#).

- *Class installers* and *co-installers* cannot call registry functions to create, change, or delete registry values that are reserved for use by the operating system.

For more information about the access permissions of registry keys, see [Registry Key Security and Access Rights](#).

# Opening a Device's Hardware Key

11/2/2020 • 2 minutes to read • [Edit Online](#)

A *hardware key* is device-specific registry subkey that contains information about the device. You must not directly open a device's hardware key. As with any registry key, the location or format of these keys might change between different versions of Windows.

**Note** You should open a device's hardware key only after the corresponding device has been found. For more information about this procedure, see [Enumerating Installed Devices](#).

To open or create a device's hardware key, follow these guidelines:

- To open an existing hardware key, use [SetupDiOpenDevRegKey](#). To create a hardware key, use [SetupDiCreateDevRegKey](#). In either case, you must set the *KeyType* parameter to DIREG\_DEV.

**Note** You must set the *samDesired* parameter to the minimal access permissions that are required. You must not set this parameter to KEY\_ALL\_ACCESS. For more information about how to specify access permissions for registry access, see [Accessing Registry Keys Safely](#).

- Kernel-mode callers should use [IoOpenDeviceRegistryKey](#) and set the *DevInstKeyType* parameter to PLUGPLAY\_REGKEY\_DEVICE.

# Opening a Device's Software Key

11/2/2020 • 2 minutes to read • [Edit Online](#)

You must not directly open a device's *software key*. As with any registry key, the location or format of these keys might change between different versions of Windows.

**Note** You should open a device's software key only after the corresponding device has been found. For more information about this procedure, see [Enumerating Installed Devices](#).

To open a device's software key, follow these guidelines:

- To open an existing software key, use [SetupDiOpenDevRegKey](#). To create a software key, use [SetupDiCreateDevRegKey](#). In either case, you must set the *KeyType* parameter to DIREG\_DRV.

**Note** You must set the *samDesired* parameter to the minimal access permissions that are required. You must not set this parameter to KEY\_ALL\_ACCESS. For more information about how to specify access permissions for registry access, see [Accessing Registry Keys Safely](#).

- Kernel-mode callers should use [IoOpenDeviceRegistryKey](#) and set the *DevInstKeyType* parameter to PLUGPLAY\_REGKEY\_DRIVER.

# Modifying Registry Values in a Device's Software Key

11/2/2020 • 2 minutes to read • [Edit Online](#)

You must not modify the values of the following registry entries (*device properties*) in a device's *software key*:

- DriverDate
- DriverDateData
- DriverDesc
- DriverVersion
- InfPath
- InfSection
- InfSectionExt
- MatchingDeviceId
- ProviderName
- EnumPropPages32

These device properties represent a device's installation state. Direct modification of these properties might invalidate the device's installation state. For example, changing information related to the [INF file](#) invalidates information about driver files that are associated with such properties as device and driver signing information. Changing driver version or driver date might break Windows Update functionality.

**Note** Starting with Windows Vista, the operating system imposes "installation-time only" access restrictions for these properties. Values can be replicated for compatibility, and direct modification of values during device installation does not affect internal state.

To safely modify the values of other registry entries in a device's software key, follow these guidelines:

- Use [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#) to retrieve and set standard or custom properties.

For more information, see [Accessing Device Driver Properties](#).

- Set only those device properties that are not reserved by the operating system.

For example, to change the name of the device as displayed to the user, change its [DEVPKEY\\_Device\\_FriendlyName](#) device property.

# Enumerating Installed Device Setup Classes

11/2/2020 • 2 minutes to read • [Edit Online](#)

To discover the [device setup classes](#) that are installed in a system, do not enumerate the device setup classes by directly accessing registry keys. As with any registry key, the location and format of these keys might change between different versions of Windows.

To safely discover the installed device setup classes, and to query and modify the properties of a setup class, follow these steps:

1. Use [SetupDiBuildClassInfoList](#) or [SetupDiBuildClassInfoListEx](#) to retrieve the set of device setup classes that are currently installed on the system.
2. Use [SetupDiGetClassDescription](#) or [SetupDiGetClassDescriptionEx](#) to retrieve the description of an installed setup class.
3. Use [SetupDiGetClassRegistryProperty](#) to query the setup class properties and [SetupDiSetDeviceRegistryProperty](#) to set the setup class properties.
4. Use [SetupDiOpenClassRegKey](#) or [SetupDiOpenClassRegKeyEx](#) to access the persistent registry storage for custom device setup class settings.

# Opening Registry Keys for a Device Setup Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

Do not directly access the registry keys for device setup classes. As with any registry key, the location and name of these keys might change between different versions of Windows.

To safely open the registry keys of a [device setup class](#), use one of the following [SetupAPI](#) functions:

- [SetupDiOpenClassRegKey](#)
- [SetupDiOpenClassRegKeyEx](#) with the *Flags* parameter set to DIOCR\_INSTALLER

# Opening Software Keys for All Devices in a Setup Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

When a user-mode application opens the *software keys* for all devices in a device setup class, it must not directly access the registry to enumerate the subkeys of a device setup class. As with any registry key, the location and name of this key might change between different versions of Windows.

To safely enumerate and open the subkeys of a device setup class, follow these steps:

1. Use [SetupDiGetClassDevs](#) or [SetupDiGetClassDevsEx](#) to retrieve a set of information about all devices for a specified device setup class.
2. Use [SetupDiEnumDeviceInfo](#) to enumerate all devices in the set.
3. Use [SetupDiOpenDevRegKey](#) to open the software key for each device. The *KeyType* parameter must be set to DIREG\_DRV.

**Note** Some devices might not have software keys, such as when a device is present and enumerated by the [Plug and Play \(PnP\) manager](#) but has not been installed.

# Enumerating Installed Device Interfaces

11/2/2020 • 2 minutes to read • [Edit Online](#)

You must not enumerate the device interface classes in a system by directly accessing registry keys. As with any registry key, the location, name, or format of the key might change between different versions of Windows.

Use the following guidelines to safely discover the attributes of device interfaces:

- User-mode applications should follow these steps:
  1. Use [SetupDiGetClassDevs](#) or [SetupDiGetClassDevsEx](#) to retrieve the devices that support interfaces for the specified device interface class. You must set the `DIGCF_DEVICEINTERFACE` flag in the *Flags* parameter, and you must set the *Enumerator* parameter to a specific device instance identifier.

To include only device interfaces that are present in a system, set the `DIGCF_PRESENT` flag in the *Flags* parameter.
  2. Use [SetupDiEnumDeviceInterfaces](#) to enumerate interfaces that are registered for a device interface class. This interface class is specified through the *InterfaceClassGuid* parameter.
- Kernel-mode drivers should use [IoGetDeviceInterfaces](#) to enumerate the device interface classes that are installed in the system.

# Accessing the Properties of Installed Device Interfaces

11/2/2020 • 2 minutes to read • [Edit Online](#)

To discover the attributes of device interfaces that are registered on the system, you must not open, read, or write the device interface subkeys by directly accessing the registry. As with any registry key, the location, name, or format of the key might change between different versions of Windows.

Use the following guidelines to safely query and modify the attributes of device interfaces:

- User-mode applications should follow these steps:
  1. Use [SetupDiOpenDeviceInterface](#) to locate a device interface and add it to a set from its name.
  2. Use [SetupDiGetDeviceInterfaceDetail](#) to retrieve details for the device interface.

The optional *DeviceInfoData* parameter will receive the [SP\\_DEVINFO\\_DATA](#) element for the device for which the interface is registered.

3. Use persistent registry storage for the custom settings for a device interface class. To do this, use [SetupDiCreateDeviceInterfaceRegKey](#) (to create a new registry key) or [SetupDiOpenDeviceInterfaceRegKey](#) (to open an existing registry key).

To save the custom settings, use [RegCloseKey](#) after the registry key has been created or opened.

- Kernel-mode drivers should use [IoOpenDeviceInterfaceRegistryKey](#) to open the registry key for a device interface class.

# Modifying Registry Keys by Class Installers and Co-installers

12/1/2020 • 2 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

Except under certain conditions, *class installers* and *co-installers* should not use the standard registry functions to create, change, or delete registry keys. In most cases, registry keys should only be modified by using directives that are put in [INF files](#). For more information about these directives, see [Summary of INF Directives](#).

The following are exceptions to this rule:

- When it is necessary, class installers and co-installers can use the standard registry functions to modify registry keys in the **HKLM\Software** subtree.

**Note** We highly recommend that class installers and co-installers save custom device properties as entries within the device's *software keys*.

- Class installers and co-installers are permitted to modify subkeys in the **HKLM\System\CurrentControlSet\Control\CoDeviceInstallers** registry key.

The following guidelines should be followed to safely modify registry keys by class installers or co-installers:

- Class installers and co-installers must first use [SetupDiCreateDevRegKey](#) or [SetupDiOpenDevRegKey](#) to open handles to the registry keys that will be modified. After a handle has been opened, class installers and co-installers can use the standard registry functions to modify registry keys.
- Class installers and co-installers must not use [SetupDiDeleteDevRegKey](#) to delete *software keys* or *hardware keys* for the device. For more information, see [Deleting the Registry Keys of a Device](#).

For more information about the standard registry functions, see [Registry Functions](#).

# Modifying Registry Values by Class Installers and Co-installers

12/1/2020 • 2 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

Except under certain conditions, *class installers* and *co-installers* must not call the standard registry functions to create, change, or delete registry values that are reserved for use by the operating system.

The following are exceptions to this rule:

- When it is necessary, class installers and co-installers can use the standard registry functions to modify registry values in the **HKLM\Software** subtree.

**Note** We highly recommend that class installers and co-installers save custom device properties as entries within the device's *software keys*.

- Class installers and co-installers can modify the value of the [RunOnce registry key](#). However, this value must consist of only calls to *Rundll32.exe*.

Class installers and co-installers must follow the restrictions about how to use the [RunOnce registry key](#) in [INF files](#). In particular, this registry key must be used only for the installation of software-only devices that are enumerated by using the software device enumerator (SWENUM).

- Class installers and co-installers can modify the **CoInstallers32** and **EnumPropPages32** registry values in the device's *software key* when the installer handles [DIF\\_REGISTER\\_COINSTALLERS](#) requests.

The following guidelines should be followed to safely modify registry values by class installers or co-installers:

- Class installers and co-installers must first use [SetupDiCreateDevRegKey](#) or [SetupDiOpenDevRegKey](#) to open handles to the registry keys that will be modified. After a handle has been opened, class installers and co-installers can use the standard registry functions to modify registry values.
- Class installers and co-installers must not use [SetupDiDeleteDevRegKey](#) to delete *software keys* or *hardware keys* for the device.

For more information about the standard registry functions, see [Registry Functions](#).

# Deleting the Registry Keys of a Device

12/1/2020 • 2 minutes to read • [Edit Online](#)

You should not use [SetupDiDeleteDevRegKey](#) to delete the *software keys* or *hardware keys* for the device for the following reasons:

- [SetupDiDeleteDevRegKey](#) removes all custom settings in registry keys. This includes the following:
  - Settings that were specified during installation.
  - Settings that were created or modified by the device driver.
  - Settings that were created or modified by applications or other components.

[SetupDiDeleteDevRegKey](#) also removes critical device installation state.

- Software or hardware keys that are opened by using [SetupDiOpenDevRegKey](#) with a scope of DICS\_FLAG\_GLOBAL contain data about the device installation state. Software or hardware keys that are accessed with a scope of DICS\_FLAG\_CONFIGSPECIFIC do not contain device installation state.

In either case, deleting these software or hardware keys could have implications for other device installation components.

You should not make assumptions about whether device registry keys are present. When the device is uninstalled, the system automatically deletes all software and hardware keys for the device.

You can safely create and delete registry subkeys under the hardware or software keys by using standard registry functions. By using these functions, you avoid naming collisions between the system and other components. Also, if you create subkeys by using these functions, the subkey inherits the default permissions of the parent registry key. For more information, see [Opening, Creating, and Closing Keys](#).

For more information about the standard registry functions, see [Registry Functions](#).

# Calling SetupAPI Functions

12/1/2020 • 2 minutes to read • [Edit Online](#)

This section provides guidelines that you should follow when you call the [SetupAPI](#) functions from a [co-installer](#) or a [device installation application](#).

- [Rules for calling SetupAPI functions](#)
- [Rules for calling the default DIF code handler functions](#)

## Rules for calling SetupAPI functions

Class installers and co-installers must not call the following [SetupAPI](#) functions:

- [SetupQueueCopy](#)
- [SetupQueueCopyIndirect](#)
- [SetupQueueCopySection](#)
- [SetupQueueDefaultCopy](#)
- [SetupQueueDelete](#)
- [SetupQueueDeleteSection](#)
- [SetupQueueRename](#)
- [SetupQueueRenameSection](#)
- [SetupScanFileQueue](#)

**Note** Class installers and co-installers are prohibited from calling [SetupScanFileQueue](#) only when the SPQ\_SCAN\_PRUNE\_COPY\_QUEUE flag is set in the *Flags* parameter.

## Rules for calling the default DIF code handler functions

Default [device installation function \(DIF\) code](#) handler functions perform system-defined default operations for certain DIF codes. As described in [Handling DIF Codes](#), [SetupDiCallClassInstaller](#) calls the default handler for a DIF request after the *co-installer* has first processed the DIF request, but before [SetupDiCallClassInstaller](#) calls the co-installers that registered for post-processing of the request.

[Co-installers](#) and [device installation applications](#) must not call the default DIF code handler functions. Direct calls to these handler functions bypass all registered co-installers and could invalidate any internal device state that these installers store.

The following table lists the DIF codes that have default DIF code handler functions.

DIF CODE	DEFAULT DIF CODE HANDLER FUNCTION
<a href="#">DIF_PROPERTYCHANGE</a>	<a href="#">SetupDiChangeState</a>
<a href="#">DIF_FINISHINSTALL_ACTION</a>	<a href="#">SetupDiFinishInstallAction</a>
<a href="#">DIF_INSTALLDEVICE</a>	<a href="#">SetupDiInstallDevice</a>
<a href="#">DIF_INSTALLINTERFACES</a>	<a href="#">SetupDiInstallDeviceInterfaces</a>

DIF CODE	DEFAULT DIF CODE HANDLER FUNCTION
DIF_INSTALLDEVICEFILES	<a href="#">SetupDiInstallDriverFiles</a>
DIF_REGISTER_COINSTALLERS	<a href="#">SetupDiRegisterCoDeviceInstallers</a>
DIF_REGISTERDEVICE	<a href="#">SetupDiRegisterDeviceInfo</a>
DIF_REMOVE	<a href="#">SetupDiRemoveDevice</a>
DIF_SELECTBESTCOMPATDRV	<a href="#">SetupDiSelectBestCompatDrv</a>
DIF_SELECTDEVICE	<a href="#">SetupDiSelectDevice</a>
DIF_UNREMOVE	<a href="#">SetupDiUnremoveDevice</a>

# Enumerating Installed Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

You should not enumerate devices by using registry keys directly. Registry keys do not contain the required information to enumerate installed devices on the system. This information, such as whether the device is actually present or is a phantom device (one that is not plugged in), is held by the [Plug and Play \(PnP\) manager](#). The PnP manager also performs additional filtering of registry information.

To enumerate installed devices safely, follow these steps:

1. Use [SetupDiGetClassDevs](#) or [SetupDiGetClassDevsEx](#) to retrieve information for a set of devices that belong to a specified device setup class. To retrieve information only for devices that are present in the system, set DIGCF\_PRESENT in the *Flags* parameter.
2. Use [SetupDiEnumDeviceInfo](#) to enumerate the devices in the set.
3. Use [SetupDiGetDeviceInstanceId](#) to retrieve unique device instance identifiers (IDs).

# General Guidelines for Device and Driver Installation

12/1/2020 • 2 minutes to read • [Edit Online](#)

The fundamental goal for device and driver installation on Windows operating systems is to make the process as easy as possible for the user. Your installation procedures and the components of your [driver packages](#) should work seamlessly with the operating system's [device installation components](#).

To provide the best possible user experience, use the following guidelines to design and implement your installation procedures:

- Do not automatically restart the system or require the user to do so, unless it is absolutely necessary.
- Always use [INF files](#) for device installation. Make sure that all INF files are well formed and use correct syntax.
- Leave your INF files on the system after installation; do not delete them. The INF file is used not only when the device or driver is first installed, but also when the user requests a driver update through Device Manager.
- Use one of the [System-Defined Device Setup Classes](#). Do not define your own setup class unless there is a compelling reason to do so.
- Do not make assumptions about the location, format, or meaning of registry keys or values. For more information about registry keys and trees, see [Registry Trees and Keys for Devices and Drivers](#).

# Porting code from SetupApi to CfgMgr32

11/2/2020 • 6 minutes to read • [Edit Online](#)

This topic provides code examples that show how to port code that uses `Setupapi.dll` functionality to use `Cfgmgr32.dll` instead. Porting your code allows you to run your code on the Universal Windows Platform (UWP), which does not support `SetupApi`. A subset of `CfgMgr32` is supported on UWP, specifically functionality exposed through the `api-ms-win-devices-config-11-1-0.dll` API set (Windows 8 and later) or the `api-ms-win-devices-config-11-1-1.dll` API set (Windows 8.1 and later). In Windows 10 and later, simply link to `oncore.lib`.

To see a list of functions in the above API sets, please refer to [Windows API Sets](#) or [Oncore.lib: APIs from api-ms-win-devices-config-11-1-1.dll](#).

The following sections include code examples that applications would typically use.

- [Get a list of present devices and retrieve a property for each device](#)
- [Get a list of interfaces, get the device exposing each interface, and get a property from the device](#)
- [Get a property from a specific device](#)
- [Disable device](#)
- [Enable device](#)
- [Restart device](#)

## Get a list of present devices and retrieve a property for each device

This example gets a list of all present devices using `SetupDiGetClassDevs` and iterates through them to retrieve the device description of each device.

```

VOID
GetDevicePropertiesSetupapi(
    VOID
)
{
    HDEVINFO DeviceInfoSet = INVALID_HANDLE_VALUE;
    SP_DEVINFO_DATA DeviceInfoData;
    DWORD Index;
    WCHAR DeviceDesc[2048];
    DEVPROPTYPE PropertyType;

    DeviceInfoSet = SetupDiGetClassDevs(NULL,
                                         NULL,
                                         NULL,
                                         DIGCF_ALLCLASSES | DIGCF_PRESENT);

    if (DeviceInfoSet == INVALID_HANDLE_VALUE)
    {
        goto Exit;
    }

    ZeroMemory(&DeviceInfoData, sizeof(DeviceInfoData));
    DeviceInfoData.cbSize = sizeof(DeviceInfoData);

    for (Index = 0;
         SetupDiEnumDeviceInfo(DeviceInfoSet,
                               Index,
                               &DeviceInfoData);
         Index++)
    {
        // Query a property on the device. For example, the device description.
        if (!SetupDiGetDeviceProperty(DeviceInfoSet,
                                      &DeviceInfoData,
                                      &DEVPKEY_Device_DeviceDesc,
                                      &.PropertyType,
                                      (PBYTE)DeviceDesc,
                                      sizeof(DeviceDesc),
                                      NULL,
                                      0))
        {
            // The error can be retrieved with GetLastError();
            continue;
        }

        if (.PropertyType != DEVPROP_TYPE_STRING)
        {
            continue;
        }
    }

    if (GetLastError() != ERROR_NO_MORE_ITEMS)
    {
        goto Exit;
    }

Exit:

    if (DeviceInfoSet != INVALID_HANDLE_VALUE)
    {
        SetupDiDestroyDeviceInfoList(DeviceInfoSet);
    }

    return;
}

```

This example gets a list of all present devices using [CM\\_Get\\_Device\\_ID\\_List](#) and iterates through them to retrieve

the device description of each device.

```
VOID
GetDevicePropertiesCfgmgr32(
    VOID
)
{
    CONFIGRET cr = CR_SUCCESS;
    PWSTR DeviceList = NULL;
    ULONG DeviceListLength = 0;
    PWSTR CurrentDevice;
    DEVINST Devinst;
    WCHAR DeviceDesc[2048];
    DEVPROPTYPE PropertyType;
    ULONG PropertySize;
    DWORD Index = 0;

    cr = CM_Get_Device_ID_List_Size(&DeviceListLength,
                                   NULL,
                                   CM_GETIDLIST_FILTER_PRESENT);

    if (cr != CR_SUCCESS)
    {
        goto Exit;
    }

    DeviceList = (PWSTR)HeapAlloc(GetProcessHeap(),
                                 HEAP_ZERO_MEMORY,
                                 DeviceListLength * sizeof(WCHAR));

    if (DeviceList == NULL) {
        goto Exit;
    }

    cr = CM_Get_Device_ID_List(NULL,
                               DeviceList,
                               DeviceListLength,
                               CM_GETIDLIST_FILTER_PRESENT);

    if (cr != CR_SUCCESS)
    {
        goto Exit;
    }

    for (CurrentDevice = DeviceList;
         *CurrentDevice;
         CurrentDevice += wcslen(CurrentDevice) + 1)
    {

        // If the list of devices also includes non-present devices,
        // CM_LOCATE_DEVNODE_PHANTOM should be used in place of
        // CM_LOCATE_DEVNODE_NORMAL.
        cr = CM_Locate_DevNode(&Devinst,
                              CurrentDevice,
                              CM_LOCATE_DEVNODE_NORMAL);

        if (cr != CR_SUCCESS)
        {
            goto Exit;
        }

        // Query a property on the device. For example, the device description.
        PropertySize = sizeof(DeviceDesc);
        cr = CM_Get_DevNode_Property(Devinst,
                                    &DEVPKEY_Device_DeviceDesc,
                                    &.PropertyType,
                                    (PBYTE)DeviceDesc,
                                    &PropertySize,
```

```

        0);

    if (cr != CR_SUCCESS)
    {
        Index++;
        continue;
    }

    if (.PropertyType != DEVPROP_TYPE_STRING)
    {
        Index++;
        continue;
    }

    Index++;
}

Exit:

if (DeviceList != NULL)
{
    HeapFree(GetProcessHeap(),
        0,
        DeviceList);
}

return;
}

```

Get a list of interfaces, get the device exposing each interface, and get a property from the device

This example gets a list of all interfaces in class GUID\_DEVINTERFACE\_VOLUME using [SetupDiGetClassDevs](#). For each interface, it gets the device exposing the interface and gets a property of that device.

```

VOID
GetInterfacesAndDevicePropertySetupapi(
    VOID
)
{
    HDEVINFO DeviceInfoSet = INVALID_HANDLE_VALUE;
    SP_DEVICE_INTERFACE_DATA DeviceInterfaceData;
    SP_DEVINFO_DATA DeviceInfoData;
    DWORD Index;
    WCHAR DeviceDesc[2048];
    DEVPROPTYPE PropertyType;

    DeviceInfoSet = SetupDiGetClassDevs(&GUID_DEVINTERFACE_VOLUME,
        NULL,
        NULL,
        DIGCF_DEVICEINTERFACE);

    if (DeviceInfoSet == INVALID_HANDLE_VALUE)
    {
        goto Exit;
    }

    ZeroMemory(&DeviceInterfaceData, sizeof(DeviceInterfaceData));
    DeviceInterfaceData.cbSize = sizeof(DeviceInterfaceData);

    for (Index = 0;
        SetupDiEnumDeviceInterfaces(DeviceInfoSet,
            NULL,
            &GUID_DEVINTERFACE_VOLUME,
            Index,

```

```

        &DeviceInterfaceData);
    Index++)
{
    ZeroMemory(&DeviceInfoData, sizeof(DeviceInfoData));
    DeviceInfoData.cbSize = sizeof(DeviceInfoData);

    if ((!SetupDiGetDeviceInterfaceDetail(DeviceInfoSet,
                                          &DeviceInterfaceData,
                                          NULL,
                                          0,
                                          NULL,
                                          &DeviceInfoData)) &&
        (GetLastError() != ERROR_INSUFFICIENT_BUFFER))
    {
        // The error can be retrieved with GetLastError();
        goto Exit;
    }

    // Query a property on the device. For example, the device description.
    if (!SetupDiGetDeviceProperty(DeviceInfoSet,
                                 &DeviceInfoData,
                                 &DEVPKEY_Device_DeviceDesc,
                                 &.PropertyType,
                                 (PBYTE)DeviceDesc,
                                 sizeof(DeviceDesc),
                                 NULL,
                                 0))
    {
        // The error can be retrieved with GetLastError();
        goto Exit;
    }

    if (.PropertyType != DEVPROP_TYPE_STRING)
    {
        goto Exit;
    }
}

if (GetLastError() != ERROR_NO_MORE_ITEMS)
{
    goto Exit;
}

Exit:

if (DeviceInfoSet != INVALID_HANDLE_VALUE)
{
    SetupDiDestroyDeviceInfoList(DeviceInfoSet);
}

return;
}

```

This example gets a list of all interfaces in class GUID\_DEVINTERFACE\_VOLUME using [CM\\_Get\\_Device\\_Interface\\_List](#). For each interface, it gets the device exposing the interface and gets a property of that device.

```

VOID
GetInterfacesAndDevicePropertyCfgmgr32(
    VOID
)
{
    CONFIGRET cr = CR_SUCCESS;
    PWSTR DeviceInterfaceList = NULL;
    ULONG DeviceInterfaceListLength = 0;

```

```

PWSTR CurrentInterface;
WCHAR CurrentDevice[MAX_DEVICE_ID_LEN];
DEVINST Devinst;
WCHAR DeviceDesc[2048];
DEVPROPTYPE PropertyType;
ULONG PropertySize;
DWORD Index = 0;

do {
    cr = CM_Get_Device_Interface_List_Size(&DeviceInterfaceListLength,
                                           (LPGUID)&GUID_DEVINTERFACE_VOLUME,
                                           NULL,
                                           CM_GET_DEVICE_INTERFACE_LIST_ALL_DEVICES);

    if (cr != CR_SUCCESS)
    {
        break;
    }

    if (DeviceInterfaceList != NULL) {
        HeapFree(GetProcessHeap(),
                 0,
                 DeviceInterfaceList);
    }

    DeviceInterfaceList = (PWSTR)HeapAlloc(GetProcessHeap(),
                                           HEAP_ZERO_MEMORY,
                                           DeviceInterfaceListLength * sizeof(WCHAR));

    if (DeviceInterfaceList == NULL)
    {
        cr = CR_OUT_OF_MEMORY;
        break;
    }

    cr = CM_Get_Device_Interface_List((LPGUID)&GUID_DEVINTERFACE_VOLUME,
                                      NULL,
                                      DeviceInterfaceList,
                                      DeviceInterfaceListLength,
                                      CM_GET_DEVICE_INTERFACE_LIST_ALL_DEVICES);
} while (cr == CR_BUFFER_SMALL);

if (cr != CR_SUCCESS)
{
    goto Exit;
}

for (CurrentInterface = DeviceInterfaceList;
     *CurrentInterface;
     CurrentInterface += wcslen(CurrentInterface) + 1)
{

    PropertySize = sizeof(CurrentDevice);
    cr = CM_Get_Device_Interface_Property(CurrentInterface,
                                          &DEVPKEY_Device_InstanceId,
                                          &.PropertyType,
                                          (PBYTE)CurrentDevice,
                                          &PropertySize,
                                          0);

    if (cr != CR_SUCCESS)
    {
        goto Exit;
    }

    if (.PropertyType != DEVPROP_TYPE_STRING)
    {
        goto Exit;
    }
}

```

```

// Since the list of interfaces includes all interfaces, enabled or not, the
// device that exposed that interface may currently be non-present, so
// CM_LOCATE_DEVNODE_PHANTOM should be used.
cr = CM_Locate_DevNode(&Devinst,
                       CurrentDevice,
                       CM_LOCATE_DEVNODE_PHANTOM);

if (cr != CR_SUCCESS)
{
    goto Exit;
}

// Query a property on the device.  For example, the device description.
PropertySize = sizeof(DeviceDesc);
cr = CM_Get_DevNode_Property(Devinst,
                             &DEVPKEY_Device_DeviceDesc,
                             &PropertyType,
                             (PBYTE)DeviceDesc,
                             &PropertySize,
                             0);

if (cr != CR_SUCCESS)
{
    goto Exit;
}

if (PropertyType != DEVPROP_TYPE_STRING)
{
    goto Exit;
}

Index++;
}

Exit:

if (DeviceInterfaceList != NULL)
{
    HeapFree(GetProcessHeap(),
             0,
             DeviceInterfaceList);
}

return;
}

```

## Get a property from a specific device

This example takes a device instance path for a particular device and retrieves a property from it using [SetupDiGetDeviceProperty](#).

```

VOID
GetDevicePropertySpecificDeviceSetupapi(
    VOID
)
{
    HDEVINFO DeviceInfoSet = INVALID_HANDLE_VALUE;
    SP_DEVINFO_DATA DeviceInfoData;
    WCHAR DeviceDesc[2048];
    DEVPROPTYPE PropertyType;

    DeviceInfoSet = SetupDiCreateDeviceInfoList(NULL, NULL);

    if (DeviceInfoSet == INVALID_HANDLE_VALUE)
    {
        goto Exit;
    }

    ZeroMemory(&DeviceInfoData, sizeof(DeviceInfoData));
    DeviceInfoData.cbSize = sizeof(DeviceInfoData);

    if (!SetupDiOpenDeviceInfo(DeviceInfoSet,
                               MY_DEVICE,
                               NULL,
                               0,
                               &DeviceInfoData))
    {
        // The error can be retrieved with GetLastError();
        goto Exit;
    }

    // Query a property on the device. For example, the device description.
    if (!SetupDiGetDeviceProperty(DeviceInfoSet,
                                 &DeviceInfoData,
                                 &DEVPKEY_Device_DeviceDesc,
                                 &.PropertyType,
                                 (PBYTE)DeviceDesc,
                                 sizeof(DeviceDesc),
                                 NULL,
                                 0)) {
        // The error can be retrieved with GetLastError();
        goto Exit;
    }

    if (.PropertyType != DEVPROP_TYPE_STRING)
    {
        goto Exit;
    }

Exit:
    if (DeviceInfoSet != INVALID_HANDLE_VALUE)
    {
        SetupDiDestroyDeviceInfoList(DeviceInfoSet);
    }

    return;
}

```

This example takes a device instance path for a particular device and retrieves a property from it using [CM\\_Get\\_DevNode\\_Property](#).

```

void
GetDevicePropertySpecificDeviceCfgmgr32(
    VOID
)
{
    CONFIGRET cr = CR_SUCCESS;
    DEVINST Devinst;
    WCHAR DeviceDesc[2048];
    DEVPROPTYPE PropertyType;
    ULONG PropertySize;

    // If MY_DEVICE could be a non-present device, CM_LOCATE_DEVNODE_PHANTOM
    // should be used in place of CM_LOCATE_DEVNODE_NORMAL.
    cr = CM_Locate_DevNode(&Devinst,
        MY_DEVICE,
        CM_LOCATE_DEVNODE_NORMAL);

    if (cr != CR_SUCCESS)
    {
        goto Exit;
    }

    // Query a property on the device.  For example, the device description.
    PropertySize = sizeof(DeviceDesc);
    cr = CM_Get_DevNode_Property(Devinst,
        &DEVPKEY_Device_DeviceDesc,
        &.PropertyType,
        (PBYTE)DeviceDesc,
        &PropertySize,
        0);

    if (cr != CR_SUCCESS)
    {
        goto Exit;
    }

    if (.PropertyType != DEVPROP_TYPE_STRING)
    {
        goto Exit;
    }

Exit:
    return;
}

```

## Disable device

This example shows how to disable a device using CfgMgr32. To do this with SetupApi, you would use [SetupDiCallClassInstaller](#) with *InstallFunction* of DIF\_PROPERTYCHANGE, specifying [DICS\\_DISABLE](#).

**Note** By default, calling [SetupDiCallClassInstaller](#) results in the device staying disabled across reboots. To disable the device across reboots when calling [CM\\_Disable\\_DevNode](#), you must specify the [CM\\_DISABLE\\_PERSIST](#) flag.

```
cr = CM_Locate_DevNode(&devinst,
                      (DEVINSTID_W)DeviceInstanceId,
                      CM_LOCATE_DEVNODE_NORMAL);

if (cr != CR_SUCCESS) {
    goto Exit;
}

cr = CM_Disable_DevNode(devinst, 0);

if (cr != CR_SUCCESS) {
    goto Exit;
}
```

## Enable device

This example shows how to enable a device using CfgMgr32. To do this with SetupApi, you would use [SetupDiCallClassInstaller](#) with *InstallFunction* of DIF\_PROPERTYCHANGE, specifying DICS\_ENABLE.

```
cr = CM_Locate_DevNode(&devinst,
                      (DEVINSTID_W)DeviceInstanceId,
                      CM_LOCATE_DEVNODE_NORMAL);

if (cr != CR_SUCCESS) {
    goto Exit;
}

cr = CM_Enable_DevNode(devinst, 0);

if (cr != CR_SUCCESS) {
    goto Exit;
}
```

## Restart device

This example shows how to restart a device using CfgMgr32. To do this with SetupApi, you would use [SetupDiCallClassInstaller](#) with *InstallFunction* of DIF\_PROPERTYCHANGE, specifying DICS\_PROPCHANGE.

```
cr = CM_Locate_DevNode(&devinst,
                      (DEVINSTID_W)DeviceInstanceId,
                      CM_LOCATE_DEVNODE_NORMAL);

if (cr != CR_SUCCESS) {
    goto Exit;
}

cr = CM_Query_And_Remove_SubTree(devinst,
                                  NULL,
                                  NULL,
                                  0,
                                  CM_REMOVE_NO_RESTART);

if (cr != CR_SUCCESS) {
    goto Exit;
}

cr = CM_Setup_DevNode(devinst,
                      CM_SETUP_DEVNODE_READY);

if (cr != CR_SUCCESS) {
    goto Exit;
}
```

# Writing Class Installers and Co-Installers

12/1/2020 • 2 minutes to read • [Edit Online](#)

**Note** Features described in this section are not supported in universal or mobile driver packages. See [Using a Universal INF File](#).

This section contains the guidelines that you should follow when you write a *co-installer*.

[Displaying a user interface](#)

[Saving device installation state](#)

[Loading executable or DLL files](#)

[Starting other processes or services](#)

For more information about how to write a co-installer, see [Writing a Co-installer](#).

## Displaying a user interface

Device installation mostly runs in a system (noninteractive) service. Therefore, a user cannot see or respond to any user interface that appears in this context. Any dialog box that is provided in *co-installer* during the processing of a [device installation function \(DIF\) code](#) causes the device installation to stop responding.

In most cases, co-installers should not interact with the user except during the processing of a [finish-install action](#). Finish-install actions run in an interactive context.

**Note** Co-installers should not fail a DIF code with `ERROR_REQUIRES_INTERACTIVE_WINDOWSTATION` because that causes the device installation to fail. If the device installation requires user interaction, co-installers should support finish-install actions.

## Saving device installation state

Do not save device installation state within the *co-installer* dynamic-link library (DLL). Because Windows generally unloads the DLL after a DIF code is handled by the installer, any state information that is saved within the DLL would not persist.

To safely preserve device installer state, class installers or co-installers should save the state information as properties within the device's *driver key* in the registry. To do this, follow these steps:

1. To retrieve a registry handle to the driver key for a *device instance*, use [SetupDiOpenDevRegKey](#) with the *KeyType* parameter set to `DIREG_DRV`.
2. Use [SetupDiGetDevicePropertyKeys](#) (to retrieve all the property keys for a device instance) or [SetupDiGetDeviceProperty](#) (to retrieve a specified device instance property key).
3. Use [SetupDiSetDeviceProperty](#) to save the device instance property key.

## Loading executable or DLL files

If your *co-installer* attempts to load an unsigned executable file or DLL on a Windows 64-bit platform, the operating systems prevents it from being loaded in this secure environment.

To safely load an executable file or DLL by a class installer or co-installer, we highly recommended that the executable file or DLL is included in your digitally signed [driver package](#). For more information about how to sign

driver packages, see [Driver Signing](#).

**Note** Class installers and co-installers must not load DLL modules by explicit function calls, such as `LoadLibrary`, or by creating link dependencies.

## Starting other processes or services

During device installation, Windows cannot track additional processes and is unable to determine what they are doing or when they are finished. For example, Windows could start or stop the device or initiate a system restart while the process is performing a critical action.

In most cases, *co-installers* should not start other processes or services. However, installers can start other processes safely by calling [CreateProcess](#) from a function or dialog that is displayed through a [finish-install action](#). The installer must not let the user continue in the dialog or procedure until the created process has exited.

# Writing INF Files

5/8/2019 • 2 minutes to read • [Edit Online](#)

When you write an [INF file](#) for your [driver package](#), you should follow these guidelines:

- An INF file must use valid structure and syntax to pass driver package validation checks at the beginning of the installation process.

Use the [INFVerif](#) tool to validate the structure and syntax of INF files.

- An INF file must contain valid INF [SourceDisksFiles](#) and [SourceDisksNames](#) sections. Starting with Windows Vista, the operating system does not copy the driver package into the [driver store](#) unless these sections are present and filled in correctly.
- It is sometimes necessary to copy INF files during device installation so that Windows 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 Windows can find these INF files without prompting the user every time that it installs one of the device's functions.

Starting with Windows XP, if you want to stage other INF files during an installation that is driven by an INF file, use the [INF CopyINF directive](#).

**Note** Do not use the [INF CopyFiles directive](#) to copy INF files.

- The components of a driver package must never directly copy or delete INF files directly into a system's `%SystemRoot%\\Inf` directory. This results in the driver's digital signature to be invalidated, and this causes the driver not to load successfully.

# Installing a Boot-Start Driver

11/2/2020 • 4 minutes to read • [Edit Online](#)

A *boot-start driver* is a driver for a device that must be installed to start the Microsoft Windows operating system. Most boot-start drivers are included "in-the-box" with Windows, and Windows automatically installs these boot-start drivers during the text-mode setup phase of Windows installation. If a boot-start driver for a device is not included "in-the-box" with Windows, a user can install an additional vendor-supplied boot-start driver for the device during text-mode setup.

To install a device that is required to start Windows, but whose driver is not included with the operating system, a user must do the following:

1. Install the device hardware and turn on the computer.
2. Begin your Windows installation (run the Windows setup program). During the text-mode phase of the installation (at the beginning of the installation), Windows displays a message that indicates that you can press a specific  $F_n$  key to install a boot-start driver.
3. When Windows displays this message, press the specified  $F_n$  key to install the boot-start driver and then insert a [boot-start driver distribution disk](#).

**Note** This procedure demonstrates how you can install a driver that is not included "in-the-box" with Windows. Do not use this procedure to replace or update a driver that is included with Windows. Instead, wait until Windows starts and use Device Manager to perform an "update driver" operation on the device.

When Windows fails to start, certain error messages that are displayed can indicate that a boot-start driver is missing. The following table describes several error messages and their possible causes.

ERROR MESSAGE	POSSIBLE CAUSE
Inaccessible boot device	The boot disk is a third-party mass-storage device that requires a driver that is not included with Windows.
Setup could not determine your machine type	A new HAL driver is required. This error does not occur on most machines, but it might occur on a high-end server.
Setup could not find any hard drives in your computer	The required boot device drivers for the hard drives are not loaded.

## Boot-Start Driver Distribution Disk

A *boot-start driver distribution disk* is a medium, such as a floppy disk or USB flash drive, that contains a *TxtSetup.oem* file and the related driver files. The *TxtSetup.oem* file is a text file that contains a list of hardware components, a list of files on the distribution disk that will be copied to the system, and a list of registry keys and values that will be created. A sample *TxtSetup.oem* file is provided with the Windows Driver Kit (WDK), under the `\src` directory of the WDK. For details about the contents of a *TxtSetup.oem* file, see [TxtSetup.oem File Format](#).

The following requirements and recommendations apply to platform-specific and cross-platform distributions disks:

- Platform-specific distribution disks (Windows Server 2003 and earlier)

Windows requires a platform-specific distribution disk for each platform that a driver supports. A platform-specific distribution disk contains one *TxtSetup.oem* file and the related driver files. The *TxtSetup.oem* file must be located in the root directory of the distribution disk.

- Cross-platform and platform-specific distribution disks (Windows Server 2003 Service Pack 1 (SP1) and later versions)

Windows supports cross-platform distribution disks that contain two or more platform-specific *TxtSetup.oem* files and the related driver files.

To distinguish between platforms on a cross-platform distribution disk, use the platform directories that are listed in the following table.

PLATFORM	PLATFORM DIRECTORY	DEFAULT DIRECTORY
x86-based	A:\i386	A:\\
Itanium-based	A:\ia64	A:\\
x64-based	A:\amd64	A:\\

On a cross-platform distribution disk, Windows uses the platform-specific *TxtSetup.oem* file that is located in the platform directory that corresponds to the platform on which Windows is running. If a corresponding platform directory that contains a platform-specific *TxtSetup.oem* file does not exist, Windows uses the *TxtSetup.oem* file in the default directory, if one is present.

Windows also supports platform-specific distribution disks. A platform-specific distribution disk contains one platform-specific *TxtSetup.oem* file and the related driver files. The *TxtSetup.oem* file must be located either in its corresponding platform directory, as is done for cross-platform distribution disks, or in the default directory of the distribution disk.

The driver files for a given platform on a cross-platform distribution disk or on a platform-specific distribution disk must be located relative to the directory that contains the platform-specific *TxtSetup.oem* file.

**Tip** Although not required, we recommend that a *TxtSetup.oem* file always be placed in a corresponding platform directory. Using platform directories eliminates the possibility that Windows might attempt to use a *TxtSetup.oem* file that is incompatible with the platform on which Windows is running. For example, if a user attempts an unattended installation on a platform with a distribution disk that does not contain a corresponding platform directory, Windows cannot determine whether the *TxtSetup.oem* file in the default directory is compatible with the platform. If a driver fails to load because the driver is incompatible with the platform, Windows displays an error message and terminates the unattended installation.

# Installing a Filter Driver

12/1/2020 • 2 minutes to read • [Edit Online](#)

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 about 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 *DD\Install.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_addrreg

[upperfilter_copyfiles]
upperfilt.sys,,,0x00004000 ; COPYFLG_IN_USE_RENAME

[upperfilter_addrreg]
; 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.

# Installing a Null Driver

12/1/2020 • 2 minutes to read • [Edit Online](#)

You might install a "null driver" (that is, nonexistent 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. Additionally, the system installs null drivers for devices that do not have a [function driver](#), if they are capable of executing in *raw mode*.

To specify a null driver in an INF file, use entries like the following:

```
:  
[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 device node (*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.

# Installing a New Bus Driver

12/1/2020 • 2 minutes to read • [Edit Online](#)

New vendor bus drivers should comply with the following guidelines for reporting the bus type and device IDs, and, possibly, for creating a new [device setup class](#) for the child devices on the bus. These guidelines apply to a new vendor bus if the configuration or operation of the bus and its child devices differ significantly from other buses. In those cases, new vendor bus drivers should do the following to ensure that the bus and its child devices are not unintentionally and inappropriately grouped with other buses and child devices:

1. Use a unique GUID to identify the bus type. A bus driver reports the bus type of a child device (represented as a physical device object (*PDO*) in response to an [IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#) request for the child device. In response to such a request, the bus driver returns a pointer to a [PNP\\_BUS\\_INFORMATION](#) structure that returns the GUID in the [PNP\\_BUS\\_INFORMATION.BusTypeGuid](#) member. In addition, the bus driver should set [PNP\\_BUS\\_INFORMATION.LegacyBusType](#) to [PNPBus](#), and [PNP\\_BUS\\_INFORMATION.BusNumber](#) to an appropriate custom value.
2. [Use custom hardware IDs](#) to uniquely identify the bus *enumerator* and the child devices of the bus.
3. If the child devices of the bus do not belong to an existing [device setup class](#), [install a new device setup class for the child devices of the bus](#).

# Using Custom Hardware IDs and Compatible IDs

12/1/2020 • 2 minutes to read • [Edit Online](#)

As described in [Device Identification Strings](#), the following is the generic format that a new bus driver should use for Plug and Play (PnP) hardware IDs and compatible IDs.

```
enumerator\enumerator-specific-device-ID
```

Where:

- *Enumerator* identifies the *enumerator* (bus driver) that detects and reports child devices on a bus to the PnP manager.
- *enumerator-specific-device-ID* is a device identifier specific to the bus driver.

If the configuration or operation of a bus differs significantly from other buses, the bus driver for the bus should use a unique enumerator name to ensure that the child devices of the bus are not unintentionally and inappropriately grouped with child devices that are enumerated by the bus drivers for these other buses. The bus driver should use the following format to report device identification strings to the PnP manager:

```
bus-type-guid\vendor-specific-id
```

Where:

- *bus-type-guid* is a unique GUID that identifies the bus and should be the same GUID that is used to identify the bus. As described in [Installing a Bus Driver](#), the bus driver identifies the bus type for a device in response to an [IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#) request for the device.
- *vendor-specific-id* is vendor-defined format that typically identifies the vendor, the device, a subsystem, a revision number, and possibly other device information. For example, the format might take the form of *Vendor&Device&Subsystem&Revision*, where the ampersand character ("&") delimits the subfields and the format of each subfield is vendor-specific. For examples of actual device identification strings, see [Device Identification Strings](#).

The PnP manager sends [IRP\\_MN\\_QUERY\\_ID](#) requests to a bus driver to obtain device identification strings for a device. The device identification strings include a device ID, a device instance ID, a list of hardware IDs, and a list of compatible IDs. The following fictitious examples include a device ID, a list of hardware IDs and a list of compatible IDs. In these examples, the enumerator is specified by the *bus-type-guid* subfield, which is the GUID "{xxxxxxxx-yyyy-zzzz-xxxx-yyyyyyyyyyyy}". The format of the *vendor-specific-id* field is *Vendor&Device&Subsystem&Revision*, where the *Vendor* subfield is "ven\_1", the *Device* subfield is "dev\_2", the *Subsystem* subfield is "subsys\_3", and the *Revision* subfield is "rev\_4".

A device ID is the hardware ID that is the most specific description of a device. In the following example, the device ID specifies the vendor, the device, the subsystem, and the revision.

```
{xxxxxxxx-yyyy-zzzz-xxxx-yyyyyyyyyyyy}\ven_1&dev_2&subsys_3&rev_4
```

A hardware ID list specifies IDs in order, from the most specific to the least specific. In the following list, a device identification string is reported as hardware ID if it specifies at least the vendor, the device, and the subsystem. The hardware ID that includes the most information is listed first.

```
{xxxxxxxx-yyy-zzzz-xxxx-yyyyyyyyyyyy}\ven_1&dev_2&subsys_3&rev_4  
{xxxxxxxx-yyy-zzzz-xxxx-yyyyyyyyyyyy}\ven_1&dev_2&subsys_3
```

In the following list, a device identification string is reported as compatible ID if it specifies at least the vendor and device (positions 1 and 2), but does not specify the subsystem (position 3). The compatible ID that includes the most information is listed first.

```
{xxxxxxxx-yyy-zzzz-xxxx-yyyyyyyyyyyy}\ven_1&dev_2&rev_4  
{xxxxxxxx-yyy-zzzz-xxxx-yyyyyyyyyyyy}\ven_1&dev_2
```

If a driver installs using a hardware ID, it implies full functionality for matching devices. If a driver installs using a compatible ID, it implies at least basic functionality for matching devices. A driver might use a compatible ID so that a generic driver can work on a large number of devices. For example, many of the Windows system-supplied drivers match compatible IDs. A driver that matches a hardware ID typically targets a small set of devices but provides full functionality.

# Installing a New Device Setup Class for a Bus

11/2/2020 • 2 minutes to read • [Edit Online](#)

If a new bus supports devices whose capabilities are significantly different from the capabilities provided by devices that belong to existing [device setup classes](#), you should install a new device setup class for the bus. For more information that helps you determine whether to install a new device setup class, see [Creating a New Device Setup Class](#).

To install a new device setup class for the bus, set related directives in the [INF Version section](#), include an [INF ClassInstall32 section](#), and include additional sections, as needed, that are referenced by the INF Class32 section.

The following annotated example illustrates the basic INF file entries you need to include to install a [device setup class](#). For information about the possible configuration settings of a device setup class, see [INF ClassInstall32 section](#).

```
[Version]
signature="$CHICAGO$"
; Specify a unique class name that identifies the manufacturer and the bus type
Class=%AbcSuperBus%

; Specify a unique GUID that identifies the device setup class
ClassGUID={17ed6609-9bc8-44ca-8548-abb79b13781b}

; Identify the provider of the INF file
Provider=%AbcCorp%

; Specify the version of the device driver
DriverVer=01/01/2007,1.0.0.0

[ClassInstall32]
; Reference an AddReg directive that specifies class properties
Addreg=AbcSuperBusClassReg

[AbcSuperBusClassReg]
; Specify the properties of the device setup class
HKR,,,,%AbcSuperBus%
HKR,,Icon,,,-19
HKR,,NoInstallClass,,1
. . .
. . .

[Strings]
AbcCorp="Abc Corporation"
AbcSuperBus="Abc Corporation Super Bus Controller"
```

# Installing Private Builds of Inbox Drivers

12/5/2018 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, the operating system provides a mechanism that can install updated versions of Microsoft-signed drivers. Otherwise, a Microsoft-signed driver has a higher selection value, or *rank*, than a driver that is signed by a third party. This makes it difficult for driver developers to install private builds of an inbox driver.

This section describes how to build and install a private build of a driver that is included in the default installation of Windows. Such a driver is referred to as an "*inbox*" driver.

**Note** To understand this material, you must be familiar with digitally signing a driver and how Windows ranks drivers based on this signature and other criteria. For more information about digital signatures for drivers, see [Driver Signing](#).

The following topics describe how you can override the default rank number for your driver so that you can install a private build of an inbox driver:

[Overview of Installing Private Builds of Inbox Drivers](#)

[Creating a Private Build of an Inbox Driver](#)

[Configuring Windows to Rank Driver Signatures Equally](#)

[Installing the Updated Version of the Driver Package](#)

**Important** Windows Update depends on Microsoft-signed drivers having priority over drivers that have third-party signatures. Configuring a system to override this priority can interfere with the ability of Windows Update to provide the correct drivers to consumers. This can result in an installation failure for drivers that are delivered by Windows Update.

# Overview of Installing Private Builds of Inbox Drivers

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, when a Plug and Play (PnP) device is installed on a computer system, Windows selects a driver based on several factors, such as the [hardware ID](#) or [compatible ID](#), date, and version. Windows analyzes these factors to assign a rank that indicates how well the driver matches the device. The lower the rank, the better a match the driver is for the device.

Also, starting with Windows Vista, if a driver has a signature from a Windows signing authority (Microsoft signature), Windows ranks it better than another driver for the same device that was signed with:

- A third-party release signature. This type of signature is generated by using a [Software Publisher Certificate](#) that is obtained from a third-party certification authority (CA) authorized by Microsoft to issue such certificates.
- A Microsoft signature for a Windows version that is earlier than the [LowerLogoVersion](#) value of the driver's [device setup class](#).

The Microsoft signature types include the following:

- Premium WHQL signatures and standard WHQL signatures
- Signatures for inbox drivers
- Windows Sustained Engineering (Windows SE) signatures
- A WHQL signature for a Windows version that is the same or later than the Windows version that is specified by the [LowerLogoVersion](#) value that is set for the device setup class of a driver

**Note** Even if a driver that has a third-party signature is a better match for the device, Windows selects the driver that has a Microsoft signature. Using a publisher identity certificate [PIC] for the third-party signature does not change this behavior.

Starting with Windows Vista, the [AllSignersEqual Group Policy](#) controls how Windows ranks Microsoft-signed drivers and third party-signed drivers. When **AllSignersEqual** is enabled, Windows treats all Microsoft signatures and third-party signatures as equal with respect to rank when selecting the driver that is the best match for a device.

**Note** In Windows Vista and Windows Server 2008, the [AllSignersEqual](#) Group Policy is disabled by default. Starting with Windows 7, this Group Policy is enabled by default.

To install a private build of an inbox driver, you must do the following:

- Build a private version of the inbox driver. You must ensure that the private build outranks the Microsoft-signed version when signatures are treated equally. The private build must also be digitally signed by using tools that are provided with the WDK.

For more information, see [Creating a Private Build of an Inbox Driver](#).

- Enable the [AllSignersEqual Group Policy](#) on the target system so that the operating system views all Microsoft signature types and third-party signatures as equal in rank when it selects the driver that is the best match for a device.

For more information, see [Configuring Windows to Rank Driver Signatures Equally](#).

For more information about how Windows ranks drivers, see [How Windows Selects Drivers](#).



# Creating a Private Build of an Inbox Driver

11/2/2020 • 2 minutes to read • [Edit Online](#)

When you build a private version of the inbox driver, and [Windows is configured to rank driver signatures equally](#), you must ensure that the private build outranks the Microsoft-signed version. The simplest way to ensure this is to update the value of the **INF DriverVer directive** in the [driver package's INF file](#). The new value must specify a date and version that is later than the value of the **DriverVer** directive in the package's INF file that is installed on the target system.

You can automate the process for building a private version of an inbox driver that outranks the Microsoft-signed version by following these steps:

1. Modify the makefile to generate a new INF file for the [driver package](#). For example, add the following line to the *Makefile*:

```
$(O)\sample.inf
```

2. Add directives to the *Makefile* that will generate a new INF file and execute the [Stampinf](#) tool to time stamp the INF file. For example, the following code example shows how you can create and time stamp an INF file that is named *Sample.inf*.

```
$(O)\ sample.inf: $(_INX)\ sample.inx $(_LNG)\ sample.txt  
    $(C_PREPROCESSOR_NAME) $(PREFLAGS) $(_LNG)\$(@B).txt > $(O)\$(@B).txt1  
    copy /b $(_INX)\$(@B).inx+$(_O)\$(@B).txt1 $@  
    @del $(O)\$(@B).txt1  
    stampinf -f sample.inf -d * -v * -c MyCatalogFile.cat  
    $(TSBINPLACE_CMD)
```

The following [Stampinf](#) command-line parameters are used in this example:

- The **-d \*** parameter uses the current date as part of the **DriverVer** directive in the INF file.
- The **-v \\*** parameter uses the current time for the version number. If the **STAMPINF\_VERSION** environment variable is set, Stampinf uses the version number value that is specified by this environment variable.
- The **-c** parameter specifies the name of the [catalog file](#) for the [driver package](#). This value is written to the **CatalogFile** directive of the **INF Version section** of the generated IF file.

**Note** If you set the environment variable **PRIVATE\_DRIVER\_PACKAGE**, Stampinf uses the current date and version for the **INF DriverVer** directive. By setting this environment variable, you do not have to use the **-d** or **-v** parameters in your *Makefile*.

Once the driver is built, you must sign the [driver package](#) and must use the same [catalog file](#) that was specified in the **-c** parameter of [Stampinf](#) within your *Makefile*. To sign the driver package, follow the steps that are outlined in [Signing Drivers During Development and Test](#).

# Configuring Windows to Rank Driver Signatures Equally

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [AllSignersEqual Group Policy](#) controls [how Windows ranks drivers](#) that are signed by Microsoft versus drivers that are signed by third-party vendors. When the AllSignersEqual Group Policy is enabled, Windows views all Microsoft signature types and third-party signatures as equal in rank when it selects the driver that is the best match for a device.

**Note** In Windows Vista and Windows Server 2008, the **AllSignersEqual** Group Policy is disabled by default. Starting with Windows 7, this Group Policy is enabled by default.

After you enable the AllSignersEqual Group Policy, this configuration change applies to all subsequent driver installations on the target system until you disable the AllSignersEqual Group Policy.

For more information about how to configure the **AllSignersEqual** Group Policy to rank driver signatures equally, see [AllSignersEqual Group Policy \(Windows Vista and Later\)](#).

# Installing the Updated Version of the Driver Package

11/2/2020 • 2 minutes to read • [Edit Online](#)

After you [configure Windows to rank driver signatures equally](#), you can install the private build of the inbox driver on the target system. To install the private build, complete the following steps:

1. Add the [driver package](#) to the driver store by using the [PnPUTil](#) utility that is provided in Windows Vista and later versions of Windows. For example:

```
pnputil.exe -a sample.inf
```

2. Use the DevCon Remove command to remove the device or device class that is installed by the updated driver package. The device or device class is specified through all or part of a [hardware ID](#), [compatible ID](#), or device instance ID of a device. For example:

```
devcon remove "PCI\VEN_8086&DEV_7110"
```

The new driver is automatically loaded when the device is reinstalled after the system is restarted. To have DevCon automatically restart the system, add the conditional reboot parameter (*/r*) to the remove command.

**Note** The [DevCon](#) tool is provided in the WDK. For more information about its commands, see [DevCon Commands](#).

An alternative to the DevCon Remove command is to update the [driver package](#) by using one of the following:

- Device Manager to perform an "update driver" operation on the device.

From within Device Manager's window, right-click the device's name or icon and choose **Properties**. In the **Properties** window, click the Driver tab and then click the **Update Driver** button.

- The DevCon Update command. For more information about this command, see [DevCon Commands](#).

# Win32 Services Interacting with Devices

11/2/2020 • 3 minutes to read • [Edit Online](#)

## Motivation:

An ideal Win32 service installed via [INF AddService](#) that interacts with devices behaves similar to how a *driver* interacts with *devices*. A driver is loaded and unloaded depending on the presence of a device, and a Win32 service that interacts with devices should follow this same pattern of **starting** and **stopping** depending on the presence of a device.

Services should start only when a device interface is present to interact and stopped when the associated device is no longer present. This design pattern ensures a robust service that minimizes undesired and undefined behavior. We will walk through how a service should be designed to follow this pattern.

## Service Install

To install the service, use the [INF AddService](#) directive. This will allow you to create and start the service.

Add the flag that makes the service demand-start. This can be accomplished by setting `StartType=0x3` which makes the service trigger started.

The final step in this section is to use the [AddTrigger](#) directive to make the service start when a device interface arrives (see [AddService](#) for more details regarding [AddTrigger](#)). Below is an example of how [AddTrigger](#) should be used:

```
[UserSvc_AddTrigger]
TriggerType = 1                                ; SERVICE_TRIGGER_TYPE_DEVICE_INTERFACE_ARRIVAL
Action      = 1                                ; SERVICE_TRIGGER_ACTION_SERVICE_START
SubType     = %GUID_DEVINTERFACE_OSRFX2%        ; Interface class GUID
DataItem    = 2, "USB\VID_0547&PID_1002"         ; SERVICE_TRIGGER_DATA_TYPE_STRING

[UserSvc_Install]
ServiceType = 0x10 ; SERVICE_WIN32_OWN_PROCESS
StartType   = 3   ; SERVICE_DEMAND_START
ErrorControl = 0   ; SERVICE_ERROR_IGNORE
ServiceBinary = %13%\oemsvc.exe
AddTrigger   = UserSvc_AddTrigger
```

Note that the `HardwareId` specified in `DataItem` is optional and generally only needed when using a generic class interface to scope the trigger to a more specific device.

## Service Runtime

From a runtime perspective, the first step for your service should be to register for device interface notifications. Prescriptive guidance on how to accomplish this can be found on this page: [Registering for Notification of Device Interface Arrival and Device Removal](#).

In particular, you should use `CM_Register_Notification` with the `CM_NOTIFY_FILTER_TYPE_DEVICEINTERFACE` flag to accomplish the appropriate registration of device interface notifications.

#### **NOTE**

When a service has started, you cannot rely on the fact that you will receive device interface notifications as the arrival notification may have already passed. Instead, you must get a list of device interfaces to check if there are already interfaces present.

Once you have registered for notifications, you can find your desired device interface in these two steps:

1. Discover the interface from the notification callback
2. Query the list of existing device interfaces and find the desired interface using

**CM\_Get\_Device\_Interface\_List**

#### **NOTE**

Note, there is a chance that the interface arrives between registering for notifications and finding the desired device interface. In that case, the interface will be listed in both the notification callback and the list of interfaces.

Once you have found the desired device interface, open a handle to the interface via [CreateFile](#).

The next step is to register for secondary per-interface notifications to operate and manage the device. This can be done using **CM\_Register\_Notification** with the **CM\_NOTIFY\_FILTER\_TYPE\_DEVICEHANDLE** flag. This will ensure that when a device is going away, the handle can be released accordingly.

Device interface arrivals and removals should be tracked so that the removal of the last device interface means the service can be stopped. Once the last interface has been removed, stop your service (detailed information can be found on this [page](#)). This can be accomplished by following these steps:

1. Post **SERVICE\_STOP\_PENDING** state to SCM to indicate the service is going down
2. Uninitialize/clean-up everything the service was using
3. Post **SERVICE\_STOP** state to SCM to complete the stop operation

If the service is being stopped, make sure to check for and go through all existing open handles to device interfaces (there may be none) and clean them up.

The device interface can come back either during device installation, device enable/disable, device re-enumeration, system reboot, or during other scenarios not listed. When the device interface comes back, the service will be triggered based on its trigger start registration.

This flow will ensure that the service starts on the arrival of a device interface and stops when the last device interface is no longer present.

## Code Sample & Related Links

There is a sample on GitHub that walks through how a service can leverage this flow of events. The sample can be found here: [Win32 Service Sample](#).

Additionally, you can find useful documentation regarding **AddTrigger** on the [AddService](#) page.

# Troubleshooting Device and Driver Installations

11/2/2020 • 2 minutes to read • [Edit Online](#)

You can use the following guidelines to either verify that your device is installed correctly or diagnose problems with your device installation:

- Follow the steps that are described in [Using Device Manager](#) to view system information about the device.
- Follow the steps that are described in [SetupAPI Logging \(Windows Vista and Later\)](#) or [SetupAPI Logging \(Windows Server 2003, Windows XP, and Windows 2000\)](#) to identify installation errors.
- On Windows Vista and later versions of Windows, follow the steps that are described in [Debugging Device Installations \(Windows Vista and Later\)](#) to debug [co-installers](#) during the core stages of device installation.
- On Windows Vista and later versions of Windows, follow the steps that are described in [Troubleshooting Install and Load Problems with Test-signed Drivers](#) to diagnose problems related to the installation and loading of test-signed drivers.
- Run test programs to exercise the device. This includes the testing and debugging tools that are supplied with the Windows Driver Kit (WDK).

Additionally, in Windows Server 2003, Windows XP, and Windows 2000, a [co-installer](#) can provide a troubleshooter that helps users diagnose problems with your device. See [DIF\\_TROUBLESHOOTER](#) for more information.

# Using Device Manager

11/2/2020 • 2 minutes to read • [Edit Online](#)

To start Device Manager, in File explorer, select and hold (or right-click) **This PC**, select **Manage**, and then select **Device Manager** from the System Tools that are listed in the resulting dialog.

Device Manager displays information about each device. This includes the device type, device status, manufacturer, device-specific properties, and information about the driver for the device.

If your device is required to start the computer, a problem with your device installation can prevent the computer from starting. In these cases, you have to use the kernel debugger to troubleshoot your device installation. For more info, see [Getting Started with WinDbg \(Kernel-Mode\)](#).

If your device is not required to start the computer, Device Manager places a yellow exclamation point next to that device's name in the Device Manager dialog. Device Manager also provides an error message describing the problem. For more information about the error messages, see [Device Manager Error Messages](#).

Device manager can show hidden devices. This is helpful when you are testing the installation of a new PnP device. For more information, see [Viewing Hidden Devices](#).

Device Manager provides detailed information in the Properties dialog for each device. Select and hold (or right-click) the name of the device, and then select **Properties**. The **General**, **Driver**, **Details**, and **Events** tabs contain information that can be useful when you debug errors. For more information, see [Device Manager Details Tab](#).

# Device Manager Error Messages

3/4/2020 • 2 minutes to read • [Edit Online](#)

When Device Manager marks a device with a yellow exclamation point, it also provides an error message.

The following table lists the errors reported by Device Manager. These error codes are defined in Cfg.h.

For related info, see [Retrieving the Status and Problem Code for a Device Instance](#).

ERROR CODE	PROBLEM NAME
Code 1	<a href="#">CM_PROB_NOT_CONFIGURED</a>
Code 3	<a href="#">CM_PROB_OUT_OF_MEMORY</a>
Code 9	<a href="#">CM_PROB_INVALID_DATA</a>
Code 10	<a href="#">CM_PROB_FAILED_START</a>
Code 12	<a href="#">CM_PROB_NORMAL_CONFLICT</a>
Code 14	<a href="#">CM_PROB_NEED_RESTART</a>
Code 16	<a href="#">CM_PROB_PARTIAL_LOG_CONF</a>
Code 18	<a href="#">CM_PROB_REINSTALL</a>
Code 19	<a href="#">CM_PROB_REGISTRY</a>
Code 21	<a href="#">CM_PROB_WILL_BE_REMOVED</a>
Code 22	<a href="#">CM_PROB_DISABLED</a>
Code 24	<a href="#">CM_PROB_DEVICE_NOT THERE</a>
Code 28	<a href="#">CM_PROB_FAILED_INSTALL</a>
Code 29	<a href="#">CM_PROB_HARDWARE_DISABLED</a>
Code 31	<a href="#">CM_PROB_FAILED_ADD</a>

ERROR CODE	PROBLEM NAME
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	CM_PROB_REGISTRY_TOO_LARGE
Code 50	CM_PROB_SETPROPERTIES_FAILED

ERROR CODE	PROBLEM NAME
Code 51	<a href="#">CM_PROB_WAITING_ON_DEPENDENCY</a>
Code 52	<a href="#">CM_PROB_UNSIGNED_DRIVER</a>
Code 53	<a href="#">CM_PROB_USED_BY_DEBUGGER</a>
Code 54	<a href="#">CM_PROB_DEVICE_RESET</a>
Code 55	<a href="#">CM_PROB_CONSOLE_LOCKED</a>
Code 56	<a href="#">CM_PROB_NEED_CLASS_CONFIG</a>
Code 57	<a href="#">CM_PROB_GUEST_ASSIGNMENT_FAILED</a>

# Code 1 - CM\_PROB\_NOT\_CONFIGURED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that there is a device on the system for which there is no **ConfigFlags** registry entry. This means no driver is installed. Typically this means an INF file could not be found.

## Error Code

1

### Display Message

"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

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

## For driver developers

This error means that PnP has not attempted to install the device. Retry installation.

If this problem status occurs in conjunction with a [Bug Check 0x7B: INACCESSIBLE\\_BOOT\\_DEVICE](#) and the device is on the path to the boot disk, the system is missing a driver for a boot critical device.

# Code 3 - CM\_PROB\_OUT\_OF\_MEMORY

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the system is probably running low on system memory.

## Error Code

3

### Display Message

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

### Recommended Resolution

If a device driver is corrupted, update the driver.

To update a device driver, follow these steps:

1. [Open Device Manager](#).
2. Right-click the icon that represents the device in the Device Manager window.
3. On the menu that appears, click **Update Driver** to start the Hardware Update wizard.

If a driver for a device is operating correctly, but the computer has insufficient memory for the device and other applications to operate simultaneously, you could close some applications to make sufficient memory to operate the device.

To check available memory and system resources, follow these steps:

1. Click the **Start** menu.
2. Point to the following sequence of submenus: **All Programs**, **Accessories**, and **System Tools**.
3. On the **System Tools** menu, click **System Information** to determine whether the amount of available physical memory is sufficient to operate the device.

You might have to install additional random access memory (RAM) to operate a device.

# Code 9 - CM\_PROB\_INVALID\_DATA

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that invalid device IDs have been detected.

## Error Code

9

### Display Message

"Windows cannot identify this hardware because it does not have a valid hardware identification number. (Code 9)"

"For assistance, contact the hardware manufacturer."

### Recommended Resolution

Contact the hardware vendor. The hardware or the driver is defective.

# Code 10 - CM\_PROB\_FAILED\_START

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device failed to start.

## Error Code

10

### Display Message

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

Select [Update Driver](#), which starts the Hardware Update 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.

See the [DEVPKEY\\_Device\\_ProblemStatus](#) property on the device for the failure code returned for the [start IRP](#).

For additional information, see [Retrieving the Status and Problem Code for a Device Instance](#).

## For driver developers

One of the drivers in the device stack failed the [start IRP](#). The [DEVPKEY\\_Device\\_ProblemStatus](#) property on the device should indicate the failure code.

If one of the drivers in the device's driver stack is a UMDF driver and the [DEVPKEY\\_Device\\_ProblemStatus](#) property on the device is STATUS\_DRIVER\_PROCESS\_TERMINATED, this information may be helpful for the owner of the driver to diagnose the problem:

- [Determining Why the Reflector Terminated the Host Process](#)
- [Troubleshooting UMDF 2.0 Driver Crashes](#)

# Code 12 - CM\_PROB\_NORMAL\_CONFLICT

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that 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

"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

[Use Device Manager](#) to determine the source of the conflict and to resolve the conflict. For more information about how to resolve device conflicts, see the Help information about how to use Device Manager.

This error message can also appear if the BIOS did not allocate sufficient resources to a device. For example, this message will be displayed if the BIOS does not allocate an interrupt to a USB controller because of an invalid multiprocessor specification (MPS) table.

# Code 14 - CM\_PROB\_NEED\_RESTART

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the system must be restarted.

## Error Code

14

### Display Message

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

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

### Recommended Resolution

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

# Code 16 - CM\_PROB\_PARTIAL\_LOG\_CONF

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is only partially configured.

## Error Code

16

### Display Message

"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

Manually configure the resources that the device requires.

To configure device resources, follow these steps:

1. [Open Device Manager](#).
2. Double-click the icon that represents the device in the Device Manager window.
3. On the device properties sheet that appears, click the **Resources** tab. The device resources are listed in the **resource settings** list on the **Resources** page.
4. If a resource in the **resource settings** list has a question mark next to it, select that resource to assign it to the device.
5. If a resource cannot be changed, click **Change Settings**. If **Change Settings** is unavailable, try to clear the **Use automatic settings** check box to make it available.

If the device is not a Plug and Play device, look in the device documentation for more information about how to configure resources for the device.

# Code 18 - CM\_PROB\_REINSTALL

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that drivers must be reinstalled.

## Error Code

18

### Display Message

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

### Recommended Resolution

To reinstall a device driver by using the Hardware Update wizard, follow these steps:

1. [Open Device Manager](#).
2. Right-click the icon that represents the device in the Device Manager window.
3. On the menu that appears, click **Update Driver** to start the Hardware Update wizard.

Alternatively, you can reinstall a device driver by following these steps:

1. Open Device Manager.
2. Right-click the icon that represents the device in the Device Manager window.
3. On the menu that appears, click **Uninstall** to uninstall the device driver.
4. Click **Action** on the Device Manager menu bar.
5. On the **Action** menu, click **Scan for hardware changes** to reinstall the device driver.

## For driver developers

This problem code is frequently transient.

# Code 19 - CM\_PROB\_REGISTRY

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that a registry problem was detected.

## Error Code

19

### Display Message

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

### Recommended Resolution

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

Try either uninstalling and reinstalling the driver or rolling back the system to the most recent successful configuration of the registry.

To uninstall and reinstall a device driver, follow these steps:

1. [Open Device Manager](#).
2. Right-click the icon that represents the device in the Device Manager window.
3. On the menu that appears, click **Uninstall** to uninstall the device driver.
4. Click **Action** on the Device Manager menu bar.
5. On the **Action** menu, click **Scan for hardware changes** to reinstall the device driver.

To roll a system back to the most recent successful configuration of the registry, restart the computer in Safe Mode and select the Last Known Good Configuration option.

# Code 21 - CM\_PROB\_WILL\_BE\_REMOVED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the system will remove the device.

## Error Code

21

### Display Message

"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

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

**Note** This problem code is transitory, and exists only during the attempts to query and then remove a device.

# Code 22 - CM\_PROB\_DISABLED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is disabled.

## Error Code

22

### Display Message

"This device is disabled. (Code 22)"

### Recommended Resolution

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

# Code 24 - CM\_PROB\_DEVICE\_NOT THERE

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device does not seem to be present.

## Error Code

24

### Display Message

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

### Recommended Resolution

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.

**Note** For Windows XP and later versions of Windows, **DriverEntry** problems have separate error codes.

# Code 28 - CM\_PROB\_FAILED\_INSTALL

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device's drivers are not installed.

## Error Code

28

### Display Message

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

### Recommended Resolution

Please visit the website of the company that manufactures the device and look for the most recent drivers for this device.

## For driver developers

The [DEVPKEY\\_Device\\_ProblemStatus](#) property on the device should indicate the failure code.

### 0xC0000490 - STATUS\_PNP\_NO\_COMPAT\_DRIVERS

PnP could not find a compatible driver for the device. This failure is often referred to as a DNF (driver not found) problem.

Examine the hardware IDs and compatible IDs of the device in question and compare to the hardware ID(s) that the INF specifies under the [Models sections](#). Also make sure that the [TargetOSVersion](#) part of the Models section name applies to the architecture and OS version you are running on.

### 0xC0000491 - STATUS\_PNP\_DRIVER\_PACKAGE\_NOT\_FOUND

This code indicates a missing driver package dependency.

Specifically, the INF that matched on the device uses [Include](#) entries in the [INF DDInstall section](#) to specify a Microsoft-supplied INF that is not present in this version of Windows.

### 0xC0000492 - STATUS\_PNP\_DRIVER\_CONFIGURATION\_NOT\_FOUND

This code also indicates a missing driver package dependency.

In this case, the INF that matched on the device uses [Needs](#) entries in the [INF DDInstall section](#) to specify a section that does not exist in any Microsoft-supplied INF referenced by an [Include](#) directive.

### 0xC0000494 - STATUS\_PNP\_FUNCTION\_DRIVER\_REQUIRED

This failure occurs when the INF does not specify an associated function driver service.

Verify that either:

1. The INF file for the device being installed contains an [AddService directive](#) that sets an associated service or function driver using the flag SPSVCINST\_ASSOCSERVICE (0x00000002).
2. The INF file specifies [Include](#) or [Needs](#) entries in a [INF DDInstall Section](#) that reference a system-supplied driver that in turn sets an associated service on the device.

## Upgrade to Windows 10

Before upgrade, the device has a driver and is working fine. After upgrade, it shows Code 28. This is frequently caused by the driver package being excluded from the migration.



# Code 29 - CM\_PROB\_HARDWARE\_DISABLED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is disabled.

## Error Code

29

### Display Message

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

### Recommended Resolution

Enable the device in the BIOS.

# Code 31 - CM\_PROB\_FAILED\_ADD

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that a driver's attempt to add a device failed.

## Error Code

31

### Display Message

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

### Recommended Resolution

Update the device driver.

# Code 32 - CM\_PROB\_DISABLED\_SERVICE

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the driver has been disabled.

## Error Code

32

### Display Message

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

### Recommended Resolution

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

# Code 33 - CM\_PROB\_TRANSLATION\_FAILED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that resource translation failed for the device.

## Error Code

33

### Display Message

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

### Recommended Resolution

Try using the BIOS setup utility, or update the BIOS.

Configure, repair, or replace hardware.

# Code 34 - CM\_PROB\_NO\_SOFTCONFIG

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device requires a forced configuration.

## Error Code

34

## Display Message

"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

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

# Code 35 - CM\_PROB BIOS\_TABLE

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the MPS table is bad and has to be updated.

## Error Code

35

### Display Message

"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

Obtain a new BIOS from the system vendor.

# Code 36 - CM\_PROB\_IRQ\_TRANSLATION\_FAILED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the IRQ translation failed for the device.

## Error Code

36

### Display Message

"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

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.)

# Code 37 - CM\_PROB\_FAILED\_DRIVER\_ENTRY

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the driver returned failure from its `DriverEntry` routine.

## Error Code

37

### Display Message

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

### Recommended Resolution

Reinstall or obtain a new driver.

**Note** If the `DriverEntry` routine returns `STATUS_INSUFFICIENT_RESOURCES`, Device Manager reports the [CM\\_PROB\\_OUT\\_OF\\_MEMORY](#) error code.

# Code 38 - CM\_PROB\_DRIVER\_FAILED\_PRIOR\_UNLOAD

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the driver could not be loaded because a previous instance is still loaded.

## Error Code

38

### Display Message

"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

A previous driver instance can still be loaded due to an incorrect 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 or more INF files.

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

# Code 39 - CM\_PROB\_DRIVER\_FAILED\_LOAD

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the driver could not be loaded.

## Error Code

39

### Display Message

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

### Recommended Resolution

Reinstall or obtain a new driver.

Reasons for this error include the following:

- A driver file 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.
- The driver does not comply with [kernel-mode code signing policy](#).

# Code 40 - CM\_PROB\_DRIVER\_SERVICE\_KEY\_INVALID

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that information in the registry's service key for the driver is invalid.

## Error Code

40

### Display Message

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

### Recommended Resolution

Reinstall the driver.

# Code 41 - CM\_PROB\_LEGACY\_SERVICE\_NO\_DEVICES

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that a driver was loaded but Windows cannot find the device.

## Error Code

41

### Display Message

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

### Recommended Resolution

Reinstall the device.

This is a legacy root service that did not create a device object.

# Code 42 - CM\_PROB\_DUPLICATE\_DEVICE

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that a duplicate device was detected.

## Error Code

42

### Display Message

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

### Recommended Resolution

This error is reported when one of the following occurs:

- A device with a serial number is discovered in a new location in the computer before the operating system notices that the device is missing from the old location. This typically happens when a device is moved, either very quickly or when the computer is in a standby or hibernate state, to a different location.

In this case, you can resolve the problem by restarting the computer.

- A bus driver incorrectly creates two identically named children on the bus. This is caused by multiple devices on the bus that report the same serial number. This can also be caused by a bus driver that incorrectly reports the same hardware identifiers for two or more devices.

In this case, you should contact [Microsoft support](#) for more assistance with this problem.

# Code 43 - CM\_PROB\_FAILED\_POST\_START

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that a driver has reported a device failure.

## Error Code

43

### Display Message

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

### Recommended Resolution

Uninstall and reinstall the device.

One of the drivers controlling the device told the operating system the device failed in some manner in response to IRP\_MN\_QUERY\_PNP\_DEVICE\_STATE.

## For driver developers

A driver [invalidated the device state](#) of the device and in the resulting query device state the device stack reported [PNP\\_DEVICE\\_FAILED](#).

If the driver is a WDF driver, the reporting of the device as failed may be indirectly caused by the WDF driver calling [WdfDeviceSetFailed](#) or returning an error from a WDF callback. For more info, see [Reporting Device Failures](#).

# Code 44 - CM\_PROB\_HALTED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device has been stopped.

## Error Code

44

### Display Message

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

### Recommended Resolution

You must restart the computer.

# Code 45 - CM\_PROB\_PHANTOM

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is not present.

## Error Code

45

### Display Message

"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

None. This problem code should only appear when the DEVMGR\_SHOW\_NONPRESENT\_DEVICES environment variable is set.

# Code 46 - CM\_PROB\_SYSTEM\_SHUTDOWN

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is not available because the system is shutting down.

## Error Code

46

### Display Message

"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

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

# Code 47 - CM\_PROB\_HELD\_FOR\_EJECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device has been prepared for ejection.

## Error Code

47

### Display Message

"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

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

This error should occur only if the user invokes the hot-plug program to prepare the device for removal (which calls [CM\\_Request\\_Device\\_Eject](#)), or if the user presses a physical eject button (which calls [IoRequestDeviceEject](#)). The user can prepare a device that is currently not removable, such as a CD-ROM trapped between the laptop and the docking station tray.

# Code 48 - CM\_PROB\_DRIVER\_BLOCKED

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the system will not load the driver because it is listed in the Windows Driver Protection database supplied by Windows Update.

## Error Code

48

### Display Message

"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

Obtain a new driver from the hardware vendor.

# Code 49 - CM\_PROB\_REGISTRY\_TOO\_LARGE

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the registry is too large.

## Error Code

49

### Display Message

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

### Recommended Resolution

Set the environment variable DEVMGR\_SHOW\_NONPRESENT\_DEVICES to 1. This causes Device Manager to display installed devices that are currently not present. Use Device Manager to remove these devices. If the registry is still too large, reinstall Windows.

# Code 50 - CM\_PROB\_SetProperties\_Failed

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that device properties cannot be set.

## Error Code

50

### Display Message

"Windows cannot apply all of the properties for this device. Device properties may include information that describes the device's capabilities and settings (such as security settings for example). (Code 50)"

"To fix this problem, you can try reinstalling this device. However, we recommend that you contact the hardware manufacturer for a new driver."

### Recommended Resolution

Try reinstalling the device. If that does not work, obtain a new driver from the hardware vendor.

# Code 51 - CM\_PROB\_WAITING\_ON\_DEPENDENCY

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device did not start because it has a dependency on another device that has not started.

## Error Code

51

### Display Message

"This device is currently waiting on another device or set of devices to start. (Code 51)."

### Recommended Resolution

There is currently no resolution to this problem.

To help diagnose the problem, examine other failed devices in the [device tree](#) that this device might depend on. If you can determine why another related device did not start, you might be able to resolve this issue.

# Code 52 - CM\_PROB\_UNSIGNED\_DRIVER

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device did not start on a 64-bit version of Windows because it has a driver that is not digitally signed. For more information about how to sign drivers, see [Driver Signing](#).

## Error

52

### Display Message

"Windows cannot verify the digital signature for the drivers required for this device. A recent hardware or software change might have installed a file that is signed incorrectly or damaged, or that might be malicious software from an unknown source. (Code 52)"

### Recommended Resolution

The driver does not comply with the [kernel-mode code signing policy](#).

For end-users, the only way to avoid this error is to obtain and install a digitally signed driver for the device.

For driver developers, you can use various methods to load an unsigned driver on a 64-bit version of Windows.

For more information, see [Installing an Unsigned Driver during Development and Test](#).

## See Also

[Code Integrity Diagnostic System Log Events](#)

# Code 53 - CM\_PROB\_USED\_BY\_DEBUGGER

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is reserved for use by the Windows kernel debugger.

## Error

53

### **Display Message (Windows 8.1 and later versions of Windows)**

"This device has been reserved for use by the Windows kernel debugger for the duration of this boot session.  
(Code 53)"

### **Recommended Resolution (Windows 8.1 and later versions of Windows)**

Disable Windows kernel debugging to allow the device to start normally.

# Code 54 - CM\_PROB\_DEVICE\_RESET

3/4/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device has failed and is undergoing a reset.

## Error

54

### **Display Message (Windows 10 and later versions of Windows)**

"This device has failed and is undergoing a reset. (Code 54)"

### **Recommended Resolution (Windows 10 and later versions of Windows)**

This is an intermittent problem code assigned while an ACPI reset method is being executed. If the device never restarts due to a failure, it will be stuck in this state and the system should be rebooted.

# Code 55 - CM\_PROB\_CONSOLE\_LOCKED

12/1/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device is blocked from enumeration due to a Group Policy / MDM (Intune) policy to protect from DMA attacks.

## Error Code

55

### Display Message

"This device is blocked from starting while the user is not logged in. (Code 55)"

### Recommended Resolution

For more info on Kernel DMA Protection, see [Kernel DMA Protection](#).

# Code 56 - CM\_PROB\_NEED\_CLASS\_CONFIG

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that Windows is still setting up the class configuration for this device.

## Error Code

56

### Display Message

"Windows is still setting up the class configuration for this device. (Code 56)"

## For driver developers

This problem code is frequently transient.

In the case of some device setup classes, after the device is installed with a driver package, additional class configuration operations are required to make the device operational. When configuration is complete, the device will be restarted and should no longer have this problem code.

For example, NET-class devices receive additional configuration based on `*IfType`, `UpperRange`, and other networking-specific INF directives.

If a NET-class device continues to have this problem code, the driver INF might have invalid networking-specific INF directives, or the system's network state may be corrupt and need to be reset.

To do this, use the Network Reset button in the Settings app.

# Code 57 - CM\_PROB\_GUEST\_ASSIGNMENT\_FAILED

11/2/2020 • 2 minutes to read • [Edit Online](#)

This Device Manager error message indicates that the device did not start because of a failure during device assignment.

## Error Code

57

### Display Message

"This device could not be assigned to a guest partition. (Code 57)"

### Recommended Resolution

For more info, see [Plan for Deploying Devices using Discrete Device Assignment](#).

# Device Manager Details tab

11/2/2020 • 2 minutes to read • [Edit Online](#)

Device Manager provides a **Details** tab for each device. This tab displays lots of information useful to driver developers and testers, and aids Microsoft Customer Support Services (CSS) in diagnosing customer problems. The tab's page displays [device identification strings](#), together with device and driver configuration information that can be useful when you debug drivers.

## Viewing a device's Details tab

The details tab is enabled by default. To enable this tab, set the user environment variable `DEVMGR_SHOW_DETAILS` to 1. After you set this environment variable, the **Details** tab of the device will be available in Device Manager. To permanently set a user environment variable, use the **Advanced** tab of the system property sheet.

## Providing firmware revision numbers for the Details tab

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`).

# Viewing Hidden Devices

11/2/2020 • 2 minutes to read • [Edit Online](#)

Device Manager lists the devices that are installed in the computer. By default, certain devices are not shown in the list. These *hidden devices* include:

- Devices that have the device node (devnode) status bit DN\_NO\_SHOW\_IN\_DM set.

There is a devnode for each device on a machine and the devnodes are organized into a hierarchical Device Tree. The PnP manager creates a devnode for a device when the device is configured.

A devnode contains the device stack (the device objects for the device's drivers) and information about the device such as whether the device has been started and which drivers have registered for notification on the device.

- Devices that are part of a setup class that is marked as a **NoDisplayClass** in the registry (for example, printers and non-PnP drivers)
- Devices that were physically removed from the computer but whose registry entries were not deleted (also known as nonpresent devices).

## NOTE

Starting with Windows 8 and Windows Server 2012, the Plug-and-Play Manager no longer creates device representations for non-PnP (legacy) devices. Thus there are no such devices to view in the Device Manager.

## NOTE

Users should never have to view nonpresent devices because a nonpresent device should not have their attention and should not cause any problems. If a user has to view your device when it is not present, there is likely a problem with your driver design. However, during testing, a developer might have to view such devices.

To include hidden devices in Device Manager display, select **View** and select **Show hidden devices**.

Prior to Windows 8, to view nonpresent devices, you must set the environment variable DEVMGR\_SHOW\_NONPRESENT\_DEVICES to 1 before you open Device Manager, then open Device Manager, and on the View menu, select **Show hidden devices**.

To permanently set the user environment variable DEVMGR\_SHOW\_NONPRESENT\_DEVICES to 1, use the **Advanced** tab of the system property sheet. After you set this environment variable, run Device Manager and select **Show hidden devices**.

# SetupAPI Logging

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, [SetupAPI](#) includes the following logging components:

- The Plug and Play (PnP) manager and SetupAPI log information about installation events in the device installation text log (*SetupAPI.dev.log*) and the application installation text log (*SetupAPI.app.log*). The device installation text log contains information about device and driver installations and the application installation text log contains information about application software installations that are associated with device driver installations. For more information about the content of SetupAPI text logs, see [SetupAPI Text Logs](#), and for information about how to enable logging, see [SetupAPI Logging Registry Settings](#).
- The [SetupAPI logging functions](#), which PnP device installation applications, class installers, and co-installers can use to write log entries to the SetupAPI text logs. For information about how to use these functions, see [Using the SetupAPI Logging Functions](#).

# SetupAPI Text Logs

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, [SetupAPI](#) supports a device installation text log (*SetupAPI.dev.log*) and an application installation text log (*SetupAPI.app.log*). The Plug and Play (PnP) manager and SetupAPI write entries to the device installation text log to provide information about operations that install devices and drivers. The PnP manager and SetupAPI write entries to the application installation text log that provide information about installation operations other than those that pertain specifically to device and driver installations.

Installation applications, class installers, and co-installers can use the [SetupAPI logging functions](#) to write entries to the device installation log and the application installation text log.

The SetupAPI text logs are ANSI plain text files, which are located by default in the `%SystemRoot%\inf` directory. The text logs are in the English (Standard) language.

The SetupAPI text logs have the following internal format:

- A *log entry* is one line in a text log.
- The first few log entries provide a *text log header* that contains information about the operating system and computer architecture. For more information, see [Format of a Text Log Header](#).
- Following the text log header are zero or more *text log sections*. Each text log section records the events during a single device installation.

The purpose of a text log section is to group and format a contiguous sequence of log entries that provide information about a particular installation operation. By creating text log sections, the PnP manager, SetupAPI, or a custom installation application can organize log entries in a conceptually meaningful way. For example, the PnP manager might create a text log section to group all log entries that apply to installing a device. Text log sections appear in a text log in the order in which they are created. For more information, see [Format of a Text Log Section](#).

- A text log can contain log entries that are not part of the text log header or a text log section. Such entries are associated with operations that are not part of any particular text log section and, in general, are interspersed between text log sections. Log entries that are not part of a text log section appear in the log in the same order in which they are written to the text log. For more information about such log entries, see [Format of Log Entries That Are Not Part of a Text Log Section](#).

# Format of a Text Log Header

11/2/2020 • 2 minutes to read • [Edit Online](#)

A *text log header* consists of the first few log entries in a SetupAPI text log. The text log header contains information about the operating system and system architecture. The following is an example of a text log header:

```
[Device Install Log]
OS Version = 6.0.5033
Service Pack = 0.0
Suite = 0x0100
ProductType = 1
Architecture = x86

[BeginLog]
```

The information in a text log header is a subset of the information that is returned by a call to [GetVersionEx](#) in an [OVSEVERSIONINFOEX](#) structure. For more information, see the Microsoft Windows SDK.

# Format of a Text Log Section

12/1/2020 • 2 minutes to read • [Edit Online](#)

A *text log section* includes a section header that opens the section, a section body that includes a sequence of log entries that apply to the section operation, and a section footer that closes the section. Section entries appear in a section in the same order in which they are written to the section.

The following example of a text log section shows the general format of a typical section, where the fields in italic font style are placeholders for section-specific text, and the remaining text in bold font style is generic text supplied by SetupAPI. The first two log entries comprise the section header and the last two log entries comprise the section footer.

```
>>> [section_title - instance_identifier]
>>> time_stamp Section start
    section body log entry
    section body log entry
    section body log entry
<<< [time_stamp: Section end]
<<< [Exit Status(status_value)]
```

The section body entries that are logged depend on the event level that is set for the log and the category levels that are enabled for the log. For more information about these settings, see [SetupAPI Logging Registry Settings](#).

The following is a typical example of a text log section that the Plug and Play (PnP) manager created to log entries that pertained to the installation of a PCI device. In the section header, the *section\_title* field is "Device Install," the *instance\_identifier* field is the device instance identifier

"PCI\VEN\_104C&DEV\_8019&SUBSYS\_8010104C&REV\_00\3&61aaa01&0&38," and the *time\_stamp* field is "2005/02/13 22:06:28.109::" In the section footer, the *status\_value* field is "0x00000000" and the *time\_stamp* field is "2005/02/13 22:06:20.000::" Only the first three section body log entries are included in this example. The event level for this example was set to TXTLOG\_DETAILS and all category levels were enabled for this example.

```
>>> [Device Install - PCI\VEN_104C&DEV_8019&SUBSYS_8010104C&REV_00\3&61aaa01&0&38]
>>> 2005/02/13 22:06:20.000: Section start
    ndv: Retrieving device info...
    ndv: Setting device parameters...
    ndv: Building driver list...
...
... additional section body log entries, which are not shown
...
<<< [2005/02/13 22:06:28.109: Section end]
<<< [Exit Status(0x00000000)]
```

For detailed information about the content and format of a text log section, see [Format of a Text Log Section Header](#), [Format of a Text Log Section Body](#), and [Format of a Text Log Section Footer](#).

# Format of a Text Log Section Header

12/5/2018 • 2 minutes to read • [Edit Online](#)

A *text log section header* consists of two log entries. The format of the first entry includes a ">>>" prefix string, followed by a *section\_title* field and an *instance\_identifier* field. In the following example, the text in italic font style is a placeholder for section-specific text that is supplied by a section creator, and the text in bold font style is supplied by SetupAPI logging.

```
>>> [section_title - instance_identifier]
```

The *section\_title* field is always present and provides a title for the operation that is associated with the section.

The *instance\_identifier* provides an identifier that, ideally, uniquely identifies the instance of the operation. If the *instance\_identifier* field is not present, the format of the first entry of a section header is as follows:

```
>>> [section_title]
```

The format of the second entry of a section header includes a ">>>" prefix string, followed by a *time\_stamp* field and a "Section start" string, as follows:

```
>>> time_stamp Section start
```

The format of the *time\_stamp* field is as follows:

```
yyyy/mm/dd hh:mm:ss.sss:
```

where the *time\_stamp* subfields are as follows:

- *yyyy* is the 4-digit year, *mm* is the 2-digit month, and *dd* is the 2-digit day
- The local time is based on a 24-hour clock, where *hh* is the 2-digit hour, *mm* is the 2-digit minute, *ss* is the 2-digit number of seconds, and *sss* is the 3-digit number of milliseconds.

The following is an example of a typical section header that the user-mode Plug and Play (PnP) manager would create to group installation operations for a PCI device. The *section\_title* field is "Device Install," the *instance\_identifier* field is the device instance identifier

"PCI\VEN\_104C&DEV\_8019&SUBSYS\_8010104C&REV\_00\3&61aaa01&0&38," and the *time\_stamp* field is "2005/02/13 22:06:28.109::"

```
>>> [Device Install - PCI\VEN_104C&DEV_8019&SUBSYS_8010104C&REV_00\3&61aaa01&0&38]
>>> 2005/02/13 22:06:28.109: Section start
```

# Format of a Text Log Section Body

11/2/2020 • 3 minutes to read • [Edit Online](#)

A *text log section body* contains zero or more log entries that apply to the operation that is associated with a text log section. The format of a section body log entry includes an *entry\_prefix* field, a *time\_stamp* field, an *event\_category* field, an *indentation* field, and a *formatted\_message* field, as follows:

*entry\_prefix* *time\_stamp* *event\_category* *indentation* *formatted\_message*

The maximum length, in characters, of a section body log entry is 336.

## *entry\_prefix* field

Indicates whether the log entry is an error message, a warning message, or an information message. The *entry\_prefix* field is always present and contains one of the strings that are listed in the following table.

ENTRY_PREFIX FIELD	TYPE OF MESSAGE
"!!! "	An error message
"! "	A warning message
" "	Information message other than an error message or a warning message

## *time\_stamp* field

Indicates the system time when the logged event occurred. The *time\_stamp* field is optional and SetupAPI does not include a time stamp by default. However, [SetupWriteTextLog](#) supports including a time stamp in a log entry. The format of the *time\_stamp* field is the same as the format of the *time\_stamp* field that is described in [Format of a Text Log Section Header](#).

## *event\_category* field

Indicates the category of SetupAPI operation that made the log entry. The *event\_category* field is usually present, but is not required. If the *event\_category* field is present, it will contain one of the strings that are listed in the following table.

EVENT_CATEGORY FIELD STRINGS	SETUPAPI OPERATION
". . . : "	Vendor-supplied operation
"bak: "	Backup data

EVENT_CATEGORY FIELD STRINGS	SETUPAPI OPERATION
"cci: "	Class installer or co-installer operation
"cpy: "	Copy files
"dvi: "	Device installation
"flq: "	Manage file queues
"inf: "	Manage INF files
"ndv: "	New device wizard
"prp: "	Manage device and driver properties
"reg: "	Manage registry settings
"set: "	General setup
"sig: "	Verify digital signatures
"sto: "	Manage the driver store
"ui : "	Manage user interface dialog boxes

EVENT_CATEGORY FIELD STRINGS	SETUPAPI OPERATION
"ump: "	User-mode PnP manager

#### *indentation* field

Consists of a sequence of zero or more *indentation units*, where an indentation unit is a monospace string that contains five spaces. The *indentation* field is optional and SetupAPI does not include indentation by default.

**SetupWriteTextLog** supports changing the number of indentation units that are included in a log entry.

#### *formatted\_message* field

Contains the specific information that applies to the log entry.

The section body entries that are logged depend on the event level that is set for the log and the category levels that are enabled for the log. For more information about these settings, see [SetupAPI Logging Registry Settings](#).

When SetupAPI creates a section that groups operations that apply to a device installation, it also recursively groups section body log entries in subsections. SetupAPI distinguishes subsections by the way it annotates and indents log entries. One such subsection appears in the following excerpt from a typical device installation section. The subsection begins with the log entry "dvi: {Build Driver List}" and ends with the log entry "dvi: {Build Driver List - exit(0x00000000)}". This subsection shows a typical sequence of log entries that include the *entry\_prefix*, *event\_category*, *indentation*, and *formatted\_message* fields. The SetupAPI operations that wrote the log entries also created the indentation and supplied the content of the formatted messages. The event level for this example was set to TXTLOG\_DETAILS and all category levels were enabled for this example.

```
>>> [Device Install - PCI\VEN_104C&DEV_8019&SUBSYS_8010104C&REV_00\3&61aaa01&0&38]
>>> 2005/02/13 22:06:28.109: Section start
...
Deleted section body log entries
...
    dvi: {Build Driver List}
    dvi:      Enumerating all INFs...
    dvi:      Found driver match:
    dvi:          HardwareID - PCI\VEN_104C&DEV_8019
    dvi:          InfName   - C:\WINDOWS\inf\1394.inf
    dvi:          DevDesc   - Texas Instruments OHCI Compliant IEEE 1394 Host Controller
    dvi:          DrvDesc   - Texas Instruments OHCI Compliant IEEE 1394 Host Controller
    dvi:          Provider   - Microsoft
    dvi:          Mfg       - Texas Instruments
    dvi:          InstallSec - TIOHCI_Install
    dvi:          ActualSec  - TIOHCI_Install.NT
    dvi:          Rank      - 0x00002001
    dvi:          DrvDate   - 10/01/2002
    dvi:          Version   - 6.0.5033.0
!!! inf:      InfCache: Error flagging 1394.inf for match string pci\ven_104c&dev_8019
    dvi: {Build Driver List - exit(0x00000000)}
...
Deleted section body log entries
...
<<< [2005/02/13 22:06:29.000: Section end]
<<< [Exit Status(0x00000000)]
```

# Format of a Text Log Section Footer

12/5/2018 • 2 minutes to read • [Edit Online](#)

A *text log section footer* closes a text log section. A section footer consists of two entries. The format of the first entry of a section footer includes a "<<< " prefix, followed by a *time\_stamp* field and the string "Section end".

```
<<< [time_stamp Section end]
```

The format of the *time\_stamp* field is the same as that described in [Format of a Text Log Section Header](#).

The format of the second entry includes a "<<< " prefix, followed by the string "Exit" and a *status* field. In the following example, the text in italic font style is a placeholder for section-specific text that is supplied by a section creator and the text in bold font style is supplied by SetupAPI logging:

```
<<< [Exit Status(status)]
```

The format of the status field is "0xhhhhhhhh", where hhhhhhhh is 8-digit hexadecimal number.

If the status field is not present, the format of the first line is as follows:

```
<<< [Exit]
```

The following is an example of a section footer that specifies an exit status of "0x00000000" and includes a *time\_stamp* field.

```
<<< [2005/02/13 22:06:28.109: Section end]
<<< [Exit Status(0x00000000)]
```

# Format of Log Entries That Are Not Part of a Text Log Section

3/6/2019 • 2 minutes to read • [Edit Online](#)

A text log can contain log entries that are not part of a text log header or a text log section. Such entries are not associated with any section and, in general, are interspersed between sections. The format of such log entries consists of an *entry\_prefix* field, a *time\_stamp* field, an *event\_category* field, and a *formatted\_message* field, as follows:

*entry\_prefix* *time\_stamp* *event\_category* *formatted\_message*

The following list describes the fields of a log entry:

## *entry\_prefix* field

Indicates the message type. The *entry\_prefix* field is always present and contains one of the strings that are listed in the left-hand column of the following table, where the meaning of the string is indicated in the right-hand column.

ENTRY_PREFIX FIELD	MESSAGE TYPE
"!!! "	An error message in a text log
"! "	A warning message in a text log
" . "	An information message in a text log (other than an error message or a warning message)
" " "	An information message in the application installation text log (other than an error message or a warning message)

## *time\_stamp* field

Indicates the system time when the logged event occurred. The *time\_stamp* field is optional and will be present only if an installation application requested that a time stamp be included for a log entry. The format of the *time\_stamp* field is the same as that described in [Format of a Text Log Section Header](#).

## *event\_category* field

Indicates the category of the SetupAPI operation that made the log entry. The *event\_category* field is usually present, but is not required. If present, the *event\_category* field contains one of the strings that are listed in [Format of a Text Log Section Body](#).

## *formatted\_message*

Contains the information that is specific to the log entry. The *formatted\_message* field is generally present, but is not required.

**Note** The maximum length, in characters, of a log entry is 336.

The following example of text log entries is taken from a device installation text log. In the example, the first two log entries are not part of a text log section. The user-mode Plug and Play (PnP) manager wrote these log entries in the device installation text log to indicate the start of a server-side installation of a PCI device. The server-side installation, in turn, created the text log section that is indicated by the text log section header that follows the first two log entries in the example.

Be aware that the *event\_category* field for the first two log entries indicates that the user-mode PnP manager wrote these log entries.

```
. ump: Start service install for: PCI\VEN_104C&DEV_8019&SUBSYS_8010104C&REV_00\3&61aaa01&0&38
. ump: Creating Install Process: rundll32.exe

>>> [Device Install - PCI\VEN_104C&DEV_8019&SUBSYS_8010104C&REV_00\3&61aaa01&0&38]
>>> 2005/02/13 22:06:28.109: Section start
```

# SetupAPI Logging Registry Settings

12/5/2018 • 2 minutes to read • [Edit Online](#)

SetupAPI logging supports a global *event level* and a global *event category* that control whether information is written to a text log. The event level controls the level of detail that is written to a text log and the event category determines the type of operations that can make log entries. If a log entry has an event level numerically less than or equal to the global event level of a text log, and if the event category of the log entry is enabled for the text log, the log entry is written to the text log; otherwise, the log entry is not written to the text log.

For information about how to set the event level, see [Setting the Event Level for a Text Log](#).

For information about how to set the event categories that are enabled for a log, see [Enabling Event Categories for a Text Log](#).

By default, the SetupAPI text logs are located in the %SystemRoot%\Inf directory. For information about how to change the directory where the text logs are located, see [Setting the Directory Path of the Text Logs](#).

# Setting the Event Level for a Text Log

11/2/2020 • 3 minutes to read • [Edit Online](#)

SetupAPI writes a log entry to a text log only if the event level set for a text log is greater than or equal to the event level for the log entry, and the [event category](#) for the log entry is enabled for the text log.

The following table lists the event levels that SetupAPI supports and the manifest constants that represent these event levels. TXTLOG\_ERROR is the lowest event level, followed by the next highest event level TXTLOG\_WARNING, and so on. TXTLOG VERY\_VERBOSE is the highest event level.

EVENT LEVEL	EVENT LEVEL MANIFEST CONSTANT	EVENT LEVEL MANIFEST VALUE
Write errors only.	TXTLOG_ERROR	1
Write errors and warnings of potential problems.	TXTLOG_WARNING	2
Write errors, warnings, and system state changes.	TXTLOG_SYSTEM_STATE_CHANGE	3
Write errors, warnings, system state changes, and high-level operations that are associated with state changes.	TXTLOG_SUMMARY	4
Write errors, warnings, system state changes, high-level operations that are associated with state changes, and most operational details.	TXTLOG_DETAILS	5
Write errors, warnings, system state changes, high-level operations that are associated with state changes, and all operational details.	TXTLOG_VERBOSE	6
Write all log entries, including those that might generate a large amount of information that is frequently superfluous.	TXTLOG VERY_VERBOSE	7

To set the event level for the SetupAPI text logs, create (or modify) the following [REG\\_DWORD](#) registry value:  
`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup\LogLevel`

If the `LogLevel` registry value does not exist or has a value of zero, SetupAPI sets the event level for the application installation and device installation text logs to the default values described in the following table:

TEXT LOG	DEFAULT VALUE (WINDOWS 7 AND LATER VERSIONS)	DEFAULT VALUE (WINDOWS VISTA SP2)	DEFAULT VALUE (WINDOWS VISTA SP1 AND PREVIOUS VERSIONS)
Application installation text log ( <i>SetupAPI.app.log</i> )	TXTLOG_SUMMARY	TXTLOG_WARNING	TXTLOG_DETAILS
Device installation text log ( <i>SetupAPI.dev.log</i> )	TXTLOG_DETAILS	TXTLOG_DETAILS	TXTLOG_DETAILS

For more information about these text log files, see [SetupAPI Text Logs](#).

The **LogLevel** registry value is formatted as  $0xUUUUGHVW$ , where:

- The low-order eight bits, represented by the mask  $0x000000VW$ , specify whether logging is turned on for the application installation log and specify the event level for the application log.
- The next highest eight bits, represented by the mask  $0x0000GH00$ , specify whether logging is turned on for the device installation text log and specify the event level for the device installation text log.
- The highest-level bits, represented by the mask  $0xUUUU0000$ , are not used.

The value of the  $0xVW$  bits controls logging for the application installation log as shown in the following table.

0XVW VALUE	DESCRIPTION
Zero (default)	Logging is turned on and the event level is set to the default value as described previously.
0x01 through 0x0F	Turns logging off.
0x10 through 0x7F	Turns logging on and sets the event level to $0xV$ .

The value of the  $0xGH$  bits controls logging for the device installation text log as shown in the following table.

0XGH VALUE	DESCRIPTION
Zero (default)	Logging is turned on and the event level is set to the default value as described previously.
0x01 through 0x0F	Turns logging off.
0x10 through 0x7F	Turns logging on and sets the event level to $0xG$ .

The following table provides examples of typical **LogLevel** values.

LOGLEVEL VALUE	EVENT LEVELS SET FOR THE TEXT LOGS
----------------	------------------------------------

LOGLEVEL VALUE	EVENT LEVELS SET FOR THE TEXT LOGS
0x00000000	By default, turns logging on for the application installation log and the device installation log. Sets the logging level to the default values for both logs.
0x00000101	Turns logging off for both the application installation log and the device installation log.
0x00001010	Turns logging on for the application installation log and the device installation log. Sets the logging level to TXTLOG_ERROR for both logs.
0x00002020	Turns logging on for the application installation log and the device installation log. Sets the logging level to TXTLOG_WARNING for both logs.
0x00005050	Turns logging on for the application installation log and the device installation log. Sets the logging level to TXTLOG_DETAILS for both logs.
0x00006060	Turns logging on for the application installation log and the device installation log. Sets the logging level to TXTLOG_VERBOSE for both logs.
0x00007070	Turns logging on for the application installation log and the device installation log. Sets the logging level to TXTLOG VERY_VERBOSE for both logs.

# Enabling Event Categories for a Text Log

11/2/2020 • 2 minutes to read • [Edit Online](#)

SetupAPI writes a log entry in a text log only if the event category for the log entry is enabled for the text log and the [event level](#) for the text log is equal to or greater than the event level for the log entry.

The following table lists the event categories that SetupAPI supports, the manifest constants that represent the event categories, and the values of the manifest constants.

EVENT CATEGORY OPERATION	EVENT CATEGORY MANIFEST CONSTANT	EVENT CATEGORY VALUE
Device installation	TXTLOG_DEVINST	0x00000001
Manage INF files	TXTLOG_INF	0x00000002
Manage file queues	TXTLOG_FILEQ	0x00000004
Copy files	TXTLOG_COPYFILES	0x00000008
Manage registry settings	TXTLOG_REGISTRY	0x00000010
Verify digital signatures	TXTLOG_SIGVERIF	0x00000020
Manage device and driver properties	TXTLOG_PROPERTIES	0x00000040
Backup data	TXTLOG_BACKUP	0x00000080
Manage user interface dialog boxes	TXTLOG_UI	0x00000100
New device manager	TXTLOG_NEWDEV	0x01000000
User-mode PnP manager	TXTLOG_UMPNPMGR	0x02000000
Manage the driver store	TXTLOG_DRIVER_STORE	0x04000000
Class installer or co-installer operation	TXTLOG_INSTALLER	0x40000000
Vendor-supplied operation	TXTLOG_VENDOR	0x80000000

To enable event categories for the SetupAPI logs, create (or modify) the following **REG\_DWORD** registry value:  
**HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup\LogMask**

The **LogMask** registry value applies to the device installation text log and the application installation text log.

If the **LogMask** registry value does not exist, SetupAPI enables all event categories for the text logs. If the **LogMask** registry value is zero, SetupAPI disables all event categories for the text logs.

The **LogMask** registry value is formatted as **0XVVVVVVVV**, where **VVVVVVVV** is a 32-bitfield. To enable all categories, set **LogMask** to **0xFFFFFFFF**. To enable only specific categories, perform a bitwise OR of the corresponding event category constants. For example:

- To enable only log entries that are written by device installation operations, set **LogMask** to the value of **TXTLOG\_DEVINST** (**0X00000001**)
- To enable only log entries that are written by device installation operations and driver store operations, set **LogMask** to (**TXTLOG\_DRIVER\_STORE | TEXTLOG\_DEVINST**) (**0x04000001**).
- To enable only log entries that are written by custom installation operations, set **LogMask** to **TXTLOG\_VENDOR** (**0x80000000**).

# Setting the Directory Path of the Text Logs

11/2/2020 • 2 minutes to read • [Edit Online](#)

By default, the SetupAPI text logs are located in the system Windows directory. The location of the SetupAPI text logs can be changed by setting the following **REG\_SZ** registry value:

**HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup\LogPath**

The **LogPath** registry value must be a fully qualified directory path. The path must exist, and the path cannot include a file name.

If the **LogPath** registry value is not present, the path does not exist, or the path includes a file name, SetupAPI locates the text logs in the **%SystemRoot%\Inf** directory.

# Using the SetupAPI Logging Functions

11/2/2020 • 2 minutes to read • [Edit Online](#)

SetupAPI supports functions that installation applications, class installers, and co-installers can use to write log entries in the SetupAPI text logs, as follows:

- To write log entries in a [SetupAPI text log](#), an installation application calls [SetupWriteTextLog](#), [SetupWriteTextLogError](#), or [SetupWriteTextLogInfoLine](#). For more information about how to write text log entries, see [Writing Log Entries in a Text Log](#).
- SetupAPI supports a mechanism to establish a log context for a thread. A log context is established for a thread by setting a log token for the thread. To set a log token for a thread, an installation application calls [SetupSetThreadLogToken](#). To retrieve a log token for a thread, an installation application calls [SetupGetThreadLogToken](#).

For more information about log tokens, see [Log Tokens](#).

For more information about how to use log tokens, see [Setting and Getting a Log Token for a Thread](#).

# Writing Log Entries in a Text Log

12/5/2018 • 2 minutes to read • [Edit Online](#)

An application performs one of the following to write a log entry in a [SetupAPI text log](#):

- [Calls SetupWriteTextLog](#) to write a single text log entry that contains information about an installation event.
- [Calls SetupWriteTextLogError](#) to write information about a SetupAPI-specific error or a Win32 error to a text log. [SetupWriteTextLogError](#) writes two consecutive entries to a text log: the first entry contains the same information in the same format as that written by [SetupWriteTextLog](#) and the second entry logs a corresponding error code and a user-friendly description of the error.
- [Calls SetupWriteTextLogInfLine](#) to write a single text log entry that contains the text of a specified INF file line.

As described in [Format of a Text Log Section Body](#), the SetupAPI logging functions write entries in the following format:

```
entry_prefix time_stamp event_category indentation formatted_message
```

The main difference between the entries that the various SetupAPI logging functions write to a text log is in the specific information content and in the format of the *formatted-message* field.

For information about how to set the *indentation* field to indent the *formatted-message* field, see [Writing Indented Log Entries](#).

# Calling SetupWriteTextLog

11/2/2020 • 2 minutes to read • [Edit Online](#)

**SetupWriteTextLog** adds a single entry with information about an installation event to a [SetupAPI text log](#).

As described in [Format of a Text Log Section Body](#), the format of a log entry consists of the following fields:

```
entry_prefix time_stamp event_category indentation formatted_message
```

To call **SetupWriteTextLog**, an application supplies following information:

- The log token for a section in a text log that was obtained by calling [SetupGetThreadLogToken](#), or one of the system-defined [log tokens](#). If the log token is associated with a text log section, **SetupWriteTextLog** writes the log entry in that section. Otherwise, **SetupWriteTextLog** adds the log entry to a part of the log that is not included in a text log section. In addition, whether **SetupWriteTextLog** writes a log entry, and to which text log **SetupWriteTextLog** writes the entry, depends on the system-defined log token value.

For more information about log tokens, see [Setting and Getting a Log Token for a Thread](#).

- One of the event categories that are described in [Enabling Event Categories for a Text Log](#). If the event category for the entry is enabled for the text log, **SetupWriteTextLog** adds the entry to the text log; otherwise, **SetupWriteTextLog** does not write the entry to the text log.
- A flag value that is a bitwise OR of system-defined constants that specify the event level, the indentation depth, and whether to include a time stamp. Event levels are described in [Setting the Event Level for a Text Log](#). If the event level set for the text log is greater than or equal to the event level for the entry, **SetupWriteTextLog** writes a log entry to the text log; otherwise, **SetupWriteTextLog** does not write a log entry to the text log. By using indentation, formatted messages can be arranged to make the information in a section easier to read and understand. For more information, see [Writing Indented Log Entries](#).
- A `printf`-compatible format string that formats both the message and the list of comma-separated variables that follows the format string.
- A comma-separated list of variables, whose values are formatted by the `printf`-compatible format string.

For an example on how to call **SetupWriteTextLog** to log information about an event that is not an error or a warning, see [Writing an Information Log Entry](#).

For an example on how to call **SetupWriteTextLog** to log information about an error or a warning, see [Writing an Error or Warning Log Entry](#).

# Writing an Information Log Entry

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following example shows how an application might typically call [SetupWriteTextLog](#) to write an information entry in a [SetupAPI text log](#) that is not a warning message or an error message.

For information about calling [SetupWriteTextLog](#) to log an error message, see [Calling SetupWriteTextLog to Log an Error or Warning Entry](#).

The application calls [SetupWriteTextLog](#), supplying the following parameter values:

- *LogToken* is set to a log token value that either was obtained by calling [SetupGetThreadLogToken](#) or is one of the system-defined log token values that are described in the [Log Tokens](#).
- *Category* is set to [TXTLOG\\_VENDOR](#), which indicates that the log entry is made by a vendor-supplied application. Event categories are described in [Enabling Event Categories for a Text Log](#).
- *Flags* is set to a bitwise OR of [TXTLOG\\_DETAILS](#) and [TXTLOG\\_TIMESTAMP](#). In this example, the indentation depth is not changed and the current indentation depth was previously set to five monospace text spaces. For information about how to change the indentation depth, see [Writing Indented Log Entries](#). Event levels are described in the [Setting the Event Level for a Text Log](#) topic.
- *MessageStr* is set to `TEXT("Variable of interest: = %d")`.
- The comma-separated parameter list supplies the variable *SomeVariable*, which corresponds to "%d" field in *MessageStr*.

```
//The LogToken value was previously returned by call to
//SetupGetThreadLogToken or one of the system-defined log token values
DWORD Category = TXTLOG_VENDOR;
DWORD Flags = TXTLOG_DETAILS | TXTLOG_TIMESTAMP;
DWORD SomeVariable = 1;    // The variable whose value will be logged

SetupWriteTextLog(LogToken, Category, Flags, TEXT("Variable of interest: = %d"), SomeVariable);
```

If the [TXTLOG\\_VENDOR](#) event category is enabled and the [TXTLOG\\_DETAILS](#) event level is set for the device installation text log, this code would create an entry in the device installation log in the following format, where the time stamp would be replaced by an actual time stamp.

```
2005/02/13 22:06:28.109:      : Variable of interest: Abc = 1
```

# Writing an Error or Warning Log Entry

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following example shows how an application might typically call [SetupWriteTextLog](#) to write an error or warning entry in a [SetupAPI text log](#). However, if an event is associated with a SetupAPI-specific error or a Win32 error, an application can call [SetupWriteTextLogError](#) instead. [SetupWriteTextLogError](#) facilitates logging and interpreting information about these types of errors.

For information about calling [SetupWriteTextLog](#) to log an error message, see [Logging an Error Message](#) and for information about calling [SetupWriteTextLog](#) to log a warning message, see [Logging a Warning Message](#).

## Logging an Error Message

In this example, the application calls [SetupWriteTextLog](#), supplying the following parameter values:

- *LogToken* is set to a log token value that either was obtained by calling [SetupGetThreadLogToken](#) or is one of the system-defined log token values that are described in the [Log Tokens](#).
- *Category* is set to [TXTLOG\\_VENDOR](#), which indicates that the log entry is made by a vendor-supplied application. Event categories are described in [Enabling Event Categories for a Text Log](#).
- *Flags* is set to a bitwise OR of [TXTLOG\\_ERROR](#) and [TXTLOG\\_TIMESTAMP](#). In this example, the indentation depth is not changed and the current indentation depth was previously set to five monospace text spaces. For information about how to change the indentation depth, see [Writing Indented Log Entries](#). Event levels are described in the [Setting the Event Level for a Text Log](#) topic.
- *MessageStr* is set to [TEXT\("Application Error \(%d\)"\)](#).
- The comma-separated list supplies the variable *ErrorCode*.

The following code calls [SetupWriteTextLog](#) to write a log entry for this example:

```
//The LogToken value was previously returned by call to
//SetupGetThreadLogToken or one of the system-defined log token values
DWORD Category = TXTLOG_VENDOR;
DWORD Flags = TXTLOG_ERROR | TXTLOG_TIMESTAMP;
DWORD ErrorCode = 1111; // An error code value

SetupWriteTextLog(LogToken, Category, Flags, TEXT("Application Error (%d)'), ErrorCode);
```

If the [TXTLOG\\_VENDOR](#) event category is enabled and the [TXTLOG\\_ERROR](#) event level is set for the text log, this code would create an entry in the text log that would be formatted as follows:

```
!!! 2005/02/13 22:06:28.109: : Application error (1111)
```

The *entry\_prefix* field "!!! " indicates that the log entry is an error message.

## Logging a Warning Message

Logging a warning message is almost identical to logging an error message. The difference is the settings for the event level. Set *Flags* to [TXTLOG\\_WARNING](#) instead of [TXTLOG\\_ERROR](#). If [SetupWriteTextLog](#) is called as described in [Logging an Error Message](#), except that *Flags* is set to a bitwise OR of [TXTLOG\\_WARNING](#) and [TXTLOG\\_TIMESTAMP](#), [SetupWriteTextLog](#) would write the following log entry:

```
! 2005/02/13 22:06:28.109: : Application error (1111)
```

The *entry\_prefix* field of the log entry is "!", which indicates that this is a warning message, as opposed to "!!!", which would indicate an error message.

# Calling SetupWriteTextLogError

11/2/2020 • 2 minutes to read • [Edit Online](#)

**SetupWriteTextLogError** writes information about a SetupAPI-specific error or a Win32 error to a [SetupAPI text log](#). **SetupWriteTextLogError** writes two consecutive entries to a text log: the first entry contains the same information in the same format as that written by [SetupWriteTextLog](#) and the second entry logs a corresponding error code and a user-friendly description of the error.

To call **SetupWriteTextLogError**, an application supplies the same information that it would supply to call [SetupWriteTextLog](#) and, in addition, supplies the value of a SetupAPI-specific error or a Win32 error.

**SetupWriteTextLogError** writes the first log entry in the following format:

*entry\_prefix time\_stamp category \*\*\*\*indentation formatted-message*

**SetupWriteTextLogError** writes the second log entry in the following format:

*entry\_prefix time\_stamp category \*\*indentation \*\*Error:error-number error-description*

Where:

- The *entry\_prefix*, *time-stamp*, *category*, *indentation*, and *formatted-message* fields are the same as those that are described in [Format of a Text Log Section Body](#).
- The *error-number* field contains the error number.
- The *error-description* field contains a user-friendly description of the error.

The following example shows how an application might typically call **SetupWriteTextLogError** to log information about an error in a text log. The error used in the example is a system start error. The application calls **SetupWriteTextLogError**, supplying the following parameter values:

- *LogToken* is set to a log token value that either was obtained by calling [SetupGetThreadLogToken](#) or is one of the system-defined log token values that are described in [Log Tokens](#).
- *Category* is set to [TXTLOG\\_VENDOR](#), which indicates that the log entry is made by a vendor-supplied application. Event categories are described in [Enabling Event Categories for a Text Log](#).
- *LogFlags* is set to [TXTLOG\\_ERROR](#). This example does not include a time stamp or change the indentation depth. The current indentation depth was previously set to five monospace text spaces. For information about how to change the indentation depth, see [Writing Indented Log Entries](#). Event levels are described in [Setting the Event Level for a Text Log](#).
- *Error* is set to the value of the Win32 error code, [ERROR\\_SERVICE\\_ALREADY\\_RUNNING](#). The decimal value of this error code is 1056.
- *MessageStr* is set to `TEXT("Start Service: Failed to start service 'SomeService'")`.
- A comma-separated parameter list is not supplied.

The parameters *LogToken*, *Category*, and *LogFlags* affect the operation of **SetupWriteTextLogError** in the same manner as these parameters affect the operation of **SetupWriteTextLog**.

The following code calls **SetupWriteTextLogError** to write the log entry for this example:

```
//The LogToken value was previously returned by call to
//SetupGetThreadLogToken or one of the system-defined log token values
DWORD Category = TXTLOG_VENDOR;
DWORD Flags = TXTLOG_ERROR ;
DWORD ErrorCode = 1056; // The corresponding Win32 error code

SetupWriteTextLog(LogToken, Category, Flags, ErrorCode, TEXT("Start Service: Failed to start service
'SomeService'"),);
```

If the TXTLOG\_VENDOR event category is enabled and the TXTLOG\_ERROR event level is set for the text log, this code would create an entry in the text log that would be formatted as follows:

```
!!!      : Start Service: Failed to start service 'SomeService'
!!!      : Error 1056: An instance of the service is already running.
```

Be aware that **SetupWriteTextLogError** provides the string "An instance of the service is already running." to describe the Win32 error whose value is 1056.

# Calling SetupWriteTextLogInfLine

11/2/2020 • 2 minutes to read • [Edit Online](#)

An application can call [SetupWriteTextLogInfLine](#) to write a log entry in a [SetupAPI text log](#) that contains the text of a specified INF file line.

To call [SetupWriteTextLogInfLine](#), an application supplies the following information:

- The log token for a section in a text log that was obtained by calling [SetupGetThreadLogToken](#) or one of the system-defined [log tokens](#). If the log token is associated with a text log section, [SetupWriteTextLogInfLine](#) writes the log entry in that section. Otherwise, [SetupWriteTextLogInfLine](#) adds the log entry to a part of the log that is not included in a text log section.

In addition, whether [SetupWriteTextLogInfLine](#) writes a log entry, and to which text log [SetupWriteTextLogInfLine](#) writes the entry, depends on the system-defined log token value.

For more information about log tokens, see [Setting and Getting a Log Token for a Thread](#).

- A flag value that is a bitwise OR of system-defined constants that specify the event level, the indentation depth, and whether to include a time stamp. Event levels are described in [Setting the Event Level for a Text Log](#).

If the event level set for the text log is greater than or equal to the event level for the entry, [SetupWriteTextLogInfLine](#) writes a log entry in the text log. Otherwise, [SetupWriteTextLogInfLine](#) does not write a log entry in the text log. By using indentation, formatted messages can be arranged to make the information in a section easier to read and understand.

For more information, see [Writing Indented Log Entries](#).

- A handle to the INF file that contains the INF file line.
- The context for the INF file line.

[SetupWriteTextLogInfLine](#) writes a log entry in the following format:

*entry\_prefix time\_stamp inf:indentation inf-line-text (inf-file-name line line-number)*

Where:

- The *entry\_prefix*, *time-stamp*, and *indentation* fields are the same as those that are described in [Format of a Text Log Section Body](#).
- The *inf:* field specifies the TXTLOG\_INF event category. Event categories are described in [Enabling Event Categories for a Text Log](#).
- The *inf-line-text* field contains the text of the specified INF file line.
- The *inf-file-name* field contains the name of the INF file that contains the specified INF file line.
- The *line* field indicates that what follows is a line number in the INF file.
- The *line-number* field contains the line number of the specified line in the INF file.

The following example shows how an application might typically log the text of an INF line in a text log. The INF line in this example is an INF **AddReg** line. The application calls [SetupWriteTextLogInfLine](#), supplying the following input parameter values:

- *LogToken* is set to a log token that was returned by [SetupGetThreadLogToken](#) or to a system-defined [log token](#).
- *LogFlags* is set to TXTLOG\_DETAILS. This example does not include a time stamp or change the indentation depth. In the example, the indentation depth is five monospace text spaces.
- *InfHandle* is set to a handle to the INF file *hidserv.inf*. This handle is obtained by calling the [SetupOpenInfFile](#) function, which is documented in the Platform SDK.
- *Context* is set to the INF file context of the INF file line that contains the text "AddReg=HidServ\_AddService\_AddReg." An INF file context for the line is obtained by calling the [SetupFindXxxLine](#) functions, which are documented in the Platform SDK.

The values of *LogToken* and *LogFlags* affect the operation of [SetupWriteTextLogInfLine](#) in the same manner as that described for [SetupWriteTextLog](#). In addition, [SetupWriteTextLogInfLine](#) uses the event catalog TXTLOG\_INF.

For this example, the following shows the type of log entry that [SetupWriteTextLogInfLine](#) would write to a text log:

```
inf:      AddReg=HidServ_AddService_AddReg  (hidserv.inf line 98)
```

# Writing Indented Log Entries

11/2/2020 • 2 minutes to read • [Edit Online](#)

As described in [Format of a Text Log Section Body](#), the format of a section body log entry in a [SetupAPI text log](#) consists of the following fields:

```
entry_prefix time_stamp event_category indentation formatted_message
```

You can use the *indentation* field in log entries to indent the *formatted\_message* fields in order to make the log entries easier to read and understand. The amount of indentation in an indentation field depends on the indentation depth that is set for the section. The indentation depth is the number of indentation units, where an indentation unit is five monospace text spaces. For example, an indentation depth of 1 results in an indentation of 5 spaces, an indentation depth of 2 results in an indentation of 10 spaces, and so on. The minimum indentation depth is zero and the maximum indentation depth is 16.

By default, the indentation depth of a section is zero. If the indentation depth is zero, the *formatted\_message* field will not be indented. If an application increases the indentation depth to write a sequence of indented section entries, the application must also write a corresponding set of section entries to reset the indentation depth to zero before the application can subsequently write additional section entries that are not indented.

To change the indentation depth for a section, call a SetupAPI logging function and use a bitwise OR between one of following system-defined manifest constants and the flags parameter that is supplied to the SetupAPI logging function.

MANIFEST CONSTANT	CHANGE IN INDENTATION DEPTH
TXTLOG_DEPTH_INCR	The indentation depth is increased by 1 for the current log entry and all subsequent log entries.
TXTLOG_DEPTH_DECR	The indentation depth is decreased by 1 for the current log entry and all subsequent log entries.
TXTLOG_TAB_1	The indentation depth is increased by 1 only for the current log entry.

For example, the following sequence of calls to [SetupWriteTextLog](#) writes a sequence of indented log entries after the section header whose *section\_title* field is "Indentation Example" and whose *instance\_identifier* field is "Instance 0".

```

// The LogToken value was previously returned by a call to
// SetupGetThreadLogToken.
// The LogToken value specifies a section in one of the text logs.

DWORD Category = TXTLOG_VENDOR;
DWORD Flags = TXTLOG_DETAILS;

SetupWriteTextLog(LogToken, Category, Flags, TEXT("Subsection A"));

// Additional SetupWriteTextLog calls that write entries at Subsection A indentation level

SetupWriteTextLog(LogToken, Category, Flags | TXTLOG_DEPTH_INCR, TEXT("Subsection A.1"));

// Additional SetupWriteTextLog calls that write entries at Subsection A.1 indentation level

SetupWriteTextLog(LogToken, Category, Flags | TXTLOG_DEPTH_INCR, TEXT("Subsection A.1.1"));

// Additional SetupWriteTextLog calls that write entries at Subsection A.1.1 indentation level

SetupWriteTextLog(LogToken, Category, Flags, TEXT("End of Subsection A.1.1"));

// Additional SetupWriteTextLog calls that write entries at Subsection A.1 indentation level

SetupWriteTextLog(LogToken, Category, Flags | TXTLOG_DEPTH_DECR, TEXT("End of Subsection A.1"));

// Additional SetupWriteTextLog calls that write entries at Subsection A indentation level
SetupWriteTextLog(LogToken, Category, Flags | TXTLOG_DEPTH_DECR, TEXT("End of Subsection A"));

```

If the event level for the text log is greater than or equal to `TXTLOG_DETAILS` and the event category `TXTLOG_VENDOR` is enabled for the text log, the previous code would write the following log entries after the section header.

In the following example, ellipsis (...) represents zero or more additional log entries at the same level of indentation as the previous log entry. The time stamp would be replaced by an actual time stamp.

```

>>> [Indentation Example - Instance 0]
>>> 2005/02/13 22:06:28.109: Section start
    : Subsection A
...
    :     Subsection A.1
...
    :         Subsection A.1.1
...
    :             End Subsection A.1.1
...
    :                 End of Subsection A.1
...
    :                     End of Subsection A

```

For another example of indented section entries that was taken from an actual text log, see [Format of a Text Log Section Body](#).

# Log Tokens

11/2/2020 • 2 minutes to read • [Edit Online](#)

SetupAPI text logging uses *log tokens* to write entries in a [SetupAPI text log](#).

A class installer or co-installer must use the log token that is returned by [SetupGetThreadLogToken](#) to write log entries in a [text log section](#) that was established by the SetupAPI installation operation that called the installer. SetupAPI text logging also provides system-defined log tokens, which an installation application can use to write log entries that are not part of a text log section.

The following system-defined log tokens are provided by SetupAPI text logging:

## LOGTOKEN\_UNSPECIFIED

Represents the part of an unspecified text log that is not part of a [text log section](#). By default, the [SetupAPI logging functions](#) write a log entry in the application installation text log if this token value is specified.

## LOGTOKEN\_NO\_LOG

Represents the null log. The SetupAPI logging functions do not write a log entry if this token value is specified.

## LOGTOKEN\_SETUPAPI\_APPLOG

Represents the part of the application text log (*SetupAPI.app.log*) that is not part of a text log section. The [SetupAPI logging functions](#) write log entries in the application installation text log if this token value is specified.

## LOGTOKEN\_SETUPAPI\_DEVLOG

Represents the part of the device installation text log (*SetupAPI.dev.log*) that is not part of a text log section. The SetupAPI logging functions write log entries in the device installation text log if this token value is specified.

# Setting and Getting a Log Token for a Thread

11/2/2020 • 3 minutes to read • [Edit Online](#)

SetupAPI logging supports a mechanism that establishes a log context for a thread. This context is established by setting a [log token](#) for the thread. SetupAPI provides this mechanism so that code that is called by a thread can write log entries to the log context of the calling thread.

For example, a thread can set a log token for its log context before it calls a class installer or co-installer. The installer, in turn, can retrieve the calling thread's log token and use that token to write log entries in the text log and section that is associated with the calling thread's log context.

## Setting a log token for a thread

The [SetupSetThreadLogToken](#) function sets a log token for the thread from which this function was called. The log token can either be a system-defined log token or a log token that was retrieved by calling [SetupGetThreadLogToken](#).

The following are examples of how a log context can be established for a thread:

- An installation application can call [SetupSetThreadLogToken](#) to establish a log context for other installation code that runs within the same thread. When it is establishing the log context for the thread, the application should use a system-defined [log token](#), such as LOGTOKEN\_SETUPAPI\_APPLOG, in the call to [SetupSetThreadLogToken](#).

**Note** If the log context is set by using a system-defined [log token](#), subsequent calls to a [SetupAPI logging function](#) that are made from that log context, write log entries to the installation text log, which are not part of a [text log section](#).

- If a class installer or co-installer starts a new thread, the installer can set the log context for that thread to be the same as the parent thread. This is done in the following way:

1. Before the parent thread starts the new thread, it acquires the current log token by calling [SetupGetThreadLogToken](#).
2. The parent thread starts the new thread and passes the current log token through an implementation-specific method, such as saving the token in a global variable.
3. The new thread calls [SetupSetThreadLogToken](#) with the current log token. As a result, the new thread "inherits" the log context of the parent thread.

**Note** If a thread of a class installer or co-installer sets the log context by using this method, subsequent calls to a [SetupAPI logging function](#) that are made from that log context write log entries to the installation text log that may be part of a [text log section](#). This only happens if a text log section was established by the SetupAPI installation operation that called the installer.

The following is an example of a call to [SetupSetThreadLogToken](#) that sets the log context of the current thread to the device installation text log (*SetupAPI.app.log*) by specifying the system-defined log token of LOGTOKEN\_SETUPAPI\_APPLOG. A subsequent call to a [SetupAPI logging function](#) that uses this log context would write the log entry to the device installation text log, but not as part of a [text log section](#).

```
SP_LOG_TOKEN LogToken = LOGTOKEN_SETUPAPI_APPLOG;
SetupSetThreadLogToken(LogToken);
```

## Getting a log token for a thread

The [SetupGetThreadLogToken](#) function retrieves a log token for the thread from which this function was called.

For example, a class installer can call **SetupGetThreadLogToken** to retrieve the log token that applies to the SetupAPI operation that called the class installer. The class installer can then use this retrieved log token to log entries in the text log that applies to the corresponding SetupAPI operation.

**Note** If the log context of a thread was not previously set by a call to **SetupSetThreadLogToken**, a call to **SetupGetThreadLogToken** returns a log token with a value of LOGTOKEN\_UNSPECIFIED.

The following is an example of a call to **SetupGetThreadLogToken** that retrieves the log token for the current thread.

```
SP_LOG_TOKEN LogToken = SetupGetThreadLogToken();
```

# SetupAPI Logging (Windows Server 2003, Windows XP, and Windows 2000)

11/2/2020 • 2 minutes to read • [Edit Online](#)

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 \(SetupXxx functions\)](#) 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](#)

# Setting SetupAPI Logging Levels

3/5/2019 • 4 minutes to read • [Edit Online](#)

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

To change the level of information written to the SetupAPI log for all device installation 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 device installation 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 device installation 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 device installation logging).
0x00000010	Log errors.
0x00000020	Log errors and warnings.
0x00000030	Log errors, warnings and other information.

GENERAL LOGGING LEVELS	MEANING
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 device installation 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.

DEVICE LOGGING LEVELS	MEANING
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 the **Registry** menu and select **Load Hive**.

On Windows XP and later, run *RegEdit*. Highlight HKEY\_LOCAL\_MACHINE, click 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 the **Security** menu, select **Permissions**, and grant full control to Administrator.

On Windows XP and later, click 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 **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.

# Interpreting a Sample SetupAPI Log File

12/5/2018 • 5 minutes to read • [Edit Online](#)

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.

## 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
#Ennn	Error message.
#Wnnn	Warning message.
#Innn	Informational message.
#Tnnn	Timing message.
#Vnnn	Verbose message.
#-nnn	Status message.

The following segments are 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 catalog
 "C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5INF.CAT".
 @ 20:14:33.810 #V132 File "C:\WINDOWS\System32\MsPorts.Dll" (key "MsPorts.Dll") is signed in catalog
 "C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5.CAT".
 @ 20:14:33.873 #V146 Using exported function "PortsClassInstaller" in module
 "C:\WINDOWS\System32\MsPorts.Dll".
 @ 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 Port"; Driver:  
"Communications Port"; Provider: "Microsoft"; Mfg: "(Standard port types)"; Section name: "ComPort".  
(rank of 0 is absolute best match)  
@ 20:14:37.413 #I023 Actual install section: [ComPort.NT]. Rank: 0x00001000. Effective driver date:  
02/14/2001.  
@ 20:14:37.413 #I022 Found "*PNP0501" in C:\WINDOWS\inf\msports.inf; Device: "Communications Port"; Driver:  
"Communications Port"; Provider: "Microsoft"; Mfg: "(Standard port types)"; Section name: "ComPort".  
@ 20:14:37.413 #I023 Actual install section: [ComPort.NT]. Rank: 0x00000000. Effective driver date:  
02/14/2001.  
@ 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\msports.inf".  
@ 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.inf": "serial.sys"  
to "serial.sys" with flags 0x80000024, target directory is "C:\WINDOWS\System32\DRIVERS".  
@ 20:15:02.590 #V096 Source in section [sourcedisksfiles] in "c:\windows\inf\layout.inf"; Media=1  
Description="Windows XP Professional CD-ROM" Tag="\win51ip.b2" Path="\i386". Driver cache will be used.  
@ 20:15:02.605 #V132 File "C:\WINDOWS\INF\certclas.inf" (key "certclas.inf") is signed in catalog  
"C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5INF.CAT".  
@ 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.inf": "serenum.sys"  
to "serenum.sys" with flags 0x80000024, target directory is "C:\WINDOWS\System32\DRIVERS".  
@ 20:15:02.685 #V096 Source in section [sourcedisksfiles] in "c:\windows\inf\layout.inf"; Media=1  
Description="Windows XP Professional CD-ROM" Tag="\win51ip.b2" Path="\i386". Driver cache will be used.  
@ 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.inf".  
@ 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 catalog  
"C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5INF.CAT".  
@ 20:15:02.812 #V132 File "c:\windows\inf\layout.inf" (key "layout.inf") is signed in catalog  
"C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5INF.CAT".  
@ 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 signed in catalog  
"C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5.CAT".  
@ 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 signed in catalog
```

```
"C:\WINDOWS\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\NT5.CAT".
@ 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.inf".
@ 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: 0x0c: CM_PROB_NORMAL_CONFLICT.
@ 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_NEWDVICEWIZARD_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.
```

# Debugging Device Installations

12/5/2018 • 2 minutes to read • [Edit Online](#)

On Windows Vista and later versions of Windows, the core stages of device installation are always run in a non-interactive context known as *server-side installations*. The host process for device installation (*Drv\Inst.exe*) runs under the security context of the LocalSystem account.

Because the server-side installations run non-interactively and must complete without any user input, it provides some challenges to the driver package developer who wants to debug the actions of the [driver package's](#) class-installer and co-installer DLLs. For the developer of a driver package, it is usually most desirable to debug the actions of a co-installer DLL during the installation of a device.

This section contains the following topics, which describe techniques that are used to debug co-installers during the core stages of device installation:

[Enabling Support for Debugging Device Installations](#)

[Debugging Device Installations with a User-mode Debugger](#)

[Debugging Device Installations with the Kernel Debugger \(KD\)](#)

For more information about co-installers, see [Writing a Co-installer](#).

# Enabling Support for Debugging Device Installations

11/2/2020 • 2 minutes to read • [Edit Online](#)

Starting with Windows Vista, when the Plug and Play (PnP) manager detects a new device in the system, the operating system starts the device installation host process (*DrvInst.exe*) to search for and install a driver for the device.

To set the type of support the operating system provides for debugging the device installation host process, create (or modify) the following **REG\_DWORD** registry value on the target system to be debugged:

**HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Device  
Installer\DebugInstall**

The following table describes the types of debugging support that is specified by using the **DebugInstall** registry value.

DEBUGINSTALL VALUE	DEBUGGING SUPPORT
2	The device installation process will be debugged by using a user-mode debugger. For more information, see <a href="#">Debugging Device Installations with a User-mode Debugger</a> .
1	The device installation process will be debugged by using the kernel debugger (KD). For more information, see <a href="#">Debugging Device Installations with the Kernel Debugger (KD)</a> .
0	No debugging of the device installation process. This is the default support if <b>DebugInstall</b> is not present in the registry

After the **DebugInstall** registry value is set you do not need to reboot the target system that you want to debug. However, **DebugInstall** registry value must be set before the start of the next device installation and remains in effect for each subsequent device installation until the value is set to zero.

**Note** Be sure to reset the **DebugInstall** registry value to zero (or delete the value) as soon as it is no longer necessary to debug a device installation on the target system.

# Debugging Device Installations with a User-mode Debugger

11/2/2020 • 3 minutes to read • [Edit Online](#)

Starting with Windows Vista, when the Plug and Play (PnP) manager detects a new device in the system, the operating system starts the device installation host process (*DrvInst.exe*) to search for and install a driver for the device.

The most efficient way to debug the user-mode device installation host process is with a user-mode debugger, such as WinDbg or Visual Studio. Because the *DrvInst.exe* process would normally complete without any user interaction, Microsoft has added support to Windows Vista and later versions of Windows to allow the developer of a [driver package](#) to attach a debugger before the core stages of device installation are processed.

For more information about user-mode debuggers and other debugging tools, see [Windows Debugging](#).

The **DebugInstall** registry value specifies the type of device installation debugging support that is enabled on the system. For more information about this registry value, see [Enabling Support for Debugging Device Installations](#).

When the **DebugInstall** registry value is set to 2, *DrvInst.exe* will wait for a user-mode debugger to be attached to its process before it continues with the installation. After a debugger has been attached, the process will break into the debugger itself. A debugger should be attached and configured such that it will not initiate its own initial breakpoint in the target system that is being debugged.

For example, a debugger can be attached to *DrvInst.exe* by name:

```
C:\>C:\Debuggers\WinDbg.exe -g -pn DrvInst.exe
```

Or, if a debugger is attached to the target system, the following debug information will be displayed:

```
DRVINST.EXE: Waiting for debugger on Process ID = 3556 .....
```

This allows the debugger to be attached to the *DrvInst.exe* process by using its unique process ID:

```
C:\>C:\Debuggers\WinDbg.exe -g -p 3556
```

After a user-mode debugger is attached to the *DrvInst.exe* process, the process will break into the debugger:

```
Debugger detected!
DRVINST.EXE: Entering debugger during PnP device installation.
Device instance = "X\Y\Z" ...

(d48.5a0): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000000 ecx=00000000 edx=77f745c0 esi=00000000 edi=00000000
eip=77f24584 esp=0105ff74 ebp=0105ffa0 iopl=0 nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00000246
ntdll!DbgBreakPoint:
77f24584 cc          int     3

0:000> |
.  oid: d48attachname: E:\Windows\system32\DrvInst.exe
```

Because the core stages of device installation have not been processed, any class installer or co-installer DLLs that are used for the device are not yet loaded.

If the module and function name for a breakpoint are known in advance, that name can be set as an unresolved breakpoint by using the "bu" debugger command. The following code example shows how to set an unresolved breakpoint for the main entry point (CoInstallerProc) of the *MyCoinst.dll* co-installer:

```
0:000> bu mycoinst!CoInstallerProc  
0:000> bl  
0 eu          0001 (0001) (mycoinst!CoInstallerProc)
```

When *MyCoinst.dll* co-installer is loaded and the breakpoint is reached:

```
Breakpoint 0 hit  
eax=00000001 ebx=00000000 ecx=00000152 edx=00000151 esi=01a57298 edi=00000002  
eip=5bcf54f1 esp=0007e204 ebp=0007e580 iopl=0          nv up ei pl nz na pe nc  
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000          efl=00000202  
mycoinst!CoInstallerProc:  
5bcf54f1 8bff        mov edi,edi  
0:000> bl  
0 e 5bcf54f1    0001 (0001) 0:**** mycoinst!CoInstallerProc
```

A class installer or co-installer DLL should not predict when either, respectively, will be loaded or unloaded from the *DrvInst.exe* process. However, a breakpoint that is set by using "bu" will remain even if the module is unloaded.

Alternatively, the *DrvInst.exe* process might be allowed to execute up to the point where a specific class installer or co-installer DLL is loaded into the process by setting a debugger exception for the load event of that DLL:

```
0:000> sxe ld mycoinst.dll
```

0:000> g

After the module is loaded, breakpoints can be set within the DLL. For example:

```
ModLoad: 5bcf0000 5bd05000  C:\WINDOWS\system32\mycoinst.dll  
eax=00000000 ebx=00000000 ecx=011b0000 edx=7c90eb94 esi=00000000 edi=00000000  
eip=7c90eb94 esp=0007da54 ebp=0007db48 iopl=0          nv up ei ng nz ac po nc  
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000          efl=00000296  
ntdll!KiFastSystemCallRet:  
7c90eb94 c3          ret  
0:000> .reload mycoinst.dll  
0:000> x mycoinst!*InstallerProc*  
5bcf54f1 mycoinst!CoInstallerProc (unsigned int, void *, struct _SP_DEVINFO_DATA *)  
  
0:000> bu mycoinst!CoInstallerProc  
0:000> bl  
0 e 3b0649d5    0001 (0001) 0:**** mycoinst!CoInstallerProc  
0:000> sxd ld mycoinst.dll  
0:000> g  
Breakpoint 0 hit  
eax=00000001 ebx=00000000 ecx=000001d4 edx=000001d3 esi=000bbac0 edi=00000002  
eip=5bcf54f1 esp=0007e204 ebp=0007e580 iopl=0          nv up ei pl nz na pe nc  
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000          efl=00000202  
mycoinst!CoInstallerProc:  
5bcf54f1 8bff        mov edi,edi  
0:000>
```

Because the breakpoint was set as an unresolved breakpoint (bu), it will remain set even if the module is unloaded.

The default time period for an installation process to complete is 5 minutes. If the process does not complete within the given time period, the system assumes that the process has hung (stopped responding), and the installation process is terminated.

If a user-mode debugger is attached to the target system during the device installation process, the system will not enforce this timeout period. This allows a [driver package](#) developer to spend the time needed to debug the installation process.

# Debugging Device Installations with the Kernel Debugger (KD)

11/2/2020 • 4 minutes to read • [Edit Online](#)

Starting with Windows Vista, when the Plug and Play (PnP) manager detects a new device in the system, the operating system starts the device installation host process (*DrvInst.exe*) to search for and install a driver for the device.

Because device installation occurs within this user-mode process, it is usually easiest to use a user-mode debugger as described in [Debugging Device Installations with a User-mode Debugger](#). In some cases, however, it might be helpful to use the kernel debugger (KD) to monitor the user-mode device installation process.

For example, by using KD while debugging the user-mode device installation, you can do the following:

- Simultaneously debug a kernel-mode issue by using !devnode, !devobj, !drvobj, !irp, and other KD extensions.
- Monitor other user-mode processes without managing multiple debuggers by using the KD extensions !process or .process /p.

For more information about KD and other debugging tools, see [Windows Debugging](#).

The **DebugInstall** registry value specifies the type of device installation debugging support enabled on the system. For more information about this registry value, see [Enabling Support for Debugging Device Installations](#).

When the **DebugInstall** registry value is set to 1, *DrvInst.exe* will first check that the kernel debugger is both enabled and currently attached before it breaks into the debugger. After this break is made, breakpoints can be set in the user-mode modules of the current process. For example:

```
kd> .reload /user  
kd> bp /p @$proc setupapi!SetupDiCallClassInstaller
```

This sets a breakpoint on the routine `SETUPAPI!SetupDiCallClassInstaller` for the current process only.

For the developer of a [driver package](#), it is usually most desirable to debug the actions of a class installer or co-installer DLL during the installation of a device. However, when *DrvInst.exe* breaks into the debugger, any class installer or co-installer DLLs from the driver package will not have been loaded. Although the user-mode debuggers support the ability to set a debugger exception when a user-mode module is loaded into the process with the "sx e ld" command, the kernel debugger only supports kernel-mode modules with that command.

The following code example shows how a "Debugger Command Program" monitors the loading of a specific class installer or co-installer into the current process. In this example, the debugger command program will set a breakpoint on the main entry point (`CoInstallerProc`) of the *Mycoinst.dll* co-installer:

```
file: Z:\bpcoinst.txt  
  
r $t1 = 0  
!for_each_module .if ($spat("@#ModuleName", "mycoinst*") = 0) {r $t1 = 1}  
.if (not @$t1 = 0) {.echo mycoinst is loaded, set bp on mycoinst!CoInstallerProc } .else {.echo mycoinst not loaded}  
.if (not @$t1 = 0) {.reload mycoinst.dll}  
.if (not @$t1 = 0) {bp[0x20] /p @$proc mycoinst!CoInstallerProc } .else {bc[0x20]}
```

When executed, the debugger command program will check the list of modules loaded into the current process for *Mycoinst.dll*. After this co-installer DLL is loaded, the debugger will set a breakpoint (with a well-known breakpoint ID) on the *ColnstallerProc* entry point function.

Starting from the debug break initiated by the *DrvInst.exe* host process, you should first set a breakpoint on the return address of the call where *DrvInst.exe* broke into the kernel debugger. This breakpoint will clear all breakpoints set during the device installation and continue execution:

```
DRVINST.EXE: Entering debugger during PnP device installation.  
Device instance = "X\Y\Z" ...  
  
Break instruction exception - code 80000003 (first chance)  
010117b7 cc     int      3  
  
kd> bp[0x13] /p @$proc @$ra "bc *;g"
```

Next, you should set some breakpoints within the process to allow the commands in the debugger command program to execute at the appropriate time during device installation.

To make sure that the breakpoint for the class installer or co-installer DLL entry point is set before the function is invoked for device installation, the debugger command program should be executed anytime a new DLL is loaded into the current process, that is, after a call to *LoadLibraryExW* returns:

```
kd> .reload  
kd> bp[0x10] /p @$proc kernel32!LoadLibraryExW "gu;$$><Z:\bpcoinst.txt;g"
```

Rather than executing the program on every *LoadLibraryEx* call within the process (*bp[0x10]*), the developer can restrict it to execute only when class installer and co-installer DLLs are loaded into the process. Because **SetupDiCallClassInstaller** is the routine that invokes class installers and co-installers that are registered for a device, these DLLs will be loaded into the process during that call.

Because no assumptions should be made about when these DLLs will be unloaded from the *DrvInst.exe* host process, you must make sure the breakpoints can handle locating the DLL entry points during any calls that are made to **SetupDiCallClassInstaller** from the *DrvInst.exe* host process.

```
kd> bd[0x10]  
kd> bp[0x11] /p @$proc setupapi!SetupDiCallClassInstaller "be[0x10];bp[0x12] /p @$proc @$ra  
\"bd[0x10];bc[0x12];g\";g"  
kd> g
```

The breakpoint to execute the debugger command program (*bp[0x10]*) is initially disabled. It is enabled whenever **SetupDiCallClassInstaller** is invoked (*bp[0x11]*), and execution continues. The debugger command program (*bp[0x10]*) is again disabled when **SetupDiCallClassInstaller** returns by setting a breakpoint on the return address of that routine itself (*bp[0x12]*).

Be aware that the breakpoint that disables the debugger command program also clears itself and continues execution until **SetupDiCallClassInstaller** is called again or until the installation program completes and all breakpoints are cleared (*bp[0x13]*).

When execution begins after the above breakpoints are set, the process will break on each call to *mycoinst!ColnstallerProc*. This allows you to debug the execution of the class installer or co-installer DLL during core device installation.

The default time period for an installation process to complete is 5 minutes. If the process does not complete within the given time period, the system assumes that the process has stopped responding, and it is terminated.

The default timeout restriction placed on device installations is still in effect while the process is being debugged through the kernel debugger. Because execution of all programs on the system is stopped while broken into the debugger, the amount of time taken by the installation process is tracked the same as it would be on a system that is not being debugged.

# Overview of Early Launch AntiMalware

11/2/2020 • 2 minutes to read • [Edit Online](#)

This section provides information about developing Early Launch Antimalware (ELAM) drivers for Windows operating systems. It provides guidelines for antimalware developers to develop drivers that are initialized before other boot-start drivers, and that ensure that subsequent drivers do not contain malware. It assumes that the reader is familiar with developing kernel-mode drivers, specifically boot-start drivers.

This information applies to the following operating systems:

- Windows 8
- Windows 10
- Windows Server 2012

The following topics describe the interface requirements for Early Launch Antimalware (ELAM) drivers. They are intended to provide information about ELAM driver interfaces. The ELAM feature provides a Microsoft-supported mechanism for antimalware (AM) software to start before other third-party components. AM drivers are initialized first and allowed to control the initialization of subsequent boot drivers, potentially not initializing unknown boot drivers. Once the boot process has initialized boot drivers and access to persistent storage is available in an efficient way, existing AM software may continue to block malware from executing.

[ELAM Prerequisites](#)

[ELAM Driver Requirements](#)

## NOTE

Because an ELAM service runs as a PPL (Protected Process Light), you need to debug using a kernel debugger.

## See also

[Protecting Anti-Malware Services.](#)

# ELAM Prerequisites

11/2/2020 • 2 minutes to read • [Edit Online](#)

Early Launch Antimalware drivers must adhere to the following program requirements to be signed by WHQL and loaded by Windows.

## Antimalware Vendor Participation Requirements

Microsoft requires that Early Launch Antimalware vendors either be members of the [Microsoft Virus Initiative \(MVI\)](#). This membership ensures that the vendors are active antimalware community participants with a positive industry reputation. If you are not a member of the MVI program and believe you need use of ELAM, please reach out to [mvi@microsoft.com](mailto:mvi@microsoft.com) for additional information.

## Windows Hardware Quality Lab (WHQL) submission

- Submit your driver for verification as documented at [ELAM Driver Submission](#)
- The WHQL process will verify that the vendor is permitted to submit early launch drivers. Your submission will fail if you are not an MVI member, or a pre-approved VIA member.

# ELAM Driver Requirements

11/2/2020 • 8 minutes to read • [Edit Online](#)

Driver installation must use existing tools for online and offline installation, registering a driver through typical INF processing. For sample ELAM driver code, please see the following: <https://github.com/Microsoft/Windows-driver-samples/tree/master/security/elam>

## AM Driver Installation

To ensure driver install compatibility, an ELAM driver advertises itself as a boot-start driver similar to all other boot-start drivers. The INF sets the start type to SERVICE\_BOOT\_START (0), which indicates that the driver should be loaded by the boot loader and initialized during kernel initialization. An ELAM Driver advertises its group as "Early-Launch". The early launch behavior for drivers in this group will be implemented in Windows, as described in the next section.

The following is an example of the driver install section of an ELAM driver INF.

```
[SampleAV.Service]
DisplayName      = %SampleAVServiceName%
Description     = %SampleAVServiceDescription%
ServiceType     = 1          ; SERVICE_KERNEL_DRIVER
StartType       = 0          ; SERVICE_BOOT_START
ErrorControl    = 3          ; SERVICE_ERROR_CRITICAL
LoadOrderGroup = "Early-Launch"
```

Because an AM driver does not own any devices, it is necessary to install the AM driver as a Legacy so the driver is only added as a service into the registry. (If the AM driver is installed as a normal PNP driver, it will be added to the enum section of the registry and therefore will have a PDO reference, which will lead to unwanted behavior when unloading the driver.)

You also need to include a [SignatureAttributes Section](#) in the INF file for the ELAM driver.

## Backup Driver Installation

To provide a recovery mechanism in the event that the ELAM driver is inadvertently corrupted, the ELAM installer also installs a copy of the driver in a backup location. This will allow WinRE to retrieve the clean copy and recover the installation.

The installer reads the backup file location from the **BackupPath** key stored in

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\EarlyLaunch
```

The installer then places the backup copy in the folder specified in the regkey.

## AM Driver Initialization

The Windows boot loader, Winload, loads all boot-start drivers and their dependent DLLs into memory prior to handoff to the Windows kernel. The boot-start drivers represent the device drivers that need to be initialized before the disk stack has been initialized. These drivers include, among others, the disk stack and volume manager, and file system driver and bus drivers for the operating system device.

# AM Driver Callback Interface

The ELAM drivers use callbacks to provide the PnP manager with a description of every boot-start driver and dependent DLL, and it can classify every boot image as a known good binary, known bad binary, or an unknown binary.

The default operating system policy is not to initialize known bad drivers and DLLs. Policy can be configured and is measured by Winload as part of boot attestation.

PnP uses policy and the classification provided by the AM driver to decide whether to initialize each boot image.

## Registry Callbacks

The Early Launch drivers can use registry or boot driver callbacks to monitor and validate the configuration data used as input for each boot-start driver. The configuration data is stored in the System registry hive, which is loaded by Winload and is available at the time of boot driver initialization.

### NOTE

Any changes to the ELAM registry hive are discarded before the system boots. As a result, an ELAM driver should use standard Event Tracing for Windows (ETW) logging rather than writing to the registry.

These callbacks are valid through the lifetime of the ELAM driver and will be unregistered when the driver is unloaded. For more info, see:

- [CmRegisterCallbackEx](#)
- [CmRegisterCallback](#)
- [CmUnRegisterCallback](#)

## Boot Driver Callbacks

Use [IoRegisterBootDriverCallback](#) and [IoUnRegisterBootDriverCallback](#) to register and unregister a *BOOT\_DRIVER\_CALLBACK\_FUNCTION*.

This callback provides status updates from Windows to an ELAM driver, including when all boot-start drivers have been initialized and the callback facility is no longer functional.

### Callback Type

The [BDCB\\_CALLBACK\\_TYPE enumeration](#) describes two types of callbacks:

- Callbacks that provide status updates to an ELAM driver (*BdCbStatusUpdate*)
- Callbacks used by the AM driver to classify boot-start drivers and dependent DLLs before initializing their images (*BdCblInitializeImage*)

The two callback types have unique context structures that provide additional information specific to the callback.

The context structure for the status update callback contains a single enumerated type describing the Windows callout.

The context structure for the initialize image callback is more complex, containing hash and certificate information for each loaded image. The structure additionally contains a field that is an output parameter where the AM driver stores the classification type for the driver.

An error returned from a status update callback is treated as a fatal error and leads to a system bug check. This provides an ELAM driver the ability to indicate when a state is reached outside of AM policy. For example, if an AM runtime driver was not loaded and initialized, the Early Launch driver can fail the prepare-to-unload callback to prevent Windows from entering a state without an AM driver loaded.

An image is treated as unknown when an error is returned from the initialize image callback. Unknown drivers are

initialized or have their initialization skipped based on OS policy.

## Malware Signatures

The malware signature data is determined by the AM ISV, but should include, at a minimum, an approved list of driver hashes. The signature data is stored in the registry in a new "Early Launch Drivers" hive under HKLM that is loaded by Winload. Each AM driver has a unique key in which to store their signature binary large object (BLOB). The registry path and key has the format:

```
HKLM\ELAM\<VendorName>\
```

Within the key, the vendor is free to define and use any of the values. There are three defined binary blob values that are measured by Measured Boot, and the vendor may use each:

- Measured
- Policy
- Config

The ELAM hive is unloaded after its use by Early Launch Antimalware for performance. If a user mode service wants to update the signature data, it should mount the hive file from the file location \Windows\System32\config\ELAM. For example, you could generate a UUID, convert it to a string, and use that as a unique key into which to mount the hive. The storage and retrieval format of these data BLOBs is left up to the ISV, but the signature data must be signed so that the AM driver can verify the integrity of the data.

### Verifying Malware Signatures

The method for verifying the integrity of the malware signature data is left up to each AM ISV. The [CNG Cryptographic Primitive Functions](#) are available to assist in verifying digital signatures and certificates on the malware signature data.

### Malware Signature Failure

If the ELAM driver checks the integrity of the signature data, and that check fails, or if there is no signature data, the initialization of the ELAM driver still succeeds. In this case, for each boot driver the ELAM driver must return "unknown" for each initialization callback. Additionally, the ELAM driver should pass this information onto the runtime AM component once it has started.

## Unloading the AM Driver

When the callback StatusType is BdCbStatusPrepareForUnload, this is an indication to the AM driver that all boot drivers have been initialized and that the AM driver should prepare to be unloaded. Before unloading, the early launch AM driver needs to deregister its callbacks. Deregistration cannot happen during a callback; rather, it has to happen in the DriverUnload function, which a driver can specify during DriverEntry.

To maintain continuity in malware protection and to ensure proper handoff, the runtime AM engine should be started prior to the early launch AM driver being unloaded. This means that the runtime AM engine should be a boot driver that is verified by the early launch AM driver.

## Performance

The driver must meet the performance requirements defined in the following table:

Scenario(s)	Start Time	End Time	Upper Bound

Evaluate loaded boot critical driver before allowing it to initialize. This also includes status update callbacks.	Kernel calls back to antimalware driver to evaluate loaded boot critical driver.	Antimalware driver returns evaluation result.	0.5ms
Evaluate all loaded boot critical drivers	Kernel calls back to antimalware driver to evaluate the first loaded boot critical driver.	Antimalware driver returns evaluation result for last boot critical driver.	50 ms
Footprint (driver + configuration data in memory)	N/A	N/A	128kB

## Initializing Drivers

Once the boot drivers are evaluated by the ELAM driver, the Kernel uses the classification returned by ELAM to decide whether to initialize the driver. This decision is dictated by policy and is stored here in the registry at:

```
HKLM\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy
```

This can be configured through Group Policy on a domain-joined client. An antimalware solution may want to expose this functionality to the end user in nonmanaged scenarios. The following values are defined for DriverLoadPolicy:

```
PNP_INITIALIZE_DRIVERS_DEFAULT 0x0 (initializes known Good drivers only)
PNP_INITIALIZE_UNKNOWN_DRIVERS 0x1
PNP_INITIALIZE_BAD_CRITICAL_DRIVERS 0x3 (this is the default setting)
PNP_INITIALIZE_BAD_DRIVERS 0x7
```

## Boot Failures

If a boot driver is skipped due to the initialization policy, the Kernel continues to attempt to initialize the next boot driver in the list. This continues until either the drivers are all initialized, or the boot failed because a boot driver that was skipped was critical to the boot. If the crash occurs after the disk stack is started, then there is a crash dump, and it contains some information about the reason or the crash, to include information about missing drivers. This can be used in WinRE to determine the cause of the failure and to attempt to remediate.

## ELAM and Measured Boot

If the ELAM driver detects a policy violation (a rootkit, for example), it should immediately call [Tbsi\\_Revoke\\_Attestation](#) to invalidate the PCRs that indicated that the system was in a good state. The function returns an error if there is a problem with measured boot, for example no TPM on the system.

[Tbsi\\_Revoke\\_Attestation](#) is callable from kernel mode. It extends PCR[12] by an unspecified value and increments the event counter in the TPM. Both actions are necessary, so the trust is broken in all quotes that are created from here forward. As a result, the Measured Boot logs will not reflect the current state of the TPM for the remainder of the time that the TPM is powered up, and remote systems will not be able to form trust in the security state of the system.

# ELAM driver submission Process

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following steps can be used to submit an Early Launch Antimalware (ELAM) driver:

1. Ensure your driver adheres to the documented requirements for ELAM drivers. See [ELAM driver requirements](#) and [INF SignatureAttributes Section](#) for more information.
2. Validate your driver using the Hardware Logo Kit (HLK) and Hardware Certification Kit (HCK). If your driver will be used in Windows 8 as well as Windows 10, you need to run both versions of the kit. Include the results with your submission. See [HLK tools technical reference](#) for more information. For information about required HCK tests, see below.
3. Follow the kernel mode driver signing policy as stated in the [Driver signing policy](#) topic.
4. Submit the driver package for evaluation at the [Windows Hardware Dev Center](#)

Each driver .sys file must be code signed by Microsoft, using a special certificate indicating that it is an Early Launch AM Driver.

The AM driver must be a single binary (not import any other DLLs).

## Hardware Certification Kit Tests

Each driver targeting a pre-Windows 10 operating system must pass the following HCK tests, which are administered by the ISV:

### PERFORMANCE TEST

- CALLBACK LATENCY - Each early launch AM driver is required to return the driver verification callbacks from the kernel within .5ms. This time is measured from when the kernel issues the callback to the driver to the time the driver returns the callback.
- MEMORY ALLOCATION - Each early launch AM driver is required to limit its footprint in memory to 128 KB, for both the driver image as well as its configuration (signature) data.
- UNLOAD BLOCKING - Each early launch AM driver receives a synchronous callback after the last boot driver has been initialized, which indicates that the AM driver will be unloaded. The AM driver can use this as an indication that it needs to do "cleanup" and save any status information that can be used by the runtime AM driver. However, the early launch AM driver must return the callback for the driver to be unloaded and for boot to continue.
- SIGNATURE DATA TEST - Each early launch AM driver must get its malware signature data from a single, well-known location and no other. This allows measurement and protection of that data by Windows. This test ensures that each AM driver only reads its configuration data from the registry hive that is created for that driver.
- BACKUP DRIVER TEST - The early launch AM driver, upon installation, must also install a backup copy of the driver to the backup driver store. This requirement is to help with remediation in the case that the primary driver gets corrupted. This test ensures that for an installed early launch AM driver, there is a corresponding driver in the backup store.

# BUS1394\_CLASS\_GUID

11/2/2020 • 2 minutes to read • [Edit Online](#)

The BUS1394\_CLASS\_GUID device interface class is defined for [1394 bus devices](#).

ATTRIBUTE	SETTING
Identifier	BUS1394_CLASS_GUID
Class GUID	{6BDD1FC1-810F-11d0-BEC7-08002BE2092F}

## Remarks

Bus drivers for 1394 buses register instances of this device interface class to notify the operating system and applications of the presence of 1394 bus devices.

For information about 1394 buses, see [1394 bus devices](#).

The WDK samples include the [1394api sample](#) application. This application uses BUS1394\_CLASS\_GUID to register to be notified of the presence of instances of this device interface class.

For information about the device interface class for IEEE 1394 devices in the 61883 [device setup class](#) that support the IEC-61883 protocol, see [GUID\\_61883\\_CLASS](#).

## Requirements

Version	Available in Windows XP and later versions of Windows.
Header	1394.h (include 1394.h)

## See also

[GUID\\_61883\\_CLASS](#)

# CdChangerClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

CdChangerClassGuid is an obsolete identifier for the [device interface class](#) for CD-ROM changer devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_CDCHANGER](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_CDCHANGER instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_CDCHANGER](#)

# CdRomClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

CdRomClassGuid is an obsolete identifier for the device interface class for CD-ROM storage devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_CDROM](#) class identifier for new instances of this class.

## Remarks

The storage samples in the WDK include the [CDROM class driver](#) sample and the [Addfilter Storage Filter Tool](#). The CDROM class driver sample uses CdRomClassGuid to register instances of the GUID\_DEVINTERFACE\_CDROM device interface class. The sample Addfilter application uses CdRomClassGuid to enumerate instances of the GUID\_DEVINTERFACE\_CDROM device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_CDROM instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_CDROM](#)

# CM\_Add\_Range

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Create\_DevNode

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Create\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Create\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Delete\_Class\_Key\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Delete\_DevNode\_Key\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Delete\_Range

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Detect\_Resource\_Conflict

11/2/2020 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported in Windows. Use [CM\\_Query\\_Resource\\_Conflict\\_List](#) instead.

# CM\_Detect\_Resource\_Conflict\_Ex

11/2/2020 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported in Windows. Use [CM\\_Query\\_Resource\\_Conflict\\_List](#) instead.

# CM\_Disable\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Dup\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Enable\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Find\_Range

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_First\_Range

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Free\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Class\_Key\_Name

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Class\_Key\_Name\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Class\_Name

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Class\_Name\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Device\_Interface\_Alias\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Device\_Interface\_List\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Device\_Interface\_List\_Size\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_DevNode\_Custom\_Property

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_DevNode\_Custom\_Property\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_DevNode\_Registry\_Property\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Global\_State

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Global\_State\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Hardware\_Profile\_Info

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Get\_Hardware\_Profile\_Info\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Intersect\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Invert\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Merge\_Range\_List

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Move\_DevNode

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Move\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Next\_Range

1/11/2019 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is reserved for system use. You should not use this function in your *class installers*, *co-installers*, or *device installation applications*.

# CM\_Open\_Class\_Key\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Open\_DevNode\_Key\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Arbitrator\_Free\_Data

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Arbitrator\_Free\_Data\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Arbitrator\_Free\_Size

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Arbitrator\_Free\_Size\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Remove\_SubTree

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Query\_Remove\_SubTree\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Register\_Device\_Driver

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported.

# CM\_Register\_Device\_Driver\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported.

# CM\_Register\_Device\_Interface

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Register\_Device\_Interface\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Remove\_SubTree

11/2/2020 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported in Windows 2000 and later versions of Windows. Use [CM\\_Query\\_And\\_Remove\\_Subtree](#) instead.

# CM\_Remove\_SubTree\_Ex

11/2/2020 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported in Windows 2000 and later versions of Windows. Use [CM\\_Query\\_And\\_Remove\\_SubTree\\_Ex](#) instead.

# CM\_Run\_Detection

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported.

# CM\_Run\_Detection\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

This function is obsolete and no longer supported.

# CM\_Set\_DevNode\_Registry\_Property\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Set\_HW\_Prof

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Set\_HW\_Prof\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Set\_HW\_Prof\_Flags

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Set\_HW\_Prof\_Flags\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Setup\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Test\_Range\_Available

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Uninstall\_DevNode\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Unregister\_Device\_Interface

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_Unregister\_Device\_Interface\_Ex

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

# CM\_WaitNoPendingInstallEvents

11/2/2020 • 2 minutes to read • [Edit Online](#)

This function is reserved for system use.

**CM\_WaitNoPendingInstallEvents** is defined in *Cfgmgr32.h* as follows:

```
#define CM_WaitNoPendingInstallEvents CMP_WaitNoPendingInstallEvents
```

In other words, **CM\_WaitNoPendingInstallEvents** is simply another name for **CMP\_WaitNoPendingInstallEvents**. Since there is no actual function named **CM\_WaitNoPendingInstallEvents**, `GetProcAddress` will return `NULL`.

# Config.DriverKey Section of a TxtSetup.ohm File

11/2/2020 • 2 minutes to read • [Edit Online](#)

A **Config.DriverKey** section specifies values to be set in the registry for particular component options. Windows automatically creates the required values in the **Services\DriverKey** key. Use this section to specify additional keys to be created under **Services\DriverKey** and values under **Services\DriverKey** and **Services\DriverKey\subkey\_name**.

```
[Config.DriverKey]
value = subkey_name,value_name,value_type,value
...
```

## *subkey\_name*

Specifies the name of a key under the **Services\DriverKey** tree where Windows places the specified value. Windows creates the key if it does not exist.

If *subkey\_name* is the empty string (""), the value is placed under the **Services\DriverKey**.

The *subkey\_name* can specify more than one level of subkey, such as "subkey1\subkey2\subkey3".

## *value\_name*

Specifies the name of the value to be set.

## *value\_type*

Specifies the type of the registry entry. The *value\_type* can be one of the following:

### REG\_DWORD

One *value* is allowed; it must be a string containing up to eight hexadecimal digits.

For example:

```
value = parameters,NumberOfButtons,REG_DWORD,2
```

### REG\_SZ or REG\_EXPAND\_SZ

One *value* is allowed; it is interpreted as the null-terminated string to be stored.

For example:

```
value = parameters,Description,REG_SZ,"This is a text string"
```

### REG\_BINARY

One *value* is allowed; it is a string of hex digits, each pair of which is interpreted as a byte value.

For example (stores the byte stream 00,34,ec,4d,04,5a):

```
value = parameters,Data,REG_BINARY,0034ec4d045a
```

### REG\_MULTI\_SZ

Multiple *value* arguments are allowed; each is interpreted as a component of the MULTI\_SZ string.

For example:

```
value = parameters.Strings,REG_MULTI_SZ,String1,"String 2",string3
```

*value*

Specifies the value; its format depends on *value\_type*.

The following example shows a **Config.DriverKey** section:

```
; ...
[Config.OEMSCSI]
value = parameters\PnpInterface,5,REG_DWORD,1
; ...
```

# Defaults Section of a *TxtSetup.oem* File

12/5/2018 • 2 minutes to read • [Edit Online](#)

The **Defaults** section lists the default driver(s) for each hardware component supported by this file. Windows highlights the default selection when it presents a list of drivers to the user.

```
[Defaults]
component = ID
...
```

*component*

Specifies a hardware component supported by this file. The *component* must be one of the following system-defined values: **computer** or **scsi**.

*ID*

Specifies a string that identifies the default option. This string matches an ID specified in the corresponding *HwComponent* section.

If a *TxtSetup.oem* file fails to define a default driver for a supported component, Windows uses the first entry in the *HwComponent* section.

The following example shows a **Defaults** section (and the *HwComponent* section) for a *TxtSetup.oem* file that supports one component (**scsi**):

```
; ...
[Defaults]
SCSI = oemscsi

[SCSI]           ; HwComponent section
oemscsi = "OEM Fast SCSI Controller"
oemscsi2 = "OEM Fast SCSI Controller 2"

; ...
```

# DEFINE\_DEVPROPKEY

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the `DEFINE_DEVPROPKEY` macro creates a `DEVPROPKEY` structure that represents a device property key in the [unified device property model](#).

```
#ifdef INITGUID
#define DEFINE_DEVPROPKEY(name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8, pid) EXTERN_C const DEVPROPKEY
DECLSPEC_SELECTANY name = { { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } }, pid }
#else
#define DEFINE_DEVPROPKEY(name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8, pid) EXTERN_C const DEVPROPKEY name
#endif // INITGUID
```

## Members

*name*

The name of a `DEVPROPKEY` structure that represents a device property key.

*l*

An unsigned long-typed variable that supplies the value of the `data1` member of the `fmtid` member of the `DEVPROPKEY` structure.

*w1*

An unsigned short-type variable that supplies the value of the `data2` member of the `fmtid` member of the `DEVPROPKEY` structure.

*w2*

An unsigned short-type variable that supplies the value of the `data3` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b1*

A byte-typed variable that supplies the value of the `data4[0]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b2*

A byte-typed variable that supplies the value of the `data4[1]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b3*

A byte-typed variable that supplies the value of the `data4[2]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b4*

A byte-typed variable that supplies the value of the `data4[3]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b5*

A byte-typed variable that supplies the value of the `data4[4]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b6*

A byte-typed variable that supplies the value of the `data4[5]` member of the `fmtid` member of the `DEVPROPKEY` structure.

*b7*

A byte-typed variable that supplies the value of the **data4[6]** member of the **fmtid** member of the **DEVPROPKEY** structure.

*b8*

A byte-typed variable that supplies the value of the **data4[7]** member of the **fmtid** member of the **DEVPROPKEY** structure.

*pid*

A **DEVPROPID**-typed variable that supplies the value of the **pid** (property identifier) member of the **DEVPROPKEY** structure. The property identifier must be greater than or equal to two.

## Remarks

The **DEFINE\_DEVPROPKEY** structure is part of the [unified device property model](#).

The **DEFINE\_DEVPROPKEY** macro can be used to create a **DEVPROPKEY** structure that represents a custom device property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

## See also

[DEVPROPKEY](#)

# deleteBinaries XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **deleteBinaries** XML element is an empty element that sets the **deleteBinaries** flag to ON, which configures DPInst to delete binary files from a system that were copied to the system when a [driver package](#) was installed.

## Element Tag

```
<deleteBinaries>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a>
Child elements	None
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **deleteBinaries** flag is set to OFF. To set the **deleteBinaries** flag to ON, include a **deleteBinaries** element as a child element of a **dpinst** element or use the /d DPInst command-line switch.

The following code example demonstrates a **deleteBinaries** element.

```
<dpinst>
  ...
  <deleteBinaries/>
  ...
</dpinst>
```

**Note** Starting with Windows 7, the operating system ignores a setting of ON for the **deleteBinaries** XML element. The binary files, which were copied to a system when a [driver package](#) was installed, can no longer be deleted by using DPInst.

## See also

[dpinst](#)

# DEVPKEY\_Device\_Address

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Address device property represents the bus-specific address of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Address
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_ADDRESS
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_Address to the address of the device on its bus. For information about the interpretation of a device address, see the **DevicePropertyAddress** value of the *DeviceProperty* parameter of [IoGetDeviceProperty](#).

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Address.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Address property key. Instead, you can use the corresponding SPDRP\_ADDRESS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IoGetDeviceProperty](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_BaseContainerId

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_BaseContainerId device property represents the *GUID* value of the base container identifier (*ID*). The Windows Plug and Play (PnP) manager assigns this value to the device node (*devnode*).

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_BaseContainerId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_GUID</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_XXX identifier	SPDRP_BASE_CONTAINERID
Localized?	No

## Remarks

The PnP manager determines the container ID for a devnode by using one of the following methods:

- A bus driver provides a container ID.

When the PnP manager assigns a container ID to a devnode, it first checks whether the bus driver of the devnode can provide a container ID. Bus drivers provide a container ID through an [IRP\\_MN\\_QUERY\\_ID](#) query request with the **Parameters.QueryId.IdType** field set to **BusQueryContainerID**.

- The PnP manager generates a container ID by using the removable device capability.

If a bus driver cannot provide a container ID for a devnode that it is enumerating, the PnP manager uses the removable device capability to generate a container ID for all devnodes that are enumerated for the device. The bus driver reports this device capability in response to an [IRP\\_MN\\_QUERY\\_CAPABILITIES](#) request.

- The PnP manager generates a container ID by using an override of the removable device capability.

Although the override mechanism does not change the value of the removable device capability, it forces the PnP manager to use the override setting and not the value of the removable device capability when it generates container IDs for devices.

For more information about these methods, see [How Container IDs are Generated](#).

Regardless of how the container ID value is obtained, the PnP manager assigns the value to the DEVPKEY\_Device\_BaseContainerId property of the devnode.

The DEVPKEY\_Device\_BaseContainerId property can be used to force the grouping of a new devnode with other devnodes that exists in the system. This lets you use the new devnode as the parent (or *base*) container ID for other related devnodes. To do this, you must first obtain the DEVPKEY\_Device\_BaseContainerID GUID of the existing devnode. Then, you must return the container ID GUID of the new devnode in response to an [IRP\\_MN\\_QUERY\\_ID](#)

query request that has the `Parameters.QueryId.IdType` field set to `BusQueryContainerID`.

**Note** The value that is returned by a query of the `DEVPKEY_Device_BaseContainerId` or `DEVPKEY_Device_ContainerId` properties can be different for the same devnode.

**Note** Do not use the `DEVPKEY_Device_BaseContainerId` property to reconstruct device container groupings in the system. Use the `DEVPKEY_Device_ContainerId` property instead.

For more information about container IDs, see [Container IDs](#).

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	<code>Devpkey.h</code> (include <code>Devpkey.h</code> )

## See also

[Container IDs](#)

[DEVPKEY\\_Device\\_ContainerId](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_BusNumber

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_BusNumber device property represents the number that identifies the bus instance that a device instance is attached to.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_BusNumber
Property-data-type identifier	<b>DEVPROP_TYPE_INT32</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_BUSNUMBER
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_BusNumber to the value of the BusNumber member of the [PNP\\_BUS\\_INFORMATION](#) structure that a bus driver returns in response to an [IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#) request.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_BusNumber.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_BusNumber property key. Instead, you can use the corresponding SPDRP\_BUSNUMBER identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#)

[PNP\\_BUS\\_INFORMATION](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_BusRelations

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_BusRelations device property represents the **bus relations** for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_BusRelations
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_BusRelations.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_BusReportedDeviceDesc

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_BusReportedDeviceDesc device property represents a string value that was reported by the bus driver for the device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_BusReportedDeviceDesc
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_BusReportedDeviceDesc is set by Windows Plug and Play (PnP) with the string value that is reported by the bus driver for a device instance. The bus driver returns this value when queried with [IRP\\_MN\\_QUERY\\_DEVICE\\_TEXT](#).

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_BusReportedDeviceDesc.

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_BusTypeGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_BusTypeGuid device property represents the GUID that identifies the bus type of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_BusTypeGuid
Property-data-type identifier	<b>DEVPROP_TYPE_GUID</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_BUSTYPEGUID
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_BusTypeGuid to the value of the BusTypeGuid member of the [PNP\\_BUS\\_INFORMATION](#) structure that a bus driver returns in response to an [IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#) request.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_BusTypeGuid.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_BusTypeGuid property key. Instead, you can use the corresponding SPDRP\_BUSTYPEGUID identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#)

[PNP\\_BUS\\_INFORMATION](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Capabilities

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Capabilities device property represents the capabilities of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Capabilities
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_CAPABILITIES
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_Capabilities to the capability value that the device driver returns in response to an [IRP\\_MN\\_QUERY\\_CAPABILITIES](#) request for device capability information. The value of DEVPKEY\_Device\_Capabilities is a bitwise OR of the CM\_DEVCAP\_Xxx capability flags that are defined in Cfgmgr32.h. The device capabilities that these flags represent correspond to a subset of the members of the [DEVICE\\_CAPABILITIES](#) structure.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Capabilities.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Capabilities property key. Instead, you can use the corresponding SPDRP\_CAPABILITIES identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVICE\\_CAPABILITIES](#)

[IRP\\_MN\\_QUERY\\_CAPABILITIES](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Characteristics

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Characteristics device property represents the characteristics of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Characteristics
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_CHARACTERISTICS
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_Characteristics is a bitwise OR of the FILE\_Xxx file characteristic flags that are defined in Wdm.h and Ntddk.h. For more information about the device characteristic flags, see the *DeviceCharacteristics* parameter of [IoCreateDevice](#) and [Specifying Device Characteristics](#).

You can set the value of DEVPKEY\_Device\_Characteristics by using an [INF AddReg directive](#) that is included in the [INF DDInstall.HW section](#) that installs a device.

You can retrieve the value of DEVPKEY\_Device\_Characteristics by calling [SetupDiGetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Characteristics property key. Instead, you can use the corresponding SPDRP\_CHARACTERISTICS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall.HW Section](#)

[IoCreateDevice](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)



# DEVPKEY\_Device\_Children

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Children device property represents a list of the device instance IDs for the devices that are children of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Children
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Children.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Class

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Class device property represents the name of the [device setup class](#) that a device instance belongs to.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Class
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_CLASS
Localized?	No

## Remarks

The value of the DEVPKEY\_Device\_Class property is set by the *class-name* value that is supplied by the Class directive in the [INF Version section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Class.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Class property key. Instead, you can use the corresponding SPDRP\_CLASS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Version Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ClassGuid device property represents the GUID of the [device setup class](#) that a device instance belongs to.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ClassGuid
Property-data-type identifier	<b>DEVPROP_TYPE_GUID</b>
Property access	Read-only by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_CLASSGUID
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_ClassGuid is set by the INF ClassGUID directive that is supplied by the [INF Version section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_ClassGuid.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_ClassGuid property key. Instead, you can use the corresponding SPDRP\_CLASSGUID identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Version Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_CompatibleIds

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DEVICE\_CompatibleIds device property represents the list of compatible identifiers for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_CompatibleIds
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Internal data format	"compatible-id1\0compatible-id2\0...compatible-idn\0\0"
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxxidentifier	SPDRP_COMPATIBLEIDS
Localized?	No

## Remarks

The value of DEVPKEY\_DEVICE\_CompatibleIds is set by the *compatible-id* entry values that are supplied for a device in the [INF Models section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_DEVICE\_CompatibleIds.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_CompatibleIds property key. Instead, you can use the corresponding SPDRP\_COMPATIBLEIDS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Models Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ConfigFlags

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ConfigFlags device property represents the configuration flags that are set for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ConfigFlags
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_XXX identifier	SPDRP_CONFIGFLAGS
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_ConfigFlags is set during a device installation to indicate the current configuration of a device.

The configuration flags are for internal use only.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_ConfigFlags and call [SetupDiSetDeviceProperty](#) to set DEVPKEY\_Device\_ConfigFlags.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_ContainerId

12/1/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ContainerId device property is used by the Plug and Play (PnP) manager to group one or more device nodes (*devnodes*) into a *device container* that represents an instance of a physical device.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ContainerId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_GUID</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

Starting with Windows 7, the PnP manager uses the device container and its identifier (*ContainerID*) to group one or more *devnodes* that originated from and belong to each instance of a particular physical device. The ContainerID for a device instance is referenced through the DEVPKEY\_Device\_ContainerId device property.

When you group all the devnodes that originated from an instance of a single device into containers, you accomplish the following results:

- The operating system can determine how functionality is related among *devnodes* that originate from a physical device.
- The user or applications are presented with a device-centric view of devices instead of the traditional function-centric view.

The DEVPKEY\_Device\_ContainerId can be used to determine the device container grouping of *devnodes* in a system. For a given devnode, you can determine all the devnodes that belong to the same container by completing the following steps:

- Call [SetupDiGetDeviceProperty](#) to query DEVPKEY\_Device\_ContainerId for the given devnode. Windows returns the ContainerID *GUID* value for the device container to which that devnode belongs.
- Enumerate all devnodes on the computer and query each devnode for its DEVPKEY\_Device\_ContainerId. Each ContainerId value that matches the ContainerId value of the original devnode is part of same container.

**Note** All *devnodes* that belong to a container on a given bus type must share the same ContainerID value.

For more information about ContainerIDs, see [Container IDs](#).

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[Container IDs](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DeviceDesc

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DeviceDesc device property represents a description of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DeviceDesc
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_DEVICEDESC
Localized?	Yes

## Remarks

The value of DEVPKEY\_Device\_DeviceDesc is set by the *device-description* entry value that is supplied by the [INF Models section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_DEVICE\_DeviceDesc.

You can retrieve the value of the [DEVPKEY\\_NAME](#) device instance property to retrieve the name of the device as it should appear in a user interface item.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DeviceDesc property key. Instead, these earlier versions of Windows use the corresponding SPDRP\_DEVICEDESC identifier to access the value of the property. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_NAME \(Device Instance\)](#)

[INF Models Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DevNodeStatus

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DevNodeStatus device property represents the status of a device node (*devnode*).

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DevNodeStatus
Property-data-type identifier	DEVPROP_TYPE_INT32
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_DevNodeStatus is a bitwise OR of the DN\_Xxx bit flags that are defined in Cfg.h.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DevNodeStatus.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to access the status of a device instance on these earlier versions of Windows, see [Retrieving the Status and Problem Code for a Device Instance](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[CM\\_Get\\_DevNode\\_Status](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DevType

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DevType device property represents the device type of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DevType
Property-data-type identifier	DEVPROP_TYPE_INT32
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_DEVTYPE
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_DevType to the value of the DeviceType member of the [DEVICE\\_OBJECT](#) structure for a device instance. The value of DEVPKEY\_Device\_DevType is one of the system-defined device type values that are listed in [Specifying Device Types](#).

You can set the value of DEVPKEY\_Device\_DevType by using an [INF AddReg directive](#) that is included in the [INF DDInstall.HW section](#) in the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DevType.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DevType property key. Instead, you can use the corresponding SPDRP\_DEVTYPE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVICE\\_OBJECT](#)

[INF DDInstall.HW section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DHP\_Rebalance\_Policy

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DHP\_Rebalance\_Policy device property represents a value that indicates whether a device will participate in resource rebalancing following a [dynamic hardware partitioning \(DHP\)](#) processor hot-add operation.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DHP_Rebalance_Policy
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read and write access by applications and services.
Localized?	No

## Remarks

On a dynamically partitionable server that is running Windows Server 2008 or later versions of Windows Server, the operating system initiates a system-wide resource rebalance whenever a new processor is dynamically added to the system. The DEVPKEY\_Device\_DHP\_Rebalance\_Policy property determines whether a device participates in such a resource rebalance. The device participates in resource rebalancing under the following circumstances:

- The DEVPKEY\_Device\_DHP\_Rebalance\_Policy device property does not exist.
- The device property exists and the value of the device property is not set.
- The device property exists and the value of the device property is set to 2.

If the DEVPKEY\_Device\_DHP\_Rebalance\_Policy device property exists and the value of the property is set to 1, the device will not participate in resource rebalancing when a new processor is dynamically added to the system.

A device's [device setup class](#) is specified in the [INF Version Section](#) of the device's INF file.

The default behavior for devices in the Network Adapter (Class = Net) device setup class is that members of the class do not participate in resource rebalancing when a new processor is dynamically added to the system. The default behavior for devices in all other device setup classes is that members participate in resource rebalancing when a new processor is dynamically added to the system.

This device property does not affect whether a device will participate in a resource rebalance that is initiated for other reasons.

You can access the DEVPKEY\_Device\_DHP\_Rebalance\_Policy property by calling [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#).

## Requirements

Version	Available in Windows Server 2008 and later versions of Windows Server.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_DmaRemappingPolicy

11/2/2020 • 2 minutes to read • [Edit Online](#)

The value of the DEVPKEY\_Device\_DmaRemappingPolicy device property indicates the DMA remapping capability of the device.

**Property key:** DEVPKEY\_Device\_DmaRemappingPolicy

**Property-data-type identifier:** [DEVPROP\\_TYPE\\_INT32](#)

**Property access:** Read-only access by applications and services.

**Localized?:** No

## Remarks

VALUE	MEANING
2	Drivers on this device are capable of using DMA remapping.
1	At least one driver on this device opted out of DMA remapping.
0 or the DMA Remapping Policy property is not visible	A DMA remapping INF directive is not specified in the INF file. DMA remapping is not enforced for this device.

You can access the DEVPKEY\_Device\_DmaRemappingPolicy property by calling [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#).

## Requirements

**Version:** Available in Windows 10, version 1803 (Redstone 4)

**Header:** Devpkey.h (include Devpkey.h)

## See also

[Enabling DMA remapping for device drivers](#)

[Kernel DMA Protection](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_Driver

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Driver device property represents the registry entry name of the *driver key* for a *device instance*.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Driver
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_DRIVER
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_Driver after it installs a driver for device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Driver.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Driver property key. Instead, you can use the corresponding SPDRP\_DRIVER identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverCoInstallers

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverCoInstallers device property represents a list of DLL names, and entry points in the DLLs, that are registered as *co-installers* for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverCoInstallers
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Data format	"AbcCoInstall.dll,AbcCoInstallEntryPoint\0...AbcCoInstall.dll, AbcCoInstallEntryPoint\0\0"
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_COINSTALLERS_32 CoInstallers32
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_DriverCoInstallers is supplied by the [INF DDInstall.Coinstallers](#) section in the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverCoInstallers.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverCoInstallers property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding CoInstallers32 registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF DDInstall.Coinstallers Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverDate

11/2/2020 • 2 minutes to read • [Edit Online](#)

The PKEY\_Device\_DriverDate device property represents the date of the driver that is currently installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverDate
Property-data-type identifier	<a href="#">DEVPROP_TYPE_FILETIME</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_DRIVERDATEDATA DriverDateData
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_DriverDate is supplied by the [INF DriverVer directive](#) that is included in the INF Version section of an INF file that installs a device or by a device-specific INF DriverVer directive that is included in the [INF DDInstall section](#) that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverDate property.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverDate property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **DriverDateData** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF DDInstall Section](#)

[INF DriverVer Directive](#)

[INF Version section SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverDesc

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverDesc device property represents the description of the driver that is installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverDesc
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_DRVDESC DriverDesc
Localized?	Yes

## Remarks

The value of DEVPKEY\_Device\_DriverDesc is set by the *device-description* entry value that is supplied by the **INF Models section** of the INF file that installs a device.

The value of DEVPKEY\_Device\_DriverDesc is not displayed in an end-user dialog box or used for any reason by the operating system.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverDesc.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_LocationPaths property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **DriverDesc** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverInfPath

11/2/2020 • 2 minutes to read • [Edit Online](#)

The PKEY\_Device\_DriverInfPath device property represents the name of the INF file that installed a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverInfPath
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_INFPATH <b>InfPath</b>
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_DriverInfPath. A copy of the INF file that installed a device is located in the system INF file directory. The name of the INF file copy is OemNnn.inf, where Nnn is a decimal number from 0 through 9999.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverInfPath.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverInfPath property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **InfPath** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverInfSection

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverInfSection device property represents the name of the **INF DDInstall section** that installs the driver for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverInfSection
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_INFSECTION InfSection
Localized?	No

## Remarks

Windows sets the value of the PKEY\_Device\_DriverInfSection property.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of PKEY\_Device\_DriverInfSection.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the PKEY\_Device\_DriverInfSection property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **InfSection** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverInfSectionExt

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverInfSectionExt device driver property represents the platform extension of the [INF DDInstall section](#) that installs the driver for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverInfSectionExt
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_INFSECTIONEXT InfSectionExt
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_DriverInfSectionExt.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverInfSectionExt.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverInfSectionExt property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **InfSectionExt** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverLogoLevel

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverLogoLevel device property represents the Microsoft Windows Logo level for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverLogoLevel
Property-data-type identifier	<a href="#">DEVPROP_TYPE_UINT32</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_DriverLogoLevel.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverLogoLevel.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverPropPageProvider

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverPropPageProvider device property represents the name of a DLL, and an entry point in the DLL, that is registered as a property page provider for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverPropPageProvider
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_ENUMPROPPAGES_32 EnumPropPages32
Localized?	No

## Remarks

You can set the value of DEVPKEY\_Device\_DriverPropPageProvider by using an [INF AddReg directive](#) that is included the [INF DDInstall section](#) of the INF file that installs a device.

You can retrieve the value of DEVPKEY\_Device\_DriverPropPageProvider by calling [SetupDiGetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverPropPageProvider property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding [EnumPropPages32](#) registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverProvider

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverProvider device property represents the name of the provider of the [driver package](#) for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverProvider
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_PROVIDER_NAME ProviderName
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_DriverProvider is supplied by the **Provider** directive that is included in the [INF Version section](#) of a device INF file.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverProvider.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverProvider property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **ProviderName** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Version Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_DriverRank

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_DriverRank device property represents the rank of the driver that is installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverRank
Property-data-type identifier	<a href="#">DEVPROP_TYPE_UINT32</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_DriverRank.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_DriverRank.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverRank property key. For information about how to access this property on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

For information about driver rank, see [How Windows Ranks Drivers](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiGetDriverInstallParams](#)

[SP\\_DRVINSTALL\\_PARAMS](#)

# DEVPKEY\_Device\_DriverVersion

11/2/2020 • 2 minutes to read • [Edit Online](#)

The PKEY\_Device\_DriverVersion device property represents the version of the driver that is currently installed on a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_DriverVersion
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_DRIVERVERSION DriverVersion
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_DriverVersion is supplied by the [INF DriverVer directive](#) that is included in the [INF Version section](#) of an INF file that installs a device or is supplied by a device-specific INF DriverVer directive that is included in the [INF DDInstall section](#) that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of PKEY\_Device\_DriverVersion.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_DriverVersion property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **DriverVersion** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF DDInstall Section](#)

[INF DriverVer Directive](#)

[INF Version Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_EjectionRelations

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_EjectionRelations device property represents the **ejection relations** for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_EjectionRelations
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_EjectionRelations.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_EnumeratorName

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_EnumeratorName device property represents the name of the enumerator for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_EnumeratorName
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_ENUMERATOR_NAME
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_EnumeratorName to the name of the enumerator for a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_EnumeratorName.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_EnumeratorName property key. Instead, you can use the corresponding SPDRP\_ENUMERATOR\_NAME identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Exclusive

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Exclusive device property represents a Boolean value that determines whether a device instance can be opened for exclusive use.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Exclusive
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_EXCLUSIVE
Localized?	No

## Remarks

The value of the DEVPKEY\_Device\_Exclusive property is DEVPROP\_TRUE if the device can be opened for exclusive use. Otherwise, the value of the property is DEVPROP\_FALSE.

You can set the value of DEVPKEY\_Device\_Exclusive by using an [INF AddReg directive](#) that is included in the [INF DDInstall.HW section](#) that installs a device.

You can retrieve or set the value of DEVPKEY\_Device\_Exclusive by calling [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Exclusive property key. Instead, you can use the corresponding SPDRP\_EXCLUSIVE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall.HW Section](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_FirstInstallDate

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_FirstInstallDate device property specifies the time stamp when the device instance was first installed in the system.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_FirstInstallDate
Property-data-type identifier	<a href="#">DEVPROP_TYPE_FILETIME</a>
Property access	Read-only access by installation applications and installers.
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_FirstInstallDate with the time stamp that specifies when the device instance was first installed in the system.

**Note** Unlike the [DEVPKEY\\_Device\\_InstallDate](#) property, the value of the DEVPKEY\_Device\_FirstInstallDate property does not change with each successive update of the device driver. For example, a driver that was updated through Windows Update does not change the value of this property.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_FirstInstallDate property.

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_FriendlyName

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_FriendlyName device property represents the friendly name of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_FriendlyName
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_FRIENDLYNAME
Localized?	No

## Remarks

You can use the [DEVPKEY\\_NAME](#) device property instead of DEVPKEY\_Device\_FriendlyName to display the name that identifies a device instance in a user interface display.

You can set the value of DEVPKEY\_Device\_FriendlyName by using an [INF AddReg directive](#) that is included in the [INF DDInstall.HW section](#) in the INF file that installs a device.

You can retrieve the value of DEVPKEY\_Device\_FriendlyName by calling [SetupDiGetDeviceProperty](#) or you can set this property by calling [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_FriendlyName property key. Instead, you can use the corresponding SPDRP\_FRIENDLYNAME identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_NAME \(Device Instance\)](#)

[INF AddReg Directive](#)

[INF DDInstall.HW Section](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_GenericDriverInstalled

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_GenericDriverInstalled device property represents a Boolean value that indicates whether the driver installed for a device instance provides only basic device functionality.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_GenericDriverInstalled
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_GenericDriverInstalled.

The value of DEVPKEY\_Device\_GenericDriverInstalled is set to DEVPROP\_TRUE to indicate that a basic driver is installed. Otherwise, the value of the property is set to DEVPROP\_FALSE.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_GenericDriverInstalled.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_HardwareIds

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DEVICE\_HardwareIds device property represents the list of hardware identifiers for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_HardwareIds
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Data format	"hw-id1\0hw-id2\0...hw-idn\0\0"
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_HARDWAREID
Localized?	No

## Remarks

The value of DEVPKEY\_DEVICE\_HardwareIds is set by the *hw-id* entry values for a device that are supplied by the [INF Models section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_DEVICE\_HardwareIds.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DEVICE\_HardwareIds property key. Instead, you can use the corresponding SPDRP\_HARDWAREID identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Models Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_InstallDate

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_InstallDate device property specifies the time stamp when the device instance was last installed in the system.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_InstallDate
Property-data-type identifier	<b>DEVPROP_TYPE_FILETIME</b>
Property access	Read-only access by installation applications and installers.
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_InstallDate with the time stamp that specifies when the device instance was last installed in the system.

This time stamp value changes for each successive update of the device driver. For example, this time stamp reports the date and time when the device driver was last updated through Windows Update.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_FirstInstallDate property.

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_InstallState

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_InstallState device property represents the installation state of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_InstallState
Property-data-type identifier	<a href="#">DEVPROP_TYPE_UINT32</a>
Property access	Read-only by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_INSTALL_STATE
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_InstallState to one of the CM\_INSTALL\_STATE\_Xxx values that are defined in Cfgmgr32.h. The CM\_INSTALL\_STATE\_Xxx values correspond to the [DEVICE\\_INSTALL\\_STATE](#) enumeration values.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_InstallState.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_Device\_InstallState property key. Instead, you can use the corresponding SPDRP\_INSTALL\_STATE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_InstanceId

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_InstanceId device property represents the device instance identifier of a device.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_InstanceId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers.
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_InstanceId is set internally by Windows during the installation of a device instance.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_InstanceId for a device instance.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_InstanceId property key. For information about how to retrieve a device instance identifier on these earlier versions of Windows, see [Retrieving a Device Instance Identifier](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceInstanceId](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Legacy

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Legacy device property represents a Boolean value that indicates whether a device is a root-enumerated device that the Plug and Play (PnP) manager automatically created when the non-PnP driver for the device was loaded.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Legacy
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The PnP manager sets the value of DEVPKEY\_Device\_Reported to DEVPROP\_TRUE if the PnP manager automatically created the device as a root-enumerated device when the non-PnP driver for the device was loaded. Otherwise, the PnP manager sets the value of the property to DEVPROP\_FALSE.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Legacy.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_LegacyBusType

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_LegacyBusType device property represents the legacy bus number of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_LegacyBusType
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_LEGACYBUSTYPE
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_LegacyBusType to the value of the LegacyBusType member of the [PNP\\_BUS\\_INFORMATION](#) structure that a bus driver returns in response to an [IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#) request. The value of DEVPKEY\_Device\_LegacyBusType is one of [INTERFACE\\_TYPE](#) enumerator values that are defined in Wdm.h and Ntddk.h.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_LegacyBusType.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_LegacyBusType property key. Instead, you can use the corresponding SPDRP\_LEGACYBUSTYPE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INTERFACE\\_TYPE](#)

[IRP\\_MN\\_QUERY\\_BUS\\_INFORMATION](#)

[PNP\\_BUS\\_INFORMATION](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_LocationInfo

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_LocationInfo device property represents the bus-specific physical location of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_LocationInfo
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_LOCATION_INFORMATION
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_LocationInfo to the value that the bus driver returns for a device instance in response to an [IRP\\_MN\\_QUERY\\_DEVICE\\_TEXT](#) IRP.

You can call [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#) to retrieve and set the value of DEVPKEY\_Device\_LocationInfo.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_LocationInfo property key. Instead, you can use the corresponding SPDRP\_LOCATION\_INFORMATION identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IRP\\_MN\\_QUERY\\_DEVICE\\_TEXT](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_LocationPaths

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_LocationPaths device property represents the location of a device instance in the device tree.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_LocationPaths
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_LOCATION_PATHS
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_LocationPaths.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_LocationPaths.

Windows Server 2003 supports this property, but does not support the DEVPKEY\_Device\_LocationPaths property key. Instead, you can use the corresponding SPDRP\_LOCATION\_PATHS identifier to access the value of the property on Windows Server 2003. For information about how to access this property value on Windows Server 2003, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_LowerFilters

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_LowerFilters device property represents a list of the service names of the lower-level filter drivers that are installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_LowerFilters
Property-data-type identifier	<b>DEVPROP_TYPE_STRING_LIST</b>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_LOWERFILTERS
Localized?	No

## Remarks

The value of the DEVPKEY\_Device\_LowerFilters property is set when a lower-level device filter driver is installed for a device. For more information about how to install a device filter driver, see [Installing a Filter Driver](#).

You can call [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#) to retrieve and set the value of DEVPKEY\_Device\_LowerFilters.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_LowerFilters property key. Instead, you can use the corresponding SPDRP\_LOWERFILTERS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_Manufacturer

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DEVICE\_Manufacturer device property represents the name of the manufacturer of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Manufacturer
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_MFG
Localized?	No

## Remarks

The value of DEVPKEY\_DEVICE\_Manufacturer is set by the *manufacturer-identifier* entry value for a device that is supplied by the [INF Manufacturer section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_DEVICE\_Manufacturer.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Manufacturer property key. Instead, you can use the corresponding SPDRP\_MFG identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Manufacturer Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_MatchingDeviceId

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_MatchingDeviceId device property represents the [hardware ID](#) or [compatible ID](#) that Windows uses to install a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_MatchingDeviceId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_MATCHINGDEVICEID MatchingDeviceId
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_MatchingDeviceId. The hardware IDs and compatible IDs for a device are supplied by the *device-description* entries that are included in the [INF Models section](#) of the INF file that installs a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of PKEY\_Device\_MatchingDeviceId.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_MatchingDeviceId property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **MatchingDeviceId** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Models Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ModelId

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ModelId device property matches a device to a [device metadata package](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ModelId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_GUID</a>
Property access	Read and write access by installation applications and installers
Localized?	No

## Remarks

The DEVPKEY\_Device\_ModelId device property provides support for IHVs and OEMs to uniquely identify devices that share the same manufacturer and model. By using a model identifier (ModelID), OEMs and IHVs can match the device model that they distribute to their own branded device metadata package.

The DEVPKEY\_Device\_ModelId device property contains the value of the [ModelID](#) XML element from the device's metadata package. When the device is installed, this PKEY is populated with the ModelID GUID value as reported by the device.

For more information, see [Device Metadata Packages](#).

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[Device Metadata Packages](#)

[ModelID](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_NoConnectSound

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_NoConnectSound device property represents a Boolean value that indicates whether to suppress the sound that the Microsoft Windows operating system plays to indicate that a removable device arrived or was removed.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_NoConnectSound
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read and write access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_NoConnectSound is set to DEVPROP\_TRUE to suppress playing sound. Otherwise, the value of the property is set to DEVPROP\_FALSE.

The DEVPKEY\_Device\_NoConnectSound property is typically set by an [INF AddProperty directive](#) in the INF file for a device.

You can call [SetupDiGetDeviceProperty](#) or [SetupDiSetDeviceProperty](#) to retrieve or set the value of DEVPKEY\_Device\_NoConnectSound.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddProperty Directive](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Parent

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Parent device property represents the device instance identifier of the parent for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Parent
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Parent property.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_PDOName

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_PDOName device property represents the name of the physical device object (PDO) that represents a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_PDOName
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_PHYSICAL_DEVICE_OBJECT_NAME
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_PDOName to the name of the physical name object (PDO) that represents a device. For more information about PDO names, see the *DeviceName* parameter that is used with the [IoCreateDevice](#) routine.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_PDOName.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_PDOName property key. Instead, you can use the corresponding SPDRP\_PHYSICAL\_DEVICE\_OBJECT\_NAME identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IoCreateDevice](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_PhysicalDeviceLocation

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_PhysicalDeviceLocation device property encapsulates the physical device location information provided by a device's firmware to Windows.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_PhysicalDeviceLocation
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BINARY</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_PHYSICAL_DEVICE_LOCATION
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_PhysicalDeviceLocation with the physical device location information. The format of the information is defined in the ACPI 4.0a Specification, section 6.1.6.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_PhysicalDeviceLocation.

## Requirements

Version	Available starting with Windows 8
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_PowerData

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_PowerData device property represents power information about a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_PowerData
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BINARY</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_DEVICE_POWER_DATA
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_PowerData. The value of DEVPKEY\_Device\_PowerData contains a [CM\\_POWER\\_DATA](#) structure.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_PowerData.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_PowerData property key. Instead, you can use the corresponding SPDRP\_DEVICE\_POWER\_DATA identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[CM\\_POWER\\_DATA](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_PowerRelations

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_PowerRelations device property represents the [power relations](#) for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_PowerRelations
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_PowerRelations.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ProblemCode

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ProblemCode device property represents the problem code for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ProblemCode
Property-data-type identifier	DEVPROP_TYPE_INT32
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_ProblemCode is one of the CM\_PROB\_XXX problem codes that are defined in Cfg.h.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_ProblemCode.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to access the problem code for a device instance on these earlier versions of Windows, see [Retrieving the Status and Problem Code for a Device Instance](#).

For info on finding problem status in Device Manager or the kernel debugger, see [Retrieving the Status and Problem Code for a Device Instance](#).

For additional information that may help with the problem code, see [DEVPKEY\\_Device\\_ProblemStatus](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[CM\\_Get\\_DevNode\\_Status](#)

[SetupDiGetDeviceProperty](#)

[DEVPKEY\\_Device\\_ProblemStatus](#)

# DEVPKEY\_Device\_ProblemStatus

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ProblemStatus device property is an NTSTATUS value that is set when a problem code is generated. It provides more context on why the problem code was set. If no additional context is available, ProblemStatus shows as STATUS\_SUCCESS (0x00000000).

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ProblemStatus
Property-data-type identifier	<a href="#">DEVPROP_TYPE_NTSTATUS</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

For info on finding problem status in Device Manager or the kernel debugger, see [Retrieving the Status and Problem Code for a Device Instance](#).

For more info about NTSTATUS values, see [Using NTSTATUS Values](#).

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_ProblemStatus.

## Requirements

Version	Available in Windows 8 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[CM\\_Get\\_DevNode\\_Status](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_RemovalPolicy

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_RemovalPolicy device property represents the current removal policy for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_RemovalPolicy
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_REMOVAL_POLICY
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_RemovalPolicy to one of the CM\_REMOVAL\_POLICY\_Xxx values that are defined in Cfgmgr32.h.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_RemovalPolicy property.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_Device\_RemovalPolicy property key. Instead, you can use the corresponding SPDRP\_REMOVAL\_POLICY identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_RemovalPolicyDefault

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_RemovalPolicyDefault device property represents the default removal policy for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_RemovalPolicyDefault
Property-data-type identifier	<b>DEVPROP_TYPE_INT32</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_REMOVAL_POLICY_HW_DEFAULT
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_RemovalPolicyDefault to one of the CM\_REMOVAL\_POLICY\_Xxx values that are defined in Cfgmgr32.h.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_RemovalPolicyDefault.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_Device\_RemovalPolicyDefault property key. Instead, you can use the corresponding SPDRP\_REMOVAL\_POLICY\_HW\_DEFAULT identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_RemovalPolicyOverride

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_RemovalPolicyOverride device property represents the removal policy override for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_RemovalPolicyOverride
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_REMOVAL_POLICY_OVERRIDE
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_RemovalPolicyOverride is one of the CM\_REMOVAL\_POLICY\_Xxx values that are defined in Cfgmgr32.h.

You can retrieve the value of DEVPKEY\_Device\_RemovalPolicyOverride by calling [SetupDiGetDeviceProperty](#) or you can also set this value by calling [SetupDiSetDeviceProperty](#).

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_Device\_RemovalPolicyOverride property key. Instead, you can use the corresponding SPDRP\_REMOVAL\_POLICY\_OVERRIDE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_RemovalRelations

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_RemovalRelations device property represents the **removal relations** for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_RemovalRelations
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_RemovalRelations.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_Reported

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Reported device property represents a Boolean value that indicates whether a device instance is a root-enumerated device that the driver for the device reported to the Plug and Play (PnP) manager by calling [IoReportDetectedDevice](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Reported
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The PnP manager sets the value of DEVPKEY\_Device\_Reported to DEVPROP\_TRUE if the device is a root-enumerated device that the driver for the device reported to the PnP manager by calling [IoReportDetectedDevice](#). Otherwise, the PnP manager sets the value of the property to DEVPROP\_FALSE.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Reported.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IoReportDetectedDevice](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ResourcePickerExceptions

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ResourcePickerExceptions device property represents the resource conflicts that are allowed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ResourcePickerExceptions
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_RESOURCE_PICKER_EXCEPTIONS ResourcePickerExceptions
Localized?	No

## Remarks

You can set the value of DEVPKEY\_Device\_ResourcePickerExceptions by using an [INF AddReg directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs a device.

You can retrieve the value of DEVPKEY\_Device\_ResourcePickerExceptions by calling [SetupDiGetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_ResourcePickerExceptions property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **ResourcePickerExceptions** registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_ResourcePickerTags

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_ResourcePickerTags device property represents resource picker tags for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_ResourcePickerTags
Property-data-type identifier	DEVPROP_TYPE_STRING
Property access	Read-only access by installation applications and installers
Corresponding registry value identifier and registry value name	REGSTR_VAL_RESOURCE_PICKER_TAGS ResourcePickerTags
Localized?	No

## Remarks

You can set the value of DEVPKEY\_Device\_ResourcePickerTags by using an [INF AddReg directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs a device.

You can retrieve the value of PKEY\_Device\_ResourcePickerTags by calling [SetupDiGetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_ResourcePickerTags property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding ResourcePickerTags registry value under the software key for the device instance. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Driver Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_SafeRemovalRequired

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_SafeRemovalRequired device property represents a Boolean value that indicates whether a hot-plug device instance requires safe removal from the computer.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_SafeRemovalRequired
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

If this property for a hot-plug device instance has a value of DEVPROP\_TRUE, the device instance requires safe removal from the computer. In this case, Windows displays the **Safely Remove Hardware** icon in the notification area on the right side of the taskbar. When the user clicks this icon, the system starts the **Safely Remove Hardware** program. By using this program, the user can instruct the system to prepare the device instance for removal before it can be surprise-removed from the computer.

**Note** If the device instance is a removable media device, such as an optical disk drive, the device instance must have media inserted and must have the DEVPKEY\_Device\_SafeRemovalRequired property value of DEVPROP\_TRUE. If both are true, the device instance is displayed in the **Safely Remove Hardware** program.

Windows Plug and Play (PnP) determines that the hot-plug device instance requires safe removal from the system if the following are true:

- The device instance is currently connected to the system.
- The device instance is either started or can be ejected automatically by the system.
- The CM\_DEVCAP\_SURPRISEREMOVALOK device capability bit for the device instance is not set. For more information about device capabilities, see [SetupDiGetDeviceRegistryProperty](#).
- The device instance does not have the [DEVPKEY\\_Device\\_SafeRemovalRequiredOverride](#) device property set to DEVPROP\_FALSE.

**Note** PnP unconditionally determines that the hot-plug device requires safe removal if the DEVPKEY\_Device\_SafeRemovalRequiredOverride device property is set to DEVPROP\_TRUE.

- The device instance is either directly removable from its parent device instance or has a removable ancestor in its device tree.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_SafeRemovalRequired.

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_Device\\_SafeRemovalRequiredOverride](#)

[SetupDiGetDeviceProperty](#)

[SetupDiGetDeviceRegistryProperty](#)

# DEVPKEY\_Device\_SafeRemovalRequiredOverride

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_SafeRemovalRequiredOverride device property represents the safe removal override for the device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_SafeRemovalRequiredOverride
Property-data-type identifier	<b>DEVPROP_TYPE_BOOLEAN</b>
Property access	Read and write access by installation applications and installers
Localized?	No

## Remarks

This device property can be used to override the result of the heuristic that Windows Plug and Play (PnP) uses to calculate the value of the **DEVPKEY\_Device\_SafeRemovalRequired** device property. This override is performed as follows:

- If the DEVPKEY\_Device\_SafeRemovalRequiredOverride device property is set to DEVPROP\_TRUE and the device instance is removable or has a removable ancestor, PnP sets the DEVPKEY\_Device\_SafeRemovalRequired device property to DEVPROP\_TRUE and does not use the heuristic.

**Note** A device instance is considered removable if its removable device capability is set. For more information, see [Overview of the Removable Device Capability](#).

- If the DEVPKEY\_Device\_SafeRemovalRequiredOverride device property is set to DEVPROP\_TRUE and the device instance (or an ancestor) is not removable, PnP sets the DEVPKEY\_Device\_SafeRemovalRequired to DEVPROP\_FALSE and does not use the heuristic.
- If the DEVPKEY\_Device\_SafeRemovalRequiredOverride device property is either not set or set to DEVPROP\_FALSE, PnP sets the DEVPKEY\_Device\_SafeRemovalRequired device property to a value that is determined by using the heuristic.

You can retrieve the value of DEVPKEY\_Device\_SafeRemovalRequiredOverride by calling [SetupDiGetDeviceProperty](#). You can also set this value by calling [SetupDiSetDeviceProperty](#).

## Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_Security

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Security device property represents a security descriptor structure for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Security
Property-data-type identifier	<a href="#">DEVPROP_TYPE_SECURITY_DESCRIPTOR</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_SECURITY
Localized?	No

## Remarks

You can set the value of DEVPKEY\_Device\_Security by using an [INF AddReg directive](#) that is included in the [INF DDInstall.HW section](#) of the INF file that installs a device.

You can retrieve the value of DEVPKEY\_Device\_Security by calling [SetupDiGetDeviceProperty](#) or you can set this property by calling [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Security property key. Instead, you can use the corresponding SPDRP\_SECURITY identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall.HW Section](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_SecuritySDS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_SecuritySDS device property represents a security descriptor string for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_SecuritySDS
Property-data-type identifier	<a href="#">DEVPROP_TYPE_SECURITY_DESCRIPTOR_STRING</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_SECURITY_SDS
Localized?	No

## Remarks

You can retrieve the value of DEVPKEY\_Device\_SecuritySDS by calling [SetupDiGetDeviceProperty](#) or you can also set his value by calling [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_SecuritySDS property key. Instead, you can use the corresponding SPDRP\_SECURITY\_SDS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_Service

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Service device property represents the name of the service that is installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Service
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_SERVICE
Localized?	No

## Remarks

The value of DEVPKEY\_Device\_Service is set by the *service-name* entry value that is supplied by the [INF AddService directive](#) in the INF file that installs a service for a device.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Service.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_Service property key. Instead, you can use the corresponding SPDRP\_SERVICE identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddService Directive](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_SessionId

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_SessionId device property represents a value that indicates the Terminal Services sessions that a device instance can be accessed in.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_SessionId
Property-data-type identifier	<a href="#">DEVPROP_TYPE_UINT32</a>
Property access	Read and write access by applications and services.
Localized?	No

## Remarks

The Terminal Server feature supports Plug and Play (PnP) device redirection. Device redirection determines whether a device can be accessed by applications and services within all Terminal Services sessions or whether a device can be accessed only within a particular Terminal Services session. The accessibility of a device within a Terminal Services session is determined by the setting of DEVPKEY\_Device\_SessionId for a device, as follows:

- If the DEVPKEY\_Device\_SessionId property does not exist, or the property exists, but the value of the property is not set, the device can be accessed in all active Terminal Services sessions.
- If the DEVPKEY\_Device\_SessionId property exists and the value of the property is set to a nonzero Terminal Services session identifier, the device can be accessed only in the Terminal Services session indicated by the Terminal Services session identifier.
- If the DEVPKEY\_Device\_SessionId property exists and the value of the property is set to zero, the device can be accessed only by services. Session zero is a special session in which only services can run.

You can access the DEVPKEY\_Device\_SessionId property by calling [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_Siblings

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_Siblings device property represents a list of the device instance IDs for the devices that are siblings of a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_Siblings
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Not applicable

## Remarks

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_Siblings.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support this property. For information about how to retrieve device relations properties on these earlier versions of Windows, see [Retrieving Device Relations](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_UINumber

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_UINumber device property represents a number for the device instance that can be displayed in a user interface item.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_UINumber
Property-data-type identifier	<b>DEVPROP_TYPE_INT32</b>
Property access	Read-only access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_UI_NUMBER
Localized?	No

## Remarks

Windows sets the value of DEVPKEY\_Device\_UINumber to the value of the UINumber member of the [DEVICE\\_CAPABILITIES](#) structure for a device instance. The bus driver for a device instance returns this value in response to an [IRP\\_MN\\_QUERY\\_CAPABILITIES](#) request.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Device\_UINumber.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_UINumber property key. Instead, you can use the corresponding SPDRP\_UI\_NUMBER identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVICE\\_CAPABILITIES](#)

[IRP\\_MN\\_QUERY\\_CAPABILITIES](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_Device\_UINumberDescFormat

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_UINumberDescFormat device property represents a `printf`-compatible format string that you should use to display the value of the DEVPKEY\_DEVICE\_UINumber device property for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_UINumberDescFormat
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_UI_NUMBER_DESC_FORMAT
Localized?	No

## Remarks

You can retrieve the value of DDEVPKEY\_Device\_UINumberDescFormat by calling [SetupDiGetDeviceProperty](#) or you can also set this value by calling [SetupDiSetDeviceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_UINumberDescFormat property key. Instead, you can use the corresponding SPDRP\_UI\_NUMBER\_DESC\_FORMAT identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF DDInstall.HW Section](#)

[SetupDiGetDeviceProperty](#)

[SetupDiSetDeviceProperty](#)

# DEVPKEY\_Device\_UpperFilters

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Device\_UpperFilters device property represents a list of the service names of the upper-level filter drivers that are installed for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_Device_UpperFilters
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read and write access by installation applications and installers
Corresponding SPDRP_Xxx identifier	SPDRP_UPPERFILTERS
Localized?	No

## Remarks

The value of the DEVPKEY\_Device\_UpperFilters property is set when an upper-level device filter driver is installed for a device. For more information about how to install a device filter driver, see [Installing a Filter Driver](#).

You can call [SetupDiGetDeviceProperty](#) and [SetupDiSetDeviceProperty](#) to retrieve and set the value of DEVPKEY\_Device\_UpperFilters.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_Device\_UpperFilters property key. Instead, you can use the corresponding SPDRP\_UPPERFILTERS identifier to access the value of the property on these earlier versions of Windows. For information about how to access this property value on these earlier versions of Windows, see [Accessing Device Instance SPDRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DeviceClass\_Characteristics

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_Characteristics device property represents the default device characteristics of all devices in a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_Characteristics
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers after a device setup class is installed
Corresponding SPCRP_Xxx identifier	SPCRP_CHARACTERISTICS
Localized?	No

## Remarks

DEVPKEY\_DeviceClass\_Characteristics should only be set when a device setup class is installed and not modified later. For information about how to install a device setup class and setting this property, see [INF ClassInstall32 Section](#) and the information about the registry entry value **DeviceCharacteristics** that is provided in the "Special value-entry-name Keywords" section of [INF AddReg Directive](#).

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_Characteristics.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_DeviceClass\_Characteristics property key. On these earlier versions of Windows, you can use the SPCRP\_CHARACTERISTICS identifier to access the value of this property. For information about how to access the value of this property, see [Retrieving Device Setup Class SPCRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IoCreateDevice](#)

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

## [SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_ClassCoInstallers

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_ClassCoInstallers device property represents a list of the class co-installers that are installed for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_ClassCoInstallers
Property-data-type identifier	<b>DEVPROP_TYPE_STRING_LIST</b>
Data format	"coinstaller1.dll,coinstaller1-entry-point\0...coinstallerN.dll,coinstallerN-entry-point\0\0"
Property access	Read and write access by installation applications and installers
Corresponding registry value name	HLM\System\CurrentControlSet\Control\CoDeviceInstallers{ <i>device-setup-class-guid</i> }
Localized?	No

## Remarks

Each class installer in the class co-installer list is identified by its DLL and entry point.

For information about how to install a class co-installer, see [Registering a Class Co-installer](#).

You can retrieve the value of DEVPKEY\_DeviceClass\_ClassCoInstallers by calling [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#). You can set DEVPKEY\_DeviceClass\_ClassCoInstallers by calling [SetupDiSetClassProperty](#) or [SetupDiSetClassPropertyEx](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_ClassCoInstallers property key. For information about how to access the corresponding information on these earlier versions of Windows, see [Accessing the Co-installers Registry Entry Value of a Device Setup Class](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiSetClassProperty](#)

[SetupDiSetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_ClassInstaller

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_ClassInstaller device property represents the class installer for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_ClassInstaller
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Data format	" <i>class-installer.dll,class-entry-point</i> "
Property access	Read-only access by installation applications and installers
Corresponding registry value name	Installer32
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_ClassInstaller is the value of the **Installer32** registry value under the class registry key. This entry contains the name of the class installer DLL and the installer's entry point for the device setup class.

The **Installer32** registry value for a device setup class can be set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 Section](#) of the INF file that installs a device setup class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_ClassInstaller.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_ClassInstaller property key. You can access the value of this property by accessing the corresponding **Installer32** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

## [SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_ClassName

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_ClassName device property represents the class name of a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_ClassName
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_ClassName is set by the **Class** directive that is included in the [INF Version section](#) that installs the device setup class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_ClassName.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_ClassName property key. For information about how to access the name of a device setup class on Windows Server 2003, Windows XP, and Windows 2000, see [Accessing the Friendly Name and Class Name of a Device Setup Class](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF Version section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiClassNameFromGuid](#)

# DEVPKEY\_DeviceClass\_DefaultService

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_DefaultService device property represents the name of the default service for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_DefaultService
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	Default Service
Localized?	No

## Remarks

If a default service is installed for a device setup class and a device does not install a device-specific service, the [INF ClassInstall32.Services section](#) of the INF file that installs the class installs the class default service for the device.

The value of DEVPKEY\_DeviceClass\_DefaultService is the value of the **Default Service** registry value under the class registry key.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_DefaultService.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_DefaultService property key. You can access the value of this property by accessing the corresponding **Default Service** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF ClassInstall32.Services Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_DevType

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_DevType device property represents the default device type for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_DevType
Property-data-type identifier	<a href="#">DEVPROP_TYPE_UINT32</a>
Property access	Read-only access by installation applications and installers after a device setup class is installed
Corresponding SPCRP_Xxx identifier	SPCRP_DEVTYPE
Localized?	No

## Remarks

You can set the value of DEVPKEY\_DeviceClass\_DevType when an installation application installs a device setup class. For information about how to install a device setup class and setting this property, see [INF ClassInstall32 Section](#) and the information about the registry entry value **DeviceType** that is provided in the "Special *value-name* Keywords" section of [INF AddReg Directive](#).

The value of DEVPKEY\_DeviceClass\_DevType is one of the FILE\_DEVICE\_Xxx values that are defined in Wdm.h and Ntddk.h. For more information about device types, see the *DeviceType* parameter of the [IoCreateDevice](#) function.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_DevType.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_DeviceClass\_DevType property key. On these earlier versions of Windows, you can use the SPCRP\_DEVTYPE identifier to access the value of this property. For information about how to access the value of this property, see [Retrieving Device Setup Class SPCRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[IoCreateDevice](#)

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_DHPRebalanceOptOut

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_DHPRebalanceOptOut device property represents a value that indicates whether an entire device class will participate in resource rebalancing after a [dynamic hardware partitioning \(DHP\)](#) processor hot-add operation has occurred.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_DHPRebalanceOptOut
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read and write access by applications and services.
Localized?	No

## Remarks

On a dynamically partitionable server that is running Windows Server 2008 or later versions of Windows Server, the operating system initiates a system-wide resource rebalance whenever a new processor is dynamically added to the system. The device class participates in resource rebalancing under the following circumstances:

- The DEVPKEY\_DeviceClass\_DHPRebalanceOptOut device property does not exist.
- The device property exists and the value of the device property is not set.
- The device property exists and the value of the device property is set to **FALSE**.

If the DEVPKEY\_DeviceClass\_DHPRebalanceOptOut device property exists and the value of the property is set to **TRUE**, the device class does not participate in resource rebalancing when a new processor is dynamically added to the system.

A device's [device setup class](#) is specified in the [INF Version Section](#) of the device's INF file.

The default value for this property for the Network Adapter (Class = Net) is **TRUE**. The default value for this property for all other device setup classes is **FALSE**.

This device property does not affect whether a device class participates in a resource rebalance that is initiated for other reasons.

You can access the DEVPKEY\_DeviceClass\_DHPRebalanceOptOut property by calling [SetupDiGetClassProperty](#) and [SetupDiSetClassProperty](#).

## Requirements

Version	Available in Windows Server 2008 and later versions of Windows Server.
---------	--

Header

Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetClassProperty](#)

[SetupDiSetClassProperty](#)

# DEVPKEY\_DeviceClass\_Exclusive

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_Exclusive device property represents a Boolean flag that indicates whether devices that are members of a [device setup class](#) are exclusive devices.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_Exclusive
Property-data-type identifier	<b>DEVPROP_TYPE_BOOLEAN</b>
Property access	Read-only access by installation applications and installers after a device setup class is installed
Corresponding SPCRP_Xxx identifier	SPCRP_EXCLUSIVE
Localized?	No

## Remarks

You can set the value of DEVPKEY\_DeviceClass\_Exclusive when an installation application installs a device setup class. For information about how to install a device setup class and setting this property, see [INF ClassInstall32 Section](#) and the information about the registry value **Exclusive** that is provided in the "Special *value-entry-name* Keywords" section of [INF AddReg Directive](#).

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_Exclusive.

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_DeviceClass\_Exclusive property key. On these earlier versions of Windows, you can use the SPCRP\_EXCLUSIVE identifier to access the value of this property. For information about how to access the value of this property, see [Retrieving Device Setup Class SPCRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_Icon

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_Icon device property represents the icon for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_Icon
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_Icon is set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) that installs the class. To set the value of DEVPKEY\_DeviceClass\_Icon, use an AddReg directive to set the **Icon** registry entry value for the class.

The **Icon** entry value is an integer number in string format. If the number is negative, the absolute value of the number is the resource identifier of the icon in setupapi.dll. If the number is positive, the number is the resource identifier of the icon in the class installer DLL, if there is a class installer, or the class property page provider, if there is no class installer and there is a property page provider. A value of zero is not valid.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_Icon.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_Icon property key. For information about how to access the mini-icon for a device setup class on Windows Server 2003, Windows XP, and Windows 2000, see [Accessing Icon Properties of a Device Setup Class](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiDrawMinilcon](#)

**SetupDiLoadClassIcon**

# DEVPKEY\_DeviceClass\_IconPath

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_IconPath device property represents an icon list for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_IconPath
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	IconPath
Localized?	No

## Remarks

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_IconPath.

A DEVPKEY\_DeviceClass\_IconPath value is a [REG\\_MULTI\\_SZ](#)-typed list of icon resource specifiers in the format that is used by the Windows shell. The format of an icon resource specifier is "*executable-file-path,resource-identifier*," where *executable-file-path* contains the fully qualified path of the file on a computer that contains the icon resource and *resource-identifier* specifies an integer that identifies the resource. For example, the icon resource specifier "%SystemRoot%\system32\DLL1.dll,-12" contains the executable file path "%SystemRoot%\system32\DLL1.dll" and the resource identifier "-12".

Windows Server 2003, Windows XP, and Windows 2000 do not support this property. For information about how to access icon information for a device setup class on these versions of Windows, see [Accessing Icon Properties of a Device Setup Class](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiLoadClassIcon](#)

# DEVPKEY\_DeviceClass\_LowerFilters

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_LowerFilters device property represents a list of the service names of the lower-level filter drivers that are installed for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_LowerFilters
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Data format	"service-name1\0service-name2\0...service-nameN\0\0"
Property access	Read-only access by installation applications and installers after a class filter is installed
Corresponding SPCRP_Xxx identifier	SPCRP_LOWERFILTERS
Corresponding registry value name	LowerFilters
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_LowerFilters is set when a class filter driver is installed. For more information about how to install a class filter driver, see [Installing a Filter Driver](#) and [INF ClassInstall32 Section](#).

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_LowerFilters.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_LowerFilters property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **LowerFilters** registry value under the class registry key. For information about how to access this property value on these earlier versions of Windows, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiOpenClassRegKeyEx](#)

# DEVPKEY\_DeviceClass\_Name

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_Name device property represents the friendly name of a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_Name
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_Name is set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) that installs the class. To set the friendly name for a class, use an [AddReg directive](#) to set the [\(Default\)](#) registry entry value for the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_Name.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_Name property key. For information about how to access the friendly name of a device setup class on Windows Server 2003, Windows XP, and Windows 2000, see [Accessing the Friendly Name and Class Name of a Device Setup Class](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiGetClassDescriptionEx](#)

# DEVPKEY\_DeviceClass\_NoDisplayClass

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_NoDisplayClass device property represents a Boolean flag that controls whether devices in a [device setup class](#) are displayed by the Device Manager.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_NoDisplayClass
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	NoDisplayClass
Localized?	No

## Remarks

If the value of DEVPKEY\_DeviceClass\_NoDisplayClass is set to DEVPROP\_TRUE, Device Manager does not display devices in the device setup class. If this value is not set to DEVPROP\_TRUE, the Device Manager does display devices in the device setup class.

The **NoDisplayClass** registry value for a device setup class can be set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) of the INF file that installs the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_NoDisplayClass.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_NoDisplayClass property key. You can access the value of this property by accessing the corresponding **NoDisplayClass** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_NoInstallClass

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_NoInstallClass device setup class property represents a Boolean flag that controls whether devices in a [device setup class](#) are displayed in the **Add Hardware Wizard**.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_NoInstallClass
Property-data-type identifier	<b>DEVPROP_TYPE_BOOLEAN</b>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	NoInstallClass
Localized?	No

## Remarks

Setting the value of DEVPKEY\_DeviceClass\_NoInstallClass to DEVPROP\_TRUE indicates that installation of devices in the class does not require end-user interaction. If this value is not set to DEVPROP\_TRUE, the Add Hardware Wizard does display devices for the device setup class.

The **NoInstallClass** registry value for a device setup class can be set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) of the INF file that installs the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_NoInstallClass.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_NoInstallClass property key. You can access the value of this property by accessing the corresponding **NoInstallClass** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_NoUseClass

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_NoUseClass device property represents a Boolean flag that controls whether the Plug and Play (PnP) manager and SetupAPI use the [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_NoUseClass
Property-data-type identifier	<b>DEVPROP_TYPE_BOOLEAN</b>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	NoUseClass
Localized?	No

## Remarks

If the value of DEVPKEY\_DeviceClass\_NoUseClass is set to 1, the PnP manager and SetupAPI do not use the device setup class. Otherwise, they do use the device setup class.

The **NoUseClass** registry value for a device setup class is set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) of the INF file that installs the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_NoUseClass.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_NoUseClass property key. You can access the value of this property by accessing the corresponding **NoUseClass** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_PropPageProvider

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_PropPageProvider device property represents the property page provider for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_PropPageProvider
Property-data-type identifier	<b>DEVPROP_TYPE_STRING</b>
Data format	"prop-provider.dll,provider-entry-point"
Property access	Read-only access by installation applications and installers
Corresponding registry value name	<b>EnumPropPages32</b>
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_PropPageProvider is the value of the **EnumPropPages32** registry value under the class registry key. This value contains the name of the class property page provider DLL and the provider's entry point for the device setup class.

The **EnumPropPages32** registry value for a device setup class can set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) of the INF file that installs the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_PropPageProvider.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_PropPageProvider property key. You can access the value of this property by accessing the corresponding **EnumPropPages32** registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_Security

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_Security device property represents a security descriptor structure for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_Security
Property-data-type identifier	<a href="#">DEVPROP_TYPE_SECURITY_DESCRIPTOR</a>
Property access	Read and write access by installation applications and installers
Corresponding SPCRP_Xxx identifier	SPCRP_SECURITY
Localized?	No

## Remarks

You can set the value of DEVPKEY\_DeviceClass\_Security either during or after an installation application installs a device setup class. For more information about how to set this property, see [Creating Secure Device Installations](#).

You can retrieve the value of DEVPKEY\_DeviceClass\_Security by calling [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#). You can set DEVPKEY\_DeviceClass\_Security by calling [SetupDiSetClassProperty](#) or [SetupDiSetClassPropertyEx](#).

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_DeviceClass\_Security property key. On these earlier versions of Windows, you can use the SPCRP\_SECURITY identifier to access the value of this property. For information about how to access the value of this property, see [Retrieving Device Setup Class SPCRP\\_Xxx Properties](#) and [Setting Device Setup Class SPCRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiSetClassProperty](#)

[SetupDiSetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_SecuritySDS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_SecuritySDS device property represents a security descriptor string for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_SecuritySDS
Property-data-type identifier	<b>DEVPROP_TYPE_SECURITY_DESCRIPTOR_STRING</b>
Property access	Read and write access by installation applications and installers
Corresponding SPCRP_Xxx identifier	SPCRP_SECURITY_SDS
Localized?	No

## Remarks

You can set the value of DEVPKEY\_DeviceClass\_SecuritySDS either during or after an installation application installs a device setup class. For more information about how to set this property, see [Creating Secure Device Installations](#).

You can retrieve the value of DEVPKEY\_DeviceClass\_SecuritySDS by calling [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#). You can set DEVPKEY\_DeviceClass\_SecuritySDS by calling [SetupDiSetClassProperty](#) or [SetupDiSetClassPropertyEx](#).

Windows Server 2003 and Windows XP support this property, but do not support the DEVPKEY\_DeviceClass\_SecuritySDS property key. On these earlier versions of Windows, you can use the SPCRP\_SECURITY\_SDS identifier to access the value of this property. For information about how to access the value of this property, see [Retrieving Device Setup Class SPCRP\\_Xxx Properties](#) and [Setting Device Setup Class SPCRP\\_Xxx Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiSetClassProperty](#)

[SetupDiSetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_SilentInstall

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_SilentInstall device property represents a Boolean flag that controls whether devices in a [device setup class](#) should be installed, if possible, without displaying any user interface items.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_SilentInstall
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only access by installation applications and installers
Corresponding registry value name	SilentInstall
Localized?	No

## Remarks

If the value of DEVPKEY\_DeviceClass\_SilentInstall is set to DEVPROP\_TRUE, Windows installs a driver for a device without displaying any user interface items if the driver is already preinstalled in the driver store. Otherwise, Windows does not suppress the display of user interface items.

The [SilentInstall](#) registry value for a device setup class can be set by an [INF AddReg directive](#) that is included in the [INF ClassInstall32 section](#) of the INF file that installs the class.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_SilentInstall.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_SilentInstall property key. You can access the value of this property by accessing the corresponding [SilentInstall](#) registry value under the class registry key. For information about how to access value entries under the class registry key, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddReg Directive](#)

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

# DEVPKEY\_DeviceClass\_UpperFilters

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceClass\_UpperFilters device property represents a list of the service names of the upper-level filter drivers that are installed for a [device setup class](#).

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceClass_UpperFilters
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Data format	"service-name1\0service-name2\0...service-nameN\0\0"
Property access	Read-only access by installation applications and installers after a class filter is installed
Corresponding SPCRP_Xxx identifier	SPCRP_UPPERFILTERS
Corresponding registry value name	UpperFilters
Localized?	No

## Remarks

The value of DEVPKEY\_DeviceClass\_UpperFilters is set when a class filter driver is installed. For more information about how to install a class filter driver, see [Installing a Filter Driver](#) and [INF ClassInstall32 Section](#).

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_DeviceClass\_UpperFilters.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceClass\_UpperFilters property key. On these earlier versions of Windows, you can access the value of this property by accessing the corresponding **UpperFilters** registry value under the class registry key. For information about how to access this property value on these earlier versions of Windows, see [Accessing Registry Entry Values Under the Class Registry Key](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF ClassInstall32 Section](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiOpenClassRegKeyEx](#)

# DEVPKEY\_DeviceDisplay\_Category

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceDisplay\_Category device property represents one or more functional categories that apply to a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceDisplay_Category
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers.
Localized?	No

## Remarks

Device categories for a physical device are specified through the [DeviceCategory](#) XML element in a [device metadata package](#). Each instance of that device in a system inherits the device categories for that physical device.

Each physical device can have one or more functional categories specified in the [device metadata package](#). Each category is used by Windows Devices and Printers to group the device instance into one of the recognized device categories.

Multifunction devices would typically identify multiple functional categories for each hardware function that the device supports. For example, a multifunction device could identify functional categories for printer, fax, scanner, and removable storage device functionality.

The first functional category string in the [DEVPROP\\_TYPE\\_STRING\\_LIST](#) specifies the physical device's primary functional category. The primary functional category is defined by the independent hardware vendor (IHV) to specify how the device is advertised, packaged, sold, and ultimately identified by users.

If the DEVPKEY\_DeviceDisplay\_Category device property specifies more than one functional category string, the remaining strings that follow the first string specifies the physical device's secondary functional categories.

The [Devices and Printers](#) user interface in Control Panel displays the primary and secondary functional categories of the device instance. These categories are displayed in the order that is specified in the DEVPKEY\_DeviceDisplay\_Category device property.

You can access the DEVPKEY\_DeviceDisplay\_Category property by calling [SetupDiGetDeviceProperty](#).

## Requirements

Version	Available in Windows 7 and later versions of Windows.
---------	---

Header

Devpkey.h (include Devpkey.h)

## See also

[DeviceCategory](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DeviceInterface\_ClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceInterface\_ClassGuid device property represents the GUID that identifies a device interface class.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceInterface_ClassGuid
Property-data-type identifier	<b>DEVPROP_TYPE_GUID</b>
Property access	Read-only by installation applications and installers
Localized?	No

## Remarks

The format of *{device-interface-class}* key value is "`{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnnn}`", where each *n* is a hexadecimal digit.

You can call [SetupDiGetDeviceInterfaceProperty](#) to retrieve the value of DEVPKEY\_DeviceInterface\_ClassGuid.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceInterface\_ClassGuid property key. For information about how to retrieve the class GUID of a device interface on these earlier versions of Windows, see the information about how to use [SetupDiEnumDeviceInterfaces](#) that is provided in [Accessing Device Interface Properties](#).

For information about how to install and accessing device interfaces, see [Device Interface Classes](#) and the [INF AddInterface Directive](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddInterface Directive](#)

[SetupDiEnumDeviceInterfaces](#)

[SetupDiGetDeviceInterfaceProperty](#)

# DEVPKEY\_DeviceInterfaceClass\_DefaultInterface

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceInterfaceClass\_DefaultInterface device property represents the symbolic link name of the default device interface for a device interface class.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceInterfaceClass_DefaultInterface
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read and write access by installation applications and installers
Localized?	No

## Remarks

For information about how to install and using device interfaces, see [Device Interface Classes](#) and the [INF AddInterface Directive](#).

You can retrieve the value of DEVPKEY\_DeviceInterfaceClass\_DefaultInterface by calling [SetupDiGetDeviceInterfaceProperty](#). You can set DEVPKEY\_DeviceInterfaceClass\_DefaultInterface by calling [SetupDiSetDeviceInterfaceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceInterfaceClass\_DefaultInterface property key. For information about how to access the default interface of a device interface class on these earlier versions of Windows, see [Accessing Device Interface Class Properties](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddInterface Directive](#)

[SetupDiGetClassDevs](#)

[SetupDiGetDeviceInterfaceProperty](#)

[SetupDiSetDeviceInterfaceProperty](#)

# DEVPKEY\_DeviceInterface\_Enabled

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **DeviceInterfaceEnabled** device property represents a Boolean flag that indicates whether a device interface is enabled.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceInterface_Enabled
Property-data-type identifier	<a href="#">DEVPROP_TYPE_BOOLEAN</a>
Property access	Read-only by installation applications and installers
Localized?	No

## Remarks

If the value of DEVPKEY\_DeviceInterface\_Enabled is DEVPROP\_TRUE, the interface is enabled. Otherwise, the interface is not enabled.

You can call [SetupDiGetDeviceInterfaceProperty](#) to retrieve the value of DEVPKEY\_DeviceInterface\_Enabled.

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceInterface\_Enabled property key. For information about how to retrieve the activity status of a device interface on these earlier versions of Windows, see the information about how to use [SetupDiEnumDeviceInterfaces](#) that is provided in [Accessing Device Interface Properties](#).

For more information about device interfaces, see [Device Interface Classes](#) and the [INF AddInterface Directive](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[INF AddInterface Directive](#)

[SetupDiEnumDeviceInterfaces](#)

[SetupDiGetDeviceInterfaceProperty](#)

# DEVPKEY\_DeviceInterface\_FriendlyName

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceInterface\_FriendlyName device property represents the friendly name of a device interface.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceInterface_FriendlyName
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read and write access by installation applications and installers
Corresponding registry value name	FriendlyName
Localized?	Yes

## Remarks

The **FriendlyName** registry value for a device interface class is set by an [INF AddInterface directive](#) that is included in the [INF DDInstall.Interface section](#) of the INF file that installs a device interface.

Windows sets the value of the [DEVPKEY\\_NAME](#) device property for an interface to the value of DEVPKEY\_DeviceInterface\_FriendlyName. To identify a device interface in a user interface item, use the value of DEVPKEY\_NAME for the device interface instead of the value of DEVPKEY\_DeviceInterface\_FriendlyName.

You can retrieve the value of the DEVPKEY\_DeviceInterface\_FriendlyName by calling [SetupDiGetDeviceInterfaceProperty](#) and set it by calling [SetupDiSetDeviceInterfaceProperty](#).

Windows Server 2003, Windows XP, and Windows 2000 support this property, but do not support the DEVPKEY\_DeviceInterface\_FriendlyName property key. You can access the value of this property by accessing the corresponding **FriendlyName** registry entry value for the device interface. For information about how to access a registry entry value for a device interface, see [Accessing Device Interface Properties](#).

For information about device interfaces, see [Device Interface Classes](#) and the [INF AddInterface Directive](#).

## Requirements

**Version:** Windows Vista and later versions of Windows **Header:** Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_NAME \(Device Interface\)](#)

[INF AddInterface Directive](#)

[INF DDInstall.Interface Section](#)

[SetupDiGetDeviceInterfaceProperty](#)

[SetupDiOpenDeviceInterfaceRegKey](#)

[SetupDiSetDeviceInterfaceProperty](#)

# DEVPKEY\_DeviceInterface\_Restricted

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DeviceInterface\_Restricted device interface property indicates that the device interface on which it is present and set to TRUE, should be treated with privileged access by system components that honor the setting.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DeviceInterface_Restricted
Property-data-type identifier	<b>DEVPROP_TYPE_BOOLEAN</b>
Property access	Read-only access by installation applications and installers
Localized?	No

## Remarks

You can call [SetupDiGetDeviceInterfaceProperty](#) to retrieve the value of DEVPKEY\_DeviceInterface\_Restricted.

## Requirements

Version	Available starting with Windows 8.
Header	Devpkey.h (include Devpkey.h)

## See also

[SetupDiGetDeviceInterfaceProperty](#)

# DEVPKEY\_DrvPkg\_BrandingIcon

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_BrandingIcon device property represents a list of icons that associate a device instance with a vendor.

Property key	DEVPKEY_DrvPkg_BrandingIcon
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

A branding icon can be specified as an .ico file or as a resource within an executable file.

The format of an icon list is the same as that described for the [DEVPKEY\\_DrvPkg\\_Icon](#) device property.

You can set the value of DEVPKEY\_DrvPkg\_BrandingIcon by an [INF AddProperty directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs a device. You can retrieve the value of DEVPKEY\_DrvPkg\_BrandingIcon by calling [SetupDiGetDeviceProperty](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_DrvPkg\\_Icon](#)

[INF AddProperty Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DrvPkg\_DetailedDescription

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_DetailedDescription device property represents a detailed description of the capabilities of a device instance.

Property key	DEVPKEY_DrvPkg_DetailedDescription
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Data format	Limited set of XML tags
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

The detailed description string is in XML format. XML format makes it possible for Windows to format the display of the information based on the following subset of supported Hypertext Markup Language (HTML) tags. The operation of these tags resembles the operation of HTML tags.

Heading level tags

<h1>, <h2>, <h3>

List tags

<ul>, <ol>, <li>

Paragraph tag

<p>

You can set the value of DEVPKEY\_DrvPkg\_DetailedDescription by an [INF AddProperty directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs the device. You can retrieve the value of DEVPKEY\_DrvPkg\_DetailedDescription by calling [SetupDiGetDeviceProperty](#).

The following is an example of how to use an INF AddProperty directive to set the value of DEVPKEY\_DrvPkg\_DetailedDescription for a device instance that is installed by an INF DDInstall section "SampleDDInstallSection":

```
[SampleDDinstallSection]
...
AddProperty=SampleAddPropertySection
...

[SampleAddPropertySection]
DeviceDetailedDescription,,,,"<xml><h1>Microsoft DiscoveryCam 530</h1><h2>Overview</h2>The Microsoft
DiscoveryCam is great.<p>Really.<p><h2>Features</h2>The Microsoft DiscoveryCam has three features.<ol>
<li>Feature 1</li><li>Feature 2</li><li>Feature 3</li></ol></xml>"
```

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[INF AddProperty Directive](#)

[INF \*DD\Install\* Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DrvPkg\_DocumentationLink

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_DocumentationLink device property represents a URL to the documentation for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DrvPkg_DocumentationLink
Property-data-type identifier	DEVPROP_TYPE_STRING
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

The documentation link URL should be a link to a file that contains information about a device. This property is intended to provide Web-accessible documentation for a device. The file can be an HTML page, a .pdf file, a .doc file, or other file type. The only restriction is that all the documentation content must be contained within the URL-specified file. For example, an \*.htm file that is self-contained is valid, an \*.htm file that refers to other graphics files is not valid, and an \*.mta Web archive file that contains referenced graphic files is valid.

The URL can contain parameters. For example, the following URL contains a **prod** parameter that supplies the value "DSC530", a **rev** parameter that supplies the value "34", and a **type** parameter that supplies the value "doc":

```
http://www.microsoft.com/redirect?prod=DSC530&rev=34&type=docs
```

Microsoft does not provide Web hosting or redirection for a webpage that is specified by a DEVPKEY\_DrvPkg\_DocumentationLink property value. The URL must link to a webpage that is maintained by the [driver package](#) provider.

When a user clicks the website link that is displayed in Setup-generated end-user dialog box, Windows adds the following information to the HTTP request that includes the URL supplied by DEVPKEY\_DrvPkg\_DocumentationLink:

- The Windows version, as specified by a **pver** parameter. For example, "pver=6.0" specifies Windows Vista.
- The stock keeping unit (SKU), as specified by the **sbp** parameter, which can be set to per or pro. For example, "sbp=pro" specifies the professional edition.
- The local identifier (LCID), as specified by the **olcid** parameter. For example, "olcid=0x409" specifies the English (Standard) language.
- The most specific hardware identifier for a device, as specified by the **pnpid** parameter. For example, "pnpid=PCI%CVEN\_8086%26DEV\_2533%26SUBSYS\_00000000%26REV\_04" specifies the hardware identifier for a PCI device.

For privacy reasons, user information and the serial number of device is not included in the HTTP request.

The following example shows the type of HTTP request that would be sent to a web server:  
[http://www.microsoft.com/redirect?prod=DSC530&rev34&type=docs&pver=6.0&spb=pro&olcid=0x409&pnpid=PCI%5CVEN\\_8086%26DEV\\_2533%26SUBSYS\\_00000000%26REV\\_04](http://www.microsoft.com/redirect?prod=DSC530&rev34&type=docs&pver=6.0&spb=pro&olcid=0x409&pnpid=PCI%5CVEN_8086%26DEV_2533%26SUBSYS_00000000%26REV_04)

You can set the value of DEVPKEY\_DrvPkg\_DocumentationLink by an **INF AddProperty directive** that is included in the **INF DDInstall section** of the INF file that installs the device. You can retrieve the value of DEVPKEY\_DrvPkg\_DocumentationLinkproperty by calling **SetupDiGetDeviceProperty**.

The following is an example of how to use an INF AddProperty directive to set the value of DEVPKEY\_DrvPkg\_DocumentationLink for a device that is installed by an INF DDInstall section "SampleDDInstallSection":

```
[SampleDDinstallSection]
...
AddProperty=SampleAddPropertySection
...

[SampleAddPropertySection]
DeviceDocumentationLink,,,,"http://www.microsoft.com/redirect?prod=DSC530&rev34&type="docs"
...
```

## Requirements

Requirement	Description
Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[INF AddProperty Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DrvPkg\_Icon

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_Icon device property represents a list of device icons that Windows uses to visually represent a device instance.

Property key	DEVPKEY_DrvPkg_Icon
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING_LIST</a>
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

Each icon in the list is specified by a path of an icon file (\*.ico) or a reference to an icon resource in an executable file.

The first icon in the list is used as the default. Additional icons can be supplied that provide different visual representations of a device. Windows includes a user interface that allows a user to select which icon Windows displays. For example, the Microsoft DiscoveryCam 530 is available in blue, green, and red. Microsoft supplies an icon for each color. Windows uses the blue icon by default because it is the first one in the list. However, Windows users can also choose the green icon or the red icon.

The icon list is a NULL-separated list of icon specifiers. An icon specifier is either a path of an icon file (\*.ico) or an icon-resource specifier, as follows:

- The format of the path to an icon file is *DirectoryPath\filename.ico*.
- An icon-resource specifier has the following entries:

```
@executable-file-path,resource-identifier
```

The first character of icon-resource specifier is the at sign (@) followed by the path of an executable file (an \*.exe or a \*.dll file), followed by a comma separator (,), and then the *resource-identifier* entry.

For example, the icon specifier "@shell32.dll,-30" represents the executable file "shell32.dll" and the resource identifier "-30".

A resource identifier must be an integer value, which corresponds to a resource within the executable file, as follows:

- If the supplied identifier is negative, Windows uses the resource in the executable file whose identifier is equal to the absolute value of the supplied identifier.
- If the supplied identifier is zero, Windows uses the resource in the executable file whose identifier has the lowest value in the executable file.

- If the supplied identifier is positive, for example, the value *n*, Windows uses the resource in the executable file whose identifier is the *n*+1 lowest value in the executable file. For example, if the value of *n* is 1, Windows uses the resource whose identifier has the second lowest value in the executable file.

You can set the value of DEVKEY\_DrvPkg\_Icon by an [INF AddProperty directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs the device. You can retrieve the value of DEVKEY\_DrvPkg\_Icon by calling [SetupDiGetDeviceProperty](#).

The following is an example of how to use an INF AddProperty directive to set DEVKEY\_DrvPkg\_Icon for a device that is installed by an INF *DDInstall* section "SampleDDInstallSection":

```
[SampleDDInstallSection]
...
AddProperty=SampleAddPropertySection
...

[SampleAddPropertySection]
DeviceIcon,,,,"SomeResource.dll,-2","SomeIcon.icon"
...
```

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devkey.h (include Devkey.h)

## See also

[INF AddProperty Directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DrvPkg\_Model

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_Model device [driver package](#) property represents the model name for a device instance.

Property key	DEVPKEY_DrvPkg_Model
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

You can set the value of DEVPKEY\_DrvPkg\_Model by an [INF AddProperty directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs the device. You can retrieve the value of the DEVPKEY\_DrvPkg\_Model property by calling [SetupDiGetDeviceProperty](#).

The following is an example of how to use an INF AddProperty directive to set the value of DEVPKEY\_DrvPkg\_Model for a device that is installed by an INF DDInstall section "SampleDDInstallSection":

```
[SampleDDInstallSection]
...
AddProperty=SampleAddPropertySection
...

[SampleAddPropertySection]
DeviceModel,,,,"DSC-530"
...
```

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[INF AddProperty directive](#)

[INF DDInstall Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_DrvPkg\_VendorWebSite

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_DrvPkg\_VendorWebSite device property represents a vendor URL for a device instance.

ATTRIBUTE	VALUE
Property key	DEVPKEY_DrvPkg_VendorWebSite
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a> ">
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

The URL can be a link to the root of the vendor website, a webpage within a website, or a redirection page. The URL can also contain parameters, for example, the following URL contains a **prod** parameter that supplies the product identifier "DSC530" and a **rev** parameter that supplies the number "34":

```
http://www.microsoft.com/redirect?prod=DSC530&rev=34
```

You can set the value of DEVPKEY\_DrvPkg\_VendorWebSite by an [INF AddProperty directive](#) that is included in the [INF DDInstall section](#) of the INF file that installs the device. You can retrieve the value of DEVPKEY\_DrvPkg\_VendorWebSite by calling [SetupDiGetDeviceProperty](#).

The following is an example of how to use an INF AddProperty directive to set the DEVPKEY\_DrvPkg\_VendorWebSite property value for a device that is installed by an INF *DDInstall* section "SampleDDInstallSection":

```
[SampleDDinstallSection]
...
AddProperty=SampleAddPropertySection
...

[SampleAddPropertySection]
DeviceVendorWebsite,,,,"http://www.microsoft.com/redirect?prod=DSC530&rev=34"
...
```

## Requirements

REQUIREMENT	DESCRIPTION
Version	Available in Windows Vista and later versions of Windows.
Header	Devkey.h (include Devpkey.h)

## See also

[INF AddProperty Directive](#)

[INF \*DDInstall\* Section](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_NAME (Device Instance)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_NAME device property represents the name of a device instance.

Property key	DEVPKEY_NAME
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

The value of DEVPKEY\_NAME device should be used to identify a device instance to an end-user in a user interface item.

The retrieved property value is the same as the value of the [DEVPKEY\\_Device\\_FriendlyName](#) device property, if [DEVPKEY\\_Device\\_FriendlyName](#) is set. Otherwise, the value of DEVPKEY\_NAME is same as the value of the [DEVPKEY\\_Device\\_DeviceDesc](#) device property.

You can call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_NAME property.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support a corresponding name property. However, these earlier versions of Windows do support properties that correspond to [DEVPKEY\\_Device\\_FriendlyName](#) and [DEVPKEY\\_Device\\_DeviceDesc](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_Device\\_DeviceDesc](#)

[DEVPKEY\\_Device\\_FriendlyName](#)

[SetupDiGetDeviceProperty](#)

# DEVPKEY\_NAME (Device Interface)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_NAME device interface property represents the name of a device interface.

Property key	DEVPKEY_NAME
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers.
Localized?	Yes

## Remarks

The value of the DEVPKEY\_NAME should be used to identify an interface to an end-user in a user interface item.

The value of DEVPKEY\_NAME is the same as the value of the [DEVPKEY\\_DeviceInterface\\_FriendlyName](#) device property, if DEVPKEY\_DeviceInterface\_FriendlyName is set. Otherwise, DEVPKEY\_NAME does not exist.

You can retrieve the value of DEVPKEY\_NAME by calling [SetupDiGetDeviceInterfaceProperty](#).

For information about device interfaces, see [Device Interface Classes](#) and the [INF AddInterface Directive](#).

Windows Server 2003, Windows XP, and Windows 2000 do not directly support a corresponding name property. However, these earlier versions of Windows do support a property that corresponds to DEVPKEY\_DeviceInterface\_FriendlyName.

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_DeviceInterface\\_FriendlyName](#)

[INF AddInterface Directive](#)

[SetupDiGetDeviceInterfaceProperty](#)

# DEVPKEY\_NAME (Device Setup Class)

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_NAME device property represents the name of a [device setup class](#).

Property key	DEVPKEY_NAME
Property-data-type identifier	<a href="#">DEVPROP_TYPE_STRING</a>
Property access	Read-only access by installation applications and installers
Localized?	Yes

## Remarks

You can use the value of DEVPKEY\_NAME to identify a device setup class to an end-user in a user interface item.

If DEVPKEY\_DeviceClass\_Name is set, the value of DEVPKEY\_NAME is the same as the value of the [DEVPKEY\\_DeviceClass\\_Name](#) device property. Otherwise, the DEVPKEY\_NAME value is the same as the value of the [DEVPKEY\\_DeviceClass\\_ClassName](#) device property.

You can call [SetupDiGetClassProperty](#) or [SetupDiGetClassPropertyEx](#) to retrieve the value of DEVPKEY\_NAME for a device setup class.

Windows Server 2003, Windows XP, and Windows 2000 do not directly support a corresponding name property. However, these earlier versions of Windows do support properties that correspond to DEVPKEY\_DeviceClass\_Name and DEVPKEY\_DeviceClass\_ClassName.

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Devpkey.h (include Devpkey.h)

## See also

[DEVPKEY\\_DeviceClass\\_ClassName](#)

[DEVPKEY\\_DeviceClass\\_Name](#)

[SetupDiGetClassProperty](#)

[SetupDiGetClassPropertyEx](#)

[SetupDiGetClassDescription](#)

[SetupDiClassNameFromGuid](#)

# DEVPKEY\_Numa\_Proximity\_Domain

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPKEY\_Numa\_Proximity\_Domain device property represents the proximity domain of a Non-Uniform Memory Architecture (NUMA).

Property key	DEVPKEY_Numa_Proximity_Domain
Property-data-type identifier	<a href="#">DEVPROP_TYPE_INT32</a>
Property access	Read-only access by installation applications and installers; read and write access by a device driver
Localized?	No

## Remarks

The value DEVPKEY\_Numa\_Proximity\_Domain is a numeric value that represents a domain ID.

Typically, the operating system sets the value of DEVPKEY\_Numa\_Proximity\_Domain by retrieving the corresponding information from system firmware.

You can retrieve the value of DEVPKEY\_Numa\_Proximity\_Domain by calling [IoSetDevicePropertyData](#) or [IoGetDevicePropertyData](#) in a device driver.

You can also call [SetupDiGetDeviceProperty](#) to retrieve the value of DEVPKEY\_Numa\_Proximity\_Domain.

The value of this property is owned by Windows and should be treated as read-only by drivers and applications.

Windows Server 2003, Windows XP, and Windows 2000 do not support this property.

## Requirements

Version	Available starting with Windows Vista.
Header	Devpkey.h (include Devpkey.h)

# DEVPRIVATE\_DES

11/2/2020 • 2 minutes to read • [Edit Online](#)

**Note** This structure is for internal use only.

The DEVPRIVATE\_DES structure is used for specifying either a resource list or a resource requirements list that describes private device-specific resource usage for a device instance. For more information about resource lists and resource requirements list, see [Hardware Resources](#).

# DEVPRIVATE\_RANGE

11/2/2020 • 2 minutes to read • [Edit Online](#)

**Note** This structure is for internal use only.

The DEVPRIVATE\_RANGE structure specifies a resource requirements list that describes private device-specific resource usage for a device instance. For more information about resource requirements list, see [Hardware Resources](#).

# DEVPRIVATE\_RESOURCE

11/2/2020 • 2 minutes to read • [Edit Online](#)

**Note** This structure is for internal use only.

The DEVPRIVATE\_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes private device-specific resource usage for a device instance. For more information about resource lists and resource requirements list, see [Hardware Resources](#).

# DEVPROP\_MASK\_TYPE

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_MASK\_TYPE mask can be combined in a bitwise AND with a [property-data-type identifier](#) to extract the [base-data-type identifier](#) from a property-data-type identifier.

## Remarks

This mask cannot be used as a base-data-type identifier, a property-data-type modifier, or a property-data-type identifier.

For information about how to extract the DEVPROP\_TYPEMOD\_Xxx [property-data-type modifier](#) from a property-data-type identifier, see [DEVPROP\\_MASK\\_TYPEMOD](#).

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

## See also

[DEVPROP\\_MASK\\_TYPEMOD](#)

# DEVPROP\_MASK\_TYPEMOD

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_MASK\_TYPEMOD mask can be combined in a bitwise AND with a [property-data-type identifier](#) to extract the DEVPROP\_TYPEMOD\_Xxx[property-data-type modifier](#) from a property-data-type identifier.

## Remarks

This mask cannot be used as a base-data-type identifier, a property-data-type modifier, or property-data-type identifier.

For information about how to extract the [base-data-type identifier](#) from a property-data-type identifier, see [DEVPROP\\_MASK\\_TYPE](#).

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

## See also

[DEVPROP\\_MASK\\_TYPE](#)

# DEVPROP\_TYPE\_BINARY

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_BINARY identifier represents the base-data-type identifier that indicates that the data type is an array of BYTE-typed unsigned values.

## Remarks

The DEVPROP\_TYPE\_BINARY property type cannot be combined with the property-data-type modifiers.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_BINARY, call the corresponding SetupDiSetXxx property function and set the function input parameters as follows:

- Set the *.PropertyType* parameter to DEVPROP\_TYPE\_BINARY, set the *PropertyBuffer* parameter to a pointer to a buffer that contains an array of BYTE value, and set the *PropertyBufferSize* parameter to the size, in bytes, of the buffer.
- Set the remaining function parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_BOOLEAN

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_BOOLEAN property type represents the base-data-type identifier that indicates that the data type is a DEVPROP\_BOOLEAN-typed Boolean value.

## Remarks

The DEVPROP\_BOOLEAN data type and valid Boolean values are defined as follows:

```
typedef CHAR DEVPROP_BOOLEAN, *PDEVPROP_BOOLEAN;
#define DEVPROP_TRUE ((DEVPROP_BOOLEAN)-1)
#define DEVPROP_FALSE ((DEVPROP_BOOLEAN) 0)
```

DEVPROP\_TYPE\_BOOLEAN can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_BOOLEAN, call the corresponding SetupDiSetXxx property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_BOOLEAN, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a DEVPROP\_FALSE or DEVPROP\_TRUE value, and set the *PropertyBufferSize* parameter to `sizeof(DEVPROP_BOOLEAN)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_BYTE

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_BYTE identifier represents the base-data-type identifier that indicates the data type is a BYTE-typed unsigned integer.

## Remarks

DEVPROP\_TYPE\_BYTE can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose data type is DEVPROP\_TYPE\_BYTE, call the corresponding SetupDiSetXxx property function, setting the function input parameters as follows:

- Set the *PropertyName* parameter to DEVPROP\_TYPE\_BYTE, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one BYTE value, and set the *PropertyBufferSize* parameter to `sizeof(BYTE)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_CURRENCY

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_CURRENCY identifier represents the base-data-type identifier that indicates that the data type is a CURRENCY-typed value.

## Remarks

DEVPROP\_TYPE\_CURRENCY can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of This Type

To set a property whose base data type is DEVPROP\_TYPE\_CURRENCY, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyName* parameter to `DEVPROP_TYPE_CURRENCY`, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a CURRENCY value, and set the *PropertyBufferSize* parameter to `sizeof(CURRENCY)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_DATE

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_DATE property type represents the base-data-type identifier that indicates that the data type is a DOUBLE-typed value that specifies the number of days since December 31, 1899. For example, January 1, 1900, is 1.0; January 2, 1900, is 2.0; and so on.

## Remarks

DEVPROP\_TYPE\_DATE can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_DATE, call the corresponding SetupDiSetXxx property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_DATE, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a DATE value, and set the *PropertyBufferSize* parameter to `sizeof(DATE)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_DECIMAL

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_INT64 identifier represents the base-data-type identifier that indicates that the data type is a DECIMAL-typed value.

## Remarks

DEVPROP\_TYPE\_DECIMAL can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose data type is DEVPROP\_TYPE\_DECIMAL, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_DECIMAL`, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one DECIMAL value, and set the *PropertyBufferSize* parameter to `sizeof(DECIMAL)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_DEVPROPKEY

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_DEVPROPKEY identifier represents the base-data-type identifier that indicates the data type is a DEVPROPKY-typed device property key.

## Remarks

The DEVPROP\_TYPE\_DEVPROPKEY property type can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_DEVPROPKEY, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_DEVPROPKEY`, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a `DEVPROPKY` structure, and set the *PropertyBufferSize* parameter to `sizeof(DEVPROPKY)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_DEVPROPTYPE

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_DEVPROPTYPE identifier represents the base-data-type identifier that indicates the data type is a DEVPROPTYPE-typed value.

## Remarks

The DEVPROP\_TYPE\_DEVPROPTYPE property type can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_DEVPROPTYPE, call the corresponding `SetupDiSetXxx` property function, setting the function input parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_DEVPROPTYPE`, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a DEVPROPTYPE value, and set the *PropertyBufferSize* parameter to `sizeof(DEVPROPTYPE)`.
- Set the remaining function parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_DOUBLE

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_DOUBLE identifier represents the base-data-type identifier that indicates that the data type is a DOUBLE-typed IEEE floating-point number.

## Remarks

DEVPROP\_TYPE\_DOUBLE can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_DOUBLE, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_DOUBLE`, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one `ULONG` value, and set the *PropertyBufferSize* parameter to `sizeof(DOUBLE)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_EMPTY

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_EMPTY identifier represents a special base-data-type identifier that indicates that a property does not exist.

## Remarks

Use this base-data-type identifier with the device property functions to delete a property.

If a device property function returns this base-data-type identifier, the property does not exist.

DEVPROP\_TYPE\_EMPTY cannot be combined with the property-data-type modifiers [DEVPROP\\_TYPEMOD\\_ARRAY](#) or [DEVPROP\\_TYPEMOD\\_LIST](#).

### Deleting a Property

To delete a property, call the corresponding SetupDiSetXxx property function and set the function parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_EMPTY, the *PropertyBuffer* parameter to **NULL**, and the *PropertyBufferSize* parameter to zero.
- Set the other function input parameters as appropriate to set the property.

If DEVPROP\_TYPE\_EMPTY is used in an attempt to delete a property that does not exist, the delete operation will fail, and a call to [GetLastError](#) will return ERROR\_NOT\_FOUND.

### Retrieving a Property that Does Not Exist

A call to a SetupDiGetXxx property function that attempts to retrieve a device property that does not exist will fail, and a subsequent call to [GetLastError](#) will return ERROR\_NOT\_FOUND. The called SetupAPI property function will set the *\*PropertyName* parameter to DEVPROP\_TYPE\_EMPTY.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_ERROR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPROP\_TYPE\_ERROR identifier represents the base-data-type identifier for the Microsoft Win32 error code values that are defined in WINERROR.H.

## Remarks

In Windows Vista and later versions of Windows, the [unified device property model](#) also defines a [DEVPROP\\_TYPE\\_NTSTATUS](#) base-data-type identifier for NTSTATUS error code values.

You can combine DEVPROP\_TYPE\_ERROR only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of This Type

To set a property whose base data type is DEVPROP\_TYPE\_ERROR, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_ERROR.
- Set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one Win32 error code value.
- Set the *PropertyBufferSize* parameter to `sizeof(ULONG)`.
- Set the remaining function parameters as appropriate to set the property.

### Retrieving the Descriptive Text for a Win32 Error Code Value

To retrieve the descriptive text that is associated with a Win32 error code, call the [FormatMessage](#) function (documented in the Windows SDK) as follows:

- Include the `FORMAT_MESSAGE_FROM_SYSTEM` flag in the value of the *dwFlags* parameter.
- Set the *dwMessageID* parameter to the error code value.
- Set the other options and parameters as appropriate to retrieve the descriptive text.

## Requirements

Version	Windows Vista and later versions of Windows.
Header	Devpropdef.h (include Devpropdef.h)

## See also

[DEVPROP\\_TYPE\\_NTSTATUS](#)

[DEVPROP\\_TYPEMOD\\_ARRAY](#)

# DEVPROP\_TYPE\_FILETIME

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_FILETIME property type represents the base-data-type identifier that indicates that the data type is a FILETIME-typed value.

## Remarks

We recommend that all time values be represented in Coordinated Universal Time (UTC) units.

DEVPROP\_TYPE\_FILETIME can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_FILETIME, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyName* parameter to `DEVPROP_TYPE_DATE`, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a FILETIME structure, and set the *PropertyBufferSize* parameter to `sizeof(FILETIME)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_FLOAT

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_INT64 identifier represents the base-data-type identifier that indicates that the data type is a FLOAT-typed IEEE floating-point number.

## Remarks

DEVPROP\_TYPE\_FLOAT can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_FLOAT, call the corresponding SetupDiSetXxx property function, setting the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_FLOAT, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one FLOAT value, and set the *PropertyBufferSize* parameter to `sizeof(FLOAT)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_GUID

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_GUID identifier represents the base-data-type identifier that indicates that the data type is a GUID-typed globally unique identifier (GUID).

## Remarks

DEVPROP\_TYPE\_GUID can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_GUID, call the corresponding SetupDiSetXxx property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_GUID, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a GUID value, and set the *PropertyBufferSize* parameter to `sizeof(GUID)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_INT16

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_INT16 identifier represents the base-data-type identifier that indicates the data type is a SHORT-typed signed integer.

## Remarks

DEVPROP\_TYPE\_SHORT can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_INT16, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_INT16`, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one `SHORT` value, and set the *PropertyBufferSize* parameter to `sizeof(SHORT)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_INT32

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_INT32 identifier represents the base-data-type identifier that indicates that the data type is a LONG-typed signed integer.

## Remarks

DEVPROP\_TYPE\_INT32 can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_INT32, call the corresponding `SetupDiSetXxx` property function, setting the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_INT32, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one LONG value, and set the *PropertyBufferSize* parameter to `sizeof(LONG)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_INT64

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_INT64 identifier represents the base-data-type identifier that indicates that the data type is a LONG64-typed signed integer.

## Remarks

DEVPROP\_TYPE\_INT64 can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_INT64, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_INT64, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one LONG64 value, and set the *PropertyBufferSize* parameter to `sizeof(LONG64)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_NTSTATUS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DEVPROP\_TYPE\_NTSTATUS identifier represents the base-data-type identifier for the NTSTATUS status code values that are defined in Ntstatus.h.

## Remarks

In Windows Vista and later versions of Windows, the [unified device property model](#) also defines a **DEVPROP\_TYPE\_ERROR** base-data-type identifier for Microsoft Win32 error code values.

You can combine DEVPROP\_TYPE\_NTSTATUS only with the **DEVPROP\_TYPEMOD\_ARRAY** property-data-type modifier.

### Setting a Property of This Type

To set a property whose base data type is DEVPROP\_TYPE\_NTSTATUS, call the corresponding **SetupDiSetXxx** property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_NTSTATUS.
- Set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one NTSTATUS value.
- Set the *PropertyBufferSize* parameter to **sizeof(NTSTATUS)**.
- Set the remaining function parameters as appropriate to set the property.

### Retrieving the Descriptive Text for a NTSTATUS Error Code Value

To retrieve the descriptive text that is associated with an NTSTATUS error code value, call the **FormatMessage** function (documented in the Windows SDK) as follows:

- Include a bitwise OR of the **FORMAT\_MESSAGE\_FROM\_SYSTEM** flag and the **FORMAT\_MESSAGE\_FROM\_HMODULE** flag in the value of the *dwFlags* parameter.
- Set the *lpSource* parameter to a handle to the *NtDLL.dll* module, which is the source for the descriptive text.
- Set the *dwMessageID* parameter to the error code value.
- Set the other options and parameters as appropriate to retrieve the descriptive text.

## Requirements

Version	Windows Vista and later versions of Windows.
Header	Devpropdef.h (include Devpropdef.h)

## See also

[DEVPROP\\_TYPE\\_ERROR](#)

[DEVPROP\\_TYPEMOD\\_ARRAY](#)

# DEVPROP\_TYPE\_NULL

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_NULL identifier represents a special base-data-type identifier that indicates that a device property exists. However, that the property has no value that is associated with the property.

## Remarks

Use this base-property-type identifier with the device property functions to delete the value that is associated with a device property.

If a device property function returns this base data type, the property exists, but the property has no value that is associated with it.

The DEVPROP\_TYPE\_NULL identifier cannot be combined with the property-data-type modifiers [DEVPROP\\_TYPEMOD\\_ARRAY](#) or [DEVPROP\\_TYPEMOD\\_LIST](#).

### Setting a Property of this Type

To set a property whose data type is DEVPROP\_TYPE\_NULL, call the corresponding [SetupDiSetXxx](#) property function and set the function parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_NULL, the *PropertyBuffer* parameter to **NULL**, and the *PropertyBufferSize* parameter to zero.
- Set the other function input parameters as appropriate to set the property.

### Retrieving a Property of this Type

A call to a [SetupDiGetXxx](#) property function that attempts to retrieve a device property that has no value will succeed and set the *\*PropertyType* parameter to DEVPROP\_TYPE\_NULL.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_SBYTE

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_BYTE identifier represents the base-data-type identifier that indicates the data type is a SBYTE-typed signed integer.

## Remarks

DEVPROP\_TYPE\_SBYTE can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose data type is DEVPROP\_TYPE\_BYTE, call the corresponding `SetupDiSetXxx` property function, and set the function parameters as follows:

- Set the *PropertyType* parameter to `DEVPROP_TYPE_BYTE`, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one `BYTE` value, and set the *PropertyBufferSize* parameter to `sizeof(BYTE)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR identifier represents the base-data-type identifier that indicates the data type is a variable-length, self-relative, SECURITY\_DESCRIPTOR-typed, security descriptor.

## Remarks

DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR cannot be combined with the property-data-type modifiers.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a variable length SECURITY\_DESCRIPTOR structure, and set the *PropertyBufferSize* parameter to the size, in bytes, of the security descriptor structure.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING identifier represents the base-data-type identifier that indicates the data type is a NULL-terminated Unicode string that contains a security descriptor in the Security Descriptor Definition Language (SDDL) format.

## Remarks

DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING can be combined only with the [DEVPROP\\_TYPEMOD\\_LIST](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING, call the corresponding `SetupDiSetXxx` property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a NULL-terminated security descriptor string, and set the *PropertyBufferSize* parameter to the size, in bytes, of the security descriptor string, including the NULL terminator.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_STRING

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_STRING property type represents the base-data-type identifier that indicates that the data type is a NULL-terminated Unicode string.

## Remarks

DEVPROP\_TYPE\_STRING can be combined only with the [DEVPROP\\_TYPEMOD\\_LIST](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_STRING, call the corresponding [SetupDiSetXxx](#) property function, setting the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_STRING, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a NULL-terminated Unicode string, and set the *PropertyBufferSize* parameter to the size, in bytes, of the string, including the NULL terminator.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdefh (include Devpropdefh)
--------	-----------------------------------

# DEVPROP\_TYPE\_STRING\_INDIRECT

12/5/2018 • 3 minutes to read • [Edit Online](#)

The DEVPROP\_TYPE\_STRING\_INDIRECT identifier represents the base-data-type identifier for a NULL-terminated Unicode string that contains an indirect string reference.

## Remarks

An indirect string reference describes a string resource that contains the actual string. The indirect string reference can appear in one of the following formats:

`@[path\]FileName,-ResourceID`

Windows extracts the string from the module that is specified by the *path* and *FileName* entries, and the resource identifier of the string is supplied by the *ResourceID* entry (excluding the required minus sign). The string resource is loaded from the module resource section that best matches one of the caller's preferred UI languages. The *path* entry is optional. If you specify the *path* entry, the module must be located in a directory that is in the system-defined search path.

`@InfName,%strkey%`

Windows extracts the string from the **INF Strings** section of the INF file in the %SystemRoot%\inf directory whose name is supplied by the *InfName* entry. The *strkey* token identifier should match the key of a line in the **Strings** section that best matches one of the caller's preferred UI languages. If no language-specific **Strings** sections exist, Windows uses the default **Strings** section.

You cannot combine DEVPROP\_TYPE\_STRING\_INDIRECT with any of the property-data-type modifiers.

## Setting a Property of This Type

To set a property whose base data type is DEVPROP\_TYPE\_STRING\_INDIRECT, call the corresponding **SetupDiSetXxx** property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_STRING\_INDIRECT.
- Set the *PropertyBuffer* parameter to a pointer to a buffer that contains the NULL-terminated string that supplies an indirect string reference.
- Set the *PropertyBufferSize* parameter to the size, in bytes, of the string.
- Set the remaining function parameters as appropriate to set the property.

## Retrieving the Value of This Property Type

When an application calls a **SetupDiGetXxx** property function to retrieve the value of a property of this base data type, Windows tries to locate the actual string that the property references. If Windows can retrieve the actual string, it returns the actual string to the caller and identifies the base data type of the retrieved property as **DEVPROP\_TYPE\_STRING**. Otherwise, Windows returns the indirect string reference and identifies the base data type of the retrieved property as DEVPROP\_TYPE\_STRING\_INDIRECT.

## Localizing Static Text

Starting with Windows Vista you can localize custom and standard string-type PnP static-text properties using resources from a PE image's string or resource tables by setting static-text property types to DEVPROP\_TYPE\_STRING\_INDIRECT. You can also add non-localized replacement-string data that can be formatted into the static text.

Strings located in a PE image's STRINGTABLE resource (as typically performed by LoadString) should use the

following format:

```
"@\"System32\\mydll.dll,-21\[:Fallback" String]"
```

```
"@System32\\mydll.dll,-21\[:Fallback String with %1, %2, ... to %n[;(Arg1,Arg2,...,ArgN)]]"
```

Strings located in a PE images's message-table resource (as typically performed by `RtlFindMessage`, more commonly used in drivers) should use the following format:

```
"@System32\\drivers\\mydrivers.sys,#21\[:Fallback String]"
```

```
"@System32\\drivers\\mydrivers.sys,#21\[:Fallback String with %1, %2, ... to %n[;(Arg1,Arg2,...,ArgN)]]"
```

A "Fallback String" is optional but useful because it can be returned if the resource can't be found or loaded. The fallback string is also returned to non-interactive system processes that are not impersonating a user, and as such cannot show localized text to users anyways.

This technique enables you to localize static-text pulled from the string or message table resource that best matches the caller's locale.

Windows will format the trailing arguments into the string (or the fallback string) when they are retrieved from the respective resource table, much as in the same manner as `RtlFormatMessage` does.

Custom and standard string-type PnP static-text is localized when you set the property by loading the resource from the component performing the set operation, which typically happens under the system default locale for system-level components.

Note: PE images can use either resource table type (STRINGTABLE resources, or message-table resources).

## Requirements

Version	Windows Vista and later versions of Windows.
Header	<code>Devpropdef.h</code> (include <code>Devpropdef.h</code> )

## See also

[DEVPROP\\_TYPE\\_STRING](#)

# DEVPROP\_TYPE\_STRING\_LIST

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_STRING\_LIST property type represents the base-data-type identifier that indicates that the data type is a [REG\\_MULTI\\_SZ](#)-typed list of Unicode strings.

## Remarks

DEVPROP\_TYPE\_STRING\_LIST cannot be combined with the property-data-type modifiers.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_STRING\_LIST, call the corresponding [SetupDiSetXxx](#) property function and set the function input parameters as follows:

- Set the *.PropertyType* parameter to DEVPROP\_TYPE\_STRING\_LIST, set the *PropertyBuffer* parameter to a pointer to a buffer that contains a [REG\\_MULTI\\_SZ](#) list of Unicode strings, and set the *PropertyBufferSize* parameter to the size, in bytes, of the list, including the final list NULL terminator.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_UINT16

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_UINT16 identifier represents the base-data-type identifier that indicates that the data type is a USHORT-typed unsigned integer.

## Remarks

DEVPROP\_TYPE\_UINT16 can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_UINT16, call the corresponding [SetupDiSetXxx](#) property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_UINT16, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one USHORT value, and set the *PropertyBufferSize* parameter to `sizeof(USHORT)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_UINT32

7/9/2019 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_UINT32 identifier represents the base-data-type identifier that indicates that the data type is a ULONG-typed unsigned integer.

## Remarks

DEVPROP\_TYPE\_UINT32 can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_UINT32, call the corresponding [SetupDiSetXxx](#) property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_UINT32, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one ULONG value, and set the *PropertyBufferSize* parameter to `sizeof(ULONG)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPE\_UINT64

12/5/2018 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPE\_UINT64 identifier represents the base-data-type identifier that indicates that the data type is a ULONG64-typed unsigned integer.

## Remarks

DEVPROP\_TYPE\_UINT64 can be combined only with the [DEVPROP\\_TYPEMOD\\_ARRAY](#) property-data-type modifier.

### Setting a Property of this Type

To set a property whose base data type is DEVPROP\_TYPE\_UINT64, call the corresponding [SetupDiSetXxx](#) property function and set the function input parameters as follows:

- Set the *PropertyType* parameter to DEVPROP\_TYPE\_UINT64, set the *PropertyBuffer* parameter to a pointer to a buffer that can contain at least one ULONG64 value, and set the *PropertyBufferSize* parameter to `sizeof(ULONG64)`.
- Set the other function input parameters as appropriate to set the property.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

# DEVPROP\_TYPEMOD\_ARRAY

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPEMOD\_ARRAY identifier represents a property-data-type modifier that can be combined with the [base-data-type identifiers](#) to create a property-data-type identifier that represents an array of base-data-type values.

## Remarks

The DEVPROP\_TYPEMOD\_ARRAY identifier can be combined only with the fixed-length base-data-type identifiers ([DEVPROPTYPE](#) values) that are associated with data. The DEVPROP\_TYPEMOD\_ARRAY identifier cannot be combined with [DEVPROP\\_TYPE\\_EMPTY](#), [DEVPROP\\_TYPE\\_NULL](#), or any of the variable-length base-data-type identifiers.

To create a property-data-type identifier that represents an array of base-data-type values, perform a bitwise OR between DEVPROP\_TYPEMOD\_ARRAY and the corresponding DEVPROP\_TYPE\_Xxx identifier. For example, to specify an array of unsigned bytes, perform the following bitwise OR: (DEVPROP\_TYPEMOD\_ARRAY | [DEVPROP\\_TYPE\\_BYTE](#)).

The size, in bytes, of an array of base-data-type values is the size, in bytes, of the array.

For information about how to create a property-data-type identifier that represents a [REG\\_MULTI\\_SZ](#) list of NULL-terminated Unicode strings, see [DEVPROP\\_TYPEMOD\\_LIST](#).

## Requirements

Header

Devpropdefh (include Devpropdefh)

## See also

[DEVPROPTYPE](#)

[DEVPROP\\_TYPEMOD\\_LIST](#)

# DEVPROP\_TYPEMOD\_LIST

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROP\_TYPEMOD\_LIST identifier represents a property-data-type modifier that can be combined only with the [base-data-type identifiers](#) **DEVPROP\_TYPE\_STRING** and **DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR\_STRING** to create a property-data-type identifier that represents a [REG\\_MULTI\\_SZ](#) list of NULL-terminated Unicode strings.

## Remarks

DEVPROP\_TYPEMOD\_LIST cannot be combined with **DEVPROP\_TYPE\_EMPTY**, **DEVPROP\_TYPE\_NULL**, **DEVPROP\_TYPE\_SECURITY\_DESCRIPTOR**, or any of the fixed length base-data-type identifiers.

To create a property-data-type identifier that represents a string list, perform a bitwise OR between the DEVPROP\_TYPEMOD\_LIST property-data-type modifier and the corresponding DEVPROP\_TYPE\_Xxx identifier. For example, to specify a [REG\\_MULTI\\_SZ](#) list of Unicode strings, perform the following bitwise OR:  
(DEVPROP\_TYPEMOD\_LIST | DEVPROP\_TYPE\_STRING).

The size of a [REG\\_MULTI\\_SZ](#) list of NULL-terminated Unicode strings is size of the list including the final **NULL** that terminated the list.

For information about how to create a property-data-type identifier that represents an array of fixed length data values, see [DEVPROP\\_TYPEMOD\\_ARRAY](#).

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

## See also

[DEVPROP\\_TYPEMOD\\_ARRAY](#)

# DEVPROPKEY structure

11/2/2020 • 2 minutes to read • [Edit Online](#)

In Windows Vista and later versions of Windows, the DEVPROPKEY structure represents a device property key for a device property in the [unified device property model](#).

## Syntax

```
struct DEVPROPKEY {  
    DEVPROPGUID fmtid;  
    DEVPROPID     pid;  
};
```

## Members

### fmtid

A DEVPROPGUID-typed value that specifies a property category.

The DEVPROPGUID data type is defined as:

```
typedef GUID  DEVPROPGUID, *PDEVPROPGUID;
```

### pid

A DEVPROPID-typed value that uniquely identifies the property within the property category. For internal system reasons, a property identifier must be greater than or equal to two.

The DEVPROPID data type is defined as:

```
typedef ULONG DEVPROPID, *PDEVPROPID;
```

## Remarks

The DEVPROPKEY structure is part of the [unified device property model](#).

The basic set of system-supplied device property keys are defined in *Devpkey.h*.

The [DEFINE\\_DEVPROPKEY](#) macro creates an instance of a DEVPROPKEY structure that represents a device property key.

## Requirements

Header	Devpropdef.h (include Devpropdef.h)
--------	-------------------------------------

## See also

[DEFINE\\_DEVPROPKEY](#)

# DIF\_ADDPROPERTYPAGE\_ADVANCED

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_ADDPROPERTYPAGE\_ADVANCED request allows an installer to supply one or more custom property pages for a device.

## When Sent

When a user clicks on the properties for a device in Device Manager or in Control Panel.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Optionally supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set. If *DeviceInfoSet* is **NULL**, Windows is requesting property pages for the [device setup class](#).

### Device Installation Parameters

Device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) are associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

### Class Installation Parameters

An [SP\\_ADDPROPERTYPAGE\\_DATA](#) structure is associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

## Installer Output

### Device Installation Parameters

An installer can modify the device installation parameters.

### Class Installation Parameters

An installer can modify the [SP\\_ADDPROPERTYPAGE\\_DATA](#) to supply custom pages.

## Installer Return Value

A co-installer can return NO\_ERROR or a Win32 error. A co-installer should not return ERROR\_DI\_POSTPROCESSING\_REQUIRED for this DIF request.

A class installer returns NO\_ERROR if it successfully supplies pages. Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

In response to this DIF request an installer can supply custom property pages. Handling this DIF request allows you to supply property pages from a class installer or co-installer and removes the need for a separate DLL that acts as a property-page provider.

An installer typically handles this DIF request to add a new device-specific or setup-class-specific property page. An installer can also replace the system-supplied driver property page, resource property page, or power property page for a device. If an installer replaces a system-supplied page, the installer must set the appropriate flag in the device installation parameters for the device:

#### DI\_DRIVERPAGE\_ADDED

The installer supplied a driver property page.

#### DI\_RESOURCEPAGE\_ADDED

The installer supplied a resource property page.

#### DI\_FLAGSEX\_POWERPAGE\_ADDED

The installer supplied a power property page.

An installer cannot replace the system-supplied general properties page.

Windows only displays one driver page, one resource page, and one power page for a device. An installer should not supply a replacement system page if a previous installer already supplied a page of that type. This constraint does not apply to nonsystem-supplied property pages.

A co-installer should add custom pages in its preprocessing pass.

If an installer allows a user to set a property that requires Windows to remove and restart the device, the installer must set the DI\_FLAGSEX\_PROPCHANGE\_PENDING flag in the device installation parameters from its **DialogProc** routine.

For more information about how to provide device property pages, see [Providing Device Property Pages](#).

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SP\\_ADDPROPERTYPAGE\\_DATA](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEWINSTALL\\_PARAMS](#)

# DIF\_ADDPROPERTYPAGE\_BASIC

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_ALLOW\_INSTALL

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_ALLOW\_INSTALL request asks the installers for a device whether Windows can proceed to install the device.

## When Sent

After selecting a driver for the device but before installing the device.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Should not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR or a Win32 error. A co-installer should not return ERROR\_DI\_POSTPROCESSING\_REQUIRED for this DIF request.

A class installer typically returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

Typical Win32 error codes for this DIF request include ERROR\_DI\_DONT\_INSTALL and ERROR\_NON\_WINDOWS\_NT\_DRIVER.

**Note** Class installers and co-installers should not return ERROR\_REQUIRES\_INTERACTIVE\_WINDOWSTATION since that causes the device installation to fail. If the device installation requires user interaction, class installers and co-installers should support a [finish-install action](#).

## Default DIF Code Handler

None

## Installer Operation

In response to a DIF\_ALLOW\_INSTALL request an installer confirms whether Windows can install the device.

An installer can fail this request if it determines that the selected driver is incorrect (for example, if the driver is a Windows 9x-only driver that will not work correctly on NT-based operating systems) or if it determines that a selected driver is known to have bugs.

An installer might fail this request if the DI QUIETINSTALL flag is set in the device installation parameters and the installer has to display UI during device installation. However, this failure is rare because an installer can typically supply any UI pages in response to the DIF\_NEWDEVICEWIZARD\_FINISHINSTALL request. In that case, UI does not prevent the installer from succeeding a DIF\_ALLOW\_INSTALL request for which the quiet flag is set. However, if an installer cannot limit its UI to the finish-install case, the installer must fail this DIF request if the DI QUIETINSTALL flag is set. An installer might have this restriction, for example, if it calls vendor-supplied code that displays UI.

If an installer fails this DIF request, Windows stops the installation.

If an installer fails this DIF request and DI QUIETINSTALL is not set in the device installation parameters, the installer should display a dialog box with a message that explains why the device is not being installed.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_ASSIGNRESOURCES

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_CALCDISKSPACE

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_DESTROYPRIVATEDATA

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_DESTROYPRIVATEDATA request directs a class installer to free any memory or resources it allocated and stored in the **ClassInstallReserved** field of the [SP\\_DEVINSTALL\\_PARAMS](#) structure.

## When Sent

When Windows destroys a [device information set](#) or an [SP\\_DEVINFO\\_DATA](#) element, or when Windows discards its list of co-installers and class installer for a device.

## Who Handles

Class Co-installer	Does not handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to a device information set.

### *DeviceInfoData*

Optionally supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies a device in the device information set.

### Device Installation Parameters

Device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) are associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer can clear the **ClassInstallReserved** field in the device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)).

## Installer Return Value

A co-installer does not handle this DIF request. It simply returns NO\_ERROR in its preprocessing pass.

A class installer typically returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

In response to a DIF\_DESTROYPRIVATEDATA request a class installer frees any memory or resources it allocated and stored in the **ClassInstallReserved** field of the [SP\\_DEVINSTALL\\_PARAMS](#) structure.

Co-installers should not use the `ClassInstallReserved` field.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	<code>Setupapi.h</code> (include <code>Setupapi.h</code> )

## See also

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_DESTROYWIZARDDATA

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

Windows uses the DIF\_NEWDEVICEWIZARD\_XXX requests instead, such as  
[DIF\\_NEWDEVICEWIZARD\\_FINISHINSTALL](#).

# DIF\_DETECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_DETECT request directs an installer to detect non-PnP devices of a particular class and add the devices to the device information set. This request is used for non-PnP devices.

## When Sent

When the **Add Hardware Wizard** is detecting non-PnP devices.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#). There is a [device setup class](#) associated with the *DeviceInfoSet*.

### *DeviceInfoData*

None

### Device Installation Parameters

There are device installation parameters associated with the *DeviceInfoSet*.

### Class Installation Parameters

An [SP\\_DETECTDEVICE\\_PARAMS](#) structure is associated with the *DeviceInfoSet*. The parameters contain a callback routine that the class installer calls to indicate the progress of the detection operation.

## Installer Output

### *DeviceInfoSet*

An installer adds a device information element to the *DeviceInfoSet* for each device it detects, regardless of whether a device was previously detected and installed.

### Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoSet* or for new device information elements it creates.

## Installer Return Value

If a co-installer does not detect devices, it returns NO\_ERROR from its preprocessing pass. If a co-installer detects devices, it can do so during preprocessing or postprocessing and return NO\_ERROR or a Win32 error code.

If a class installer detects devices, it returns NO\_ERROR or an appropriate Win32 error code. If a class installer does not handle this DIF request, it returns ERROR\_DI\_DO\_DEFAULT.

## Default DIF Code Handler

None

## Installer Operation

In response to a DIF\_DETECT request an installer can detect devices of its setup class.

If an installer detects devices, it should do at least the following:

- Call the **DetectProgressNotify** callback routine in the **SP\_DETECTDEVICE\_PARAMS** class installation parameters, if detection will potentially take a noticeable amount of time.
- For each device the installer detects, it should:
  - Create a device information element (**SetupDiCreateDeviceInfo**).
  - Provide information for driver selection.

The installer can manually select the driver for the device or the installer can set the device's hardware ID that Windows will use to find an INF for the device. An installer sets the hardware ID by calling **SetupDiSetDeviceRegistryProperty** with a *Property* value of SPDRP\_HARDWAREID.

- Possibly set some device installation parameters.
- Return NO\_ERROR for successful detection or return a Win32 error code.

If one or more installers detects device(s) in response to this DIF code, Windows compares the list of detected devices to its current list of devices. If the installers detected a new device, Windows attempts to install the device. If the installers omitted a device that appears in Setup's list, Windows typically removes the device.

To detect non-PnP devices during GUI-mode setup, an installer must handle the **DIF\_FIRSTTIMESETUP** request. GUI-mode setup does not send a DIF\_DETECT request to the installer.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_DETECT](#)

[DIF\\_FIRSTTIMESETUP](#)

[SetupDiCreateDeviceInfo](#)

[SP\\_DETECTDEVICE\\_PARAMS](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_DETECTCANCEL

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_DETECTVERIFY

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_ENABLECLASS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_FINISHINSTALL\_ACTION

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_FINISHINSTALL\_ACTION request allows an installer to run finish-install actions in an interactive administrator context after all other device installation operations have completed.

## When Sent

In Windows 8 and later versions, finish-install actions do not automatically run as part of device installation. To complete a device finish-install action, a user must click on “Finish installing device software” in the Action Center to complete the installation.

For more information, see [Running Finish-Install Actions](#).

In Windows 7, the finish-install process runs only in the context of a user with administrator credentials at one of the following times:

- The next time that a user who has administrator credentials logs on while the device is attached.
- When the device is reattached.
- When the user selects Scan for hardware changes in Device Manager.

If a user is signed in without administrative privileges, Windows prompts the user for consent and credentials to run the finish-install actions in an administrator context.

## Who Handles

Class co-installer	Can handle
Device co-installer	Can handle
Class installer	Can handle

## Installer Input

### *DeviceInfoSet*

A handle to the [device information set](#) that contains the device being installed.

### *DeviceInfoData*

A pointer to an [SP\\_DEVINFO\\_DATA](#) structure that represents the device being installed.

### Device Installation Parameters

There are device installation parameters (a [SP\\_DEVINSTALL\\_PARAMS](#) structure) associated with *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer sets the DI\_NEEDREBOOT flag if a system restart is required to complete its finish-install actions.

## Installer Return Value

An installer returns one of the values that are listed in the following table.

RETURN VALUE	MEANING
ERROR_DI_DO_DEFAULT	Class installer: The installer has no finish-install actions, has successfully completed the finish-install actions, or has determined that it cannot ever successfully complete its finish install actions. Device installation should perform the default processing for the request.  Co-installer: Co-installers must not return this error code.
NO_ERROR	Class installer: A class installer should not return this error code. If a class installer returns this error code, device installation does not perform the default processing for the request.  Co-installer: The installer has no finish-install actions, has successfully completed the finish-install actions, or has determined that it cannot ever successfully complete its finish install actions.
Win32 error code	Class installer or co-installer: The installer encountered an error while processing a finish-install action, and device installation should attempt to complete the finish-install actions the next time the device is enumerated in the context of an administrator.

### Default DIF Code Handler

Windows 7 uses [SetupDiFinishInstallAction](#).

There is no default DIF Code Handler in Windows 8 and later versions, and [SetupDiFinishInstallAction](#) has been removed.

### Comments

Because device installation cannot determine from an ERROR\_DI\_DO\_DEFAULT return code or a NO\_ERROR return code whether a finish-install action actually succeeded, the installer should notify the user of the status of a finish-installer action.

For more information about finish-install actions, see [How Device Installation Processes Finish-Install Actions](#) and [Implementing Finish-Install Actions](#).

For general information about DIF codes, see [Handling DIF Codes](#) and [Calling Default DIF Code Handlers](#).

## Requirements

Version	Supported in Windows Vista through Windows 7.
Header	Setupapi.h (include Setupapi.h)

## See also

[SetupDiFinishInstallAction](#)

# DIF\_FIRSTTIMESETUP

11/2/2020 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request unless the vendor provides non-PnP devices that must be detected by the installer.

A DIF\_FIRSTTIMESETUP request directs an installer to perform any class-specific installation tasks that have to be completed during the initial installation of the operating system.

## When Sent

During GUI-mode setup.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the device information set. There is a [device setup class](#) associated with the *DeviceInfoSet*.

### *DeviceInfoData*

None

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoSet*.

### Class Installation Parameters

None

## Installer Output

### *DeviceInfoSet*

An installer adds a device information element to the *DeviceInfoSet* for each detected device it wants to have installed. An installer might also build a global class driver list.

### Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoSet* or for new device information elements it creates.

## Installer Return Value

A class co-installer can detect devices during preprocessing or postprocessing. Such a co-installer returns ERROR\_DI\_POSTPROCESSING\_REQUIRED (for postprocessing) and/or returns NO\_ERROR or a Win32 error code after its detection operations. If a co-installer does not detect devices, it returns NO\_ERROR from its preprocessing pass.

If a class installer detects devices, the installer returns NO\_ERROR or an appropriate Win32 error code. If a class installer does not handle this DIF request, the installer returns ERROR\_DI\_DO\_DEFAULT.

## Default DIF Code Handler

None

## Installer Operation

To detect non-PnP devices during GUI-mode setup, an installer must handle the DIF\_FIRSTTIMESETUP request. GUI-mode setup does not send a [DIF\\_DETECT](#) request to the installer.

GUI-mode setup sends a DIF\_FIRSTTIMESETUP request with an empty *DeviceInfoSet*. The installers can perform legacy detection of non-PnP devices and add them to the *DeviceInfoSet*. System-supplied installers can also handle this DIF request when migrating legacy device installations from Windows 9x/Me or Windows NT to Microsoft Windows 2000 and later versions of Windows.

An installer detects new devices of its setup class, based on registry information, by calling into a kernel-mode detection component, or by consulting *unattend.txt* information that is stored when a migration DLL ran during an operating system upgrade.

If an installer detects a non-PnP device, the installer should select a driver for the device as follows: create a device information element ([SetupDiCreateDeviceInfo](#)), set the SPDRP\_HARDWAREID property by calling [SetupDiSetDeviceRegistryProperty](#), call [SetupDiBuildDriverInfoList](#), and then call [SetupDiCallClassInstaller](#) to send a [DIF\\_SELECTBESTCOMPATDRV](#) request.

If one or more installers detect device(s) in response to this DIF code, GUI-mode setup attempts to install the device(s). GUI-mode setup attempts to install all devices in the list; if an installer returns a device that was previously configured, GUI-mode setup will install the device twice.

An installer must handle this DIF request silently. That is, without displaying UI to the user.

Installers should not perform tasks when they handle this DIF request that require the computer to be restarted. For example, a class installer should not set drivers to load at the next startup for the purpose of determining which drivers succeed after the restart.

To detect non-PnP devices during GUI-mode setup, an installer must handle this request. GUI-mode setup does not send a [DIF\\_DETECT](#) request.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_SELECTBESTCOMPATDRV](#)

[SetupDiBuildDriverInfoList](#)

[SetupDiCallClassInstaller](#)

[SetupDiCreateDeviceInfo](#)

[SetupDiSetDeviceRegistryProperty](#)

[SP\\_DEVINFO\\_DATA](#)

SP\_DEVINSTALL\_PARAMS

# DIF\_FOUNDDEVICE

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_INSTALLCLASSDRIVERS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_INSTALLDEVICE

12/1/2020 • 3 minutes to read • [Edit Online](#)

A DIF\_INSTALLDEVICE request allows an installer to perform tasks before and/or after the device is installed.

## When Sent

After selecting the driver, registering any device co-installers, and registering any device interfaces.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device to be installed.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure for the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoData*. For example, an installer might set the DI\_NEEDREBOOT flag or it might set the DI\_DONOTCALLCONFIGMG flag to prevent Windows from bringing the device online dynamically with its newly installed driver and settings.

## Installer Return Value

A co-installer typically returns NO\_ERROR or ERROR\_DI\_POSTPROCESSING\_REQUIRED. A co-installer might also return a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler. For more information about calling a default DIF code handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and

**SetupDiCallClassInstaller** will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiInstallDevice](#)

#### Installer Operation

In response to a DIF\_INSTALLDEVICE request an installer typically performs any final installation operations before the default handler installs the device. For example, an installer can check, and possibly modify, the upper-filter drivers and lower-filter drivers for the device that is listed in the registry.

Unless the DI\_NOFILECOPY flag is set in the device installation parameters, an installer that handles this DIF request should copy files that are required for the device, such as driver files and control panel files.

If the DI\_NOFILECOPY flag is clear but the DI\_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

A co-installer can handle this DIF request in its preprocessing pass and/or in its postprocessing pass. In its preprocessing pass, a co-installer performs any operations that must occur before Windows loads the drivers and starts the device.

In its postprocessing pass, the device is up and running unless the DI\_NEEDREBOOT flag was set. If this flag is set, Windows could not bring the device online dynamically.

If the installer returns a Win32 error code, Windows abandons the installation.

If Windows cannot locate an INF file for a new device, it sends DIF\_INSTALLDEVICE in an attempt to install a *null driver*. The default handler ([SetupDiInstallDevice](#)) checks whether the device either supports *raw mode* or is a non-PnP device (reported by [IoReportDetectedDevice](#)). In the latter case, Windows installs a null driver for the device.

If this attempt fails, Windows sends DIF\_INSTALLDEVICE again, this time with the DI\_FLAGSEX\_SETFAILEDINSTALL flag set in the [SP\\_DEVINSTALL\\_PARAMS](#) structure. In this case, the default handler just sets the FAILEDINSTALL flag in the device's **ConfigFlags** registry value. If the DI\_FLAGSEX\_SETFAILEDINSTALL flag is set, class installers must return NO\_ERROR or ERROR\_DI\_DO\_DEFAULT and co-installers must return NO\_ERROR.

For more information about DIF codes, see [Handling DIF Codes](#).

#### Calling the Default Handler [SetupDiInstallDevice](#)

For general information about when and how to call a [SetupDiInstallDevice](#), see [Calling Default DIF Code Handlers](#).

In the rare situation where the class installer must perform operations after all [SetupDiInstallDevice](#) operations, except for starting a device, have completed, the class installer must:

1. Perform operations that must be done before calling [SetupDiInstallDevice](#).
2. Set the DI\_DONOTCALLCONFIGMGR flag in the [SP\\_DEVINSTALL\\_PARAMS.Flags](#) member for the device. If this flag is set, [SetupDiInstallDevice](#) performs all default installation operations except for starting the device.
3. Call [SetupDiInstallDevice](#) to perform all default installation operations except for starting the device.
4. Perform the operations that must be done after all default installation operations, except for starting the device, have completed.
5. Call [SetupDiRestartDevices](#) to start the device.
6. Return NO\_ERROR if the class installer successfully completed the installation operation or return a Win32 error if the installation operation failed.

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_INSTALLDEVICEFILES](#)

[SetupDiInstallDevice](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_INSTALLDEVICEFILES

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_INSTALLDEVICEFILES request allows an installer to participate in copying the files to support a device or to make a list of the files for a device. The device files include files for the selected driver, any device interfaces, and any co-installers.

## When Sent

The [system-provided device installation components](#) send this DIF request for a variety of reasons. Some device installation components send this DIF request before DIF\_REGISTER\_COINSTALLERS, DIF\_INSTALLINTERFACES, and DIF\_INSTALL\_DEVICE to ensure that all the relevant files can be copied before proceeding with the installation. Some device installation components omit this DIF request and expect the files to be copied during the handling of those three DIF requests. In addition, some device installation components send this DIF request to retrieve the list of the files associated with a device.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device whose supporting files are to be copied.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

If the DI\_NOVCP flag is set, the device installation parameters contain a valid [FileQueue](#) handle and installers that handle this DIF request add their file operations to this queue and do not commit the queue.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer can modify the [FileQueue](#), if there is one.

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class

installer should return NO\_ERROR and `SetupDiCallClassInstaller` will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and `SetupDiCallClassInstaller` will not subsequently call the default handler.

#### **Default DIF Code Handler**

#### [\*\*SetupDiInstallDriverFiles\*\*](#)

#### **Installer Operation**

In response to a DIF\_INSTALLDEVICEFILES request an installer specifies any necessary file operations. For example, an installer can specify an additional file to be copied that is required for device installation. If the DI\_NOVCP flag is set, an installer specifies file operations by adding them to the `FileQueue` in the device installation parameters. See the Microsoft Windows SDK for information about how to use file queues and for reference pages on file-queuing functions such as `SetupInstallFilesFromInfSection`.

If this DIF request is sent during device installation, and the installer returns a Microsoft Win32 error code, Windows stops the installation.

If a [system-provided device installation component](#) sends this DIF request to retrieve a list of the files associated with a device, the component retrieves the file queue but does not commit the queue.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[\*\*SetupDiInstallDriverFiles\*\*](#)

[\*\*SP\\_DEVINFO\\_DATA\*\*](#)

[\*\*SP\\_DEVINSTALL\\_PARAMS\*\*](#)

# DIF\_INSTALLINTERFACES

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_INSTALLINTERFACES request allows an installer to participate in the registration of the device interfaces for a device.

## When Sent

After registering device co-installers but before completing device installation.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [\*\*SP\\_DEVINFO\\_DATA\*\*](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([\*\*SP\\_DEVINSTALL\\_PARAMS\*\*](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer might modify the device installation parameters, but not usually for this DIF request.

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [\*\*SetupDiCallClassInstaller\*\*](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [\*\*SetupDiCallClassInstaller\*\*](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and [\*\*SetupDiCallClassInstaller\*\*](#) will not subsequently call the default handler.

## Default DIF Code Handler

## [SetupDiInstallDeviceInterfaces](#)

### **Installer Operation**

In response to a DIF\_INSTALLINTERFACES request an installer might register a device interface programmatically instead of having the interface registered through the INF file. Typically, vendor-supplied installers do not handle this DIF request.

Unless the DI\_NOFILECOPY flag is set, an installer that handles this DIF request should copy files that are required for the device interface(s).

If the DI\_NOFILECOPY flag is clear but the DI\_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

If an installer registers a device interface, a kernel-mode component for the device (for example, a driver) must call [IoSetDeviceInterfaceState](#) to enable the interface.

If the installer returns a Win32 error code, Windows stops the installation.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SetupDiInstallDeviceInterfaces](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_INSTALLWIZARD

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

For PnP devices, Windows uses the DIF\_NEWDEVICEWIZARD\_XXX requests instead, such as [DIF\\_NEWDEVICEWIZARD\\_FINISHINSTALL](#).

# DIF\_MOVEDevice

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

# DIF\_NEWDEVICEWIZARD\_FINISHINSTALL

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_NEWDEVICEWIZARD\_FINISHINSTALL request allows an installer to supply finish-install wizard pages that Windows displays to the user after a device is installed but before Windows displays the standard finish page. Windows sends this request when it installs Plug and Play (PnP) devices and when an administrator uses the **Add Hardware Wizard** to install non-PnP devices.

## When Sent

After Windows installs a device (on successful completion of [DIF\\_INSTALLDEVICE](#) processing), but before it displays the Finish wizard page.

## Who Handles

Class co-installer	Can handle
Device co-installer	Can handle
Class installer	Can handle

## Installer Input

### *DeviceInfoSet*

A handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

A pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_NEWDDEVICEWIZARD\\_DATA](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Device Installation Parameters

An installer can modify the flags in the device installation parameters.

### Class Installation Parameters

An installer can modify the SP\_NEWDDEVICEWIZARD\_DATA structure to supply finish-install wizard pages.

## Installer Return Value

If a co-installer does not handle this DIF request, the co-installer returns NO\_ERROR from its preprocessing pass. If a co-installer handles this request, the co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if the installer successfully supplies pages. Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_FINISHINSTALL\\_ACTION](#)

[DIF\\_INSTALLDEVICE](#)

[SetupDiChangeState](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_NEWDEVICEWIZARD\\_DATA](#)

# DIF\_NEWDEVICEWIZARD\_POSTANALYZE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_NEWDEVICEWIZARD\_POSTANALYZE request allows an installer to supply wizard pages that Windows displays to the user after the device node (*devnode*) is registered but before Windows installs the drivers for the device. This request is only used during manual installation of non-PnP devices.

## When Sent

After Windows registers the device, which makes the devnode "live," but before Windows installs the drivers for the device.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_NEWDEVICEWIZARD\\_DATA](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Device Installation Parameters

An installer can modify the flags in the device installation parameters. Windows does not check the flags upon completion of this DIF request. However, it will check them later in the installation process.

### Class Installation Parameters

An installer can modify the [SP\\_NEWDEVICEWIZARD\\_DATA](#) to supply custom page(s).

## Installer Return Value

If a co-installer does not handle this DIF request it returns NO\_ERROR from its preprocessing pass. If a co-installer handles this request it can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

A DIF\_NEWDEVICEWIZARD\_POSTANALYZE request allows an installer to supply wizard pages that Windows displays to the user after the devnode is registered but before Windows installs the drivers for the device. This request is only used during manual installation of non-PnP devices.

If an installer adds custom postanalyze page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX\_INSTALLWIZARD\_DYNAPAGES.

After the user clicks **Next** on a custom page, Windows installs the drivers for the device and the PnP manager starts the device. A postanalyze wizard page is the last opportunity for an installer to do work before the drivers are loaded and the device is started.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Microsoft Windows SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_NEWDEVICEWIZARD\\_PREANALYZE](#)

[DIF\\_NEWDEVICEWIZARD\\_PRESELECT](#)

[DIF\\_NEWDEVICEWIZARD\\_SELECT](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEWINSTALL\\_PARAMS](#)

[SP\\_NEWDEVICEWIZARD\\_DATA](#)

# DIF\_NEWDEVICEWIZARD\_PREANALYZE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_NEWDEVICEWIZARD\_PREANALYZE request allows an installer to supply wizard pages that Windows displays to the user before it displays the analyze page. This request is only used during manual installation of non-PnP devices.

## When Sent

After the user has selected a driver, but before Windows registers the device that makes the device node (*devnode*) "live."

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [\*\*SP\\_DEVINFO\\_DATA\*\*](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([\*\*SP\\_DEVINSTALL\\_PARAMS\*\*](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [\*\*SP\\_NEWDEVICEWIZARD\\_DATA\*\*](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Device Installation Parameters

An installer can modify the flags in the device installation parameters. Windows does not check the flags upon completion of this DIF request. However, it checks them later in the installation process.

### Class Installation Parameters

An installer can modify the [\*\*SP\\_NEWDEVICEWIZARD\\_DATA\*\*](#) to supply custom wizard page(s).

## Installer Return Value

If a co-installer does not handle this DIF request it returns NO\_ERROR from its preprocessing pass. If a co-installer handles this request it can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

A DIF\_NEWDEVICEWIZARD\_PREANALYZE request allows an installer to supply wizard pages that Windows displays to the user before it displays the analyze page. These pages can be thought of as "postselect" pages. This request is only used during manual installation of non-PnP devices.

An installer might use a custom preanalyze page, for example, to choose a COM port after a modem device is selected.

If an installer adds custom preselect page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX\_INSTALLWIZARD\_DYNAPAGES.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Microsoft Windows SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_NEWDEVICEWIZARD\\_PRESELECT](#)

[DIF\\_NEWDEVICEWIZARD\\_POSTANALYZE](#)

[DIF\\_NEWDEVICEWIZARD\\_SELECT](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVESTALL\\_PARAMS](#)

[SP\\_NEWDEVICEWIZARD\\_DATA](#)

# DIF\_NEWDEVICEWIZARD\_PRESELECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_NEWDEVICEWIZARD\_PRESELECT request allows an installer to supply wizard pages that Windows displays to the user before it displays the select-driver page. This request is only used during manual installation of non-PnP devices.

## When Sent

After the user has selected the class for the device but before Windows displays the "Select a Device Driver" page.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_NEWDEVICEWIZARD\\_DATA](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Device Installation Parameters

An installer can modify the flags in the device installation parameters. Windows does not check the flags upon completion of this DIF request. However, it checks them later in the installation process.

### Class Installation Parameters

An installer can modify the [SP\\_NEWDEVICEWIZARD\\_DATA](#) to supply custom page(s).

## Installer Return Value

If a co-installer does not handle this DIF request it returns NO\_ERROR from its preprocessing pass. If a co-installer handles this request it can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

A DIF\_NEWDEVICEWIZARD\_PRESELECT request allows an installer to supply wizard pages that Windows displays to the user before it displays the select-driver page. This request is only used during manual installation of non-PnP devices.

If an installer adds custom preselect page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX\_INSTALLWIZARD\_DYNAPAGES.

A co-installer can add custom pages in its preprocessing pass and/or in its postprocessing pass. If it adds page(s) in its preprocessing pass, those pages are displayed before any page(s) supplied by the class installer.

If one or more installers add custom preselect pages, Windows displays the pages before the "Select a Device Driver" page. However, if the user presses "Back" on the select-driver page, Windows skips the custom preselect pages and goes back to the "Hardware Type" class-selection page.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Microsoft Windows SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_NEWDEVICEWIZARD\\_PREANALYZE](#)

[DIF\\_NEWDEVICEWIZARD\\_POSTANALYZE](#)

[DIF\\_NEWDEVICEWIZARD\\_SELECT](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_NEWDEVICEWIZARD\\_DATA](#)

# DIF\_NEWDEVICEWIZARD\_SELECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_NEWDEVICEWIZARD\_SELECT request allows an installer to supply custom wizard page(s) that replace the standard select-driver page. This request is only used during manual installation of non-PnP devices.

## When Sent

Immediately before Windows displays the "Select a Device Driver" page.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_NEWDEVICEWIZARD\\_DATA](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Device Installation Parameters

An installer can modify the flags in the device installation parameters. Windows does not check the flags upon completion of this DIF request. However, it checks them later in the installation process.

### Class Installation Parameters

An installer can modify the [SP\\_NEWDEVICEWIZARD\\_DATA](#) to supply custom page(s).

## Installer Return Value

If a co-installer does not handle this DIF request it returns NO\_ERROR from its preprocessing pass. If a co-installer handles this request it can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

A DIF\_NEWDEVICEWIZARD\_SELECT request allows an installer to supply custom wizard page(s) that replace the

standard select-driver page. This request is only used during manual installation of non-PnP devices.

An installer responds to this DIF request to completely replace the standard select-driver wizard page. If, instead, the installer only has to modify the standard page or modify the list of drivers from which to choose, the installer should do so in response to the [DIF\\_SELECTDEVICE](#) request.

A co-installer should add custom page(s) in its postprocessing pass and only if the class installer did not add custom page(s). If the class installer added page(s), the co-installer should not. Otherwise, the user might be asked to choose a driver twice.

If an installer supplies a custom select page, the installer must set the selected driver. In the installer's code that supports the wizard page, after the user clicks **Next**, the installer must call [SetupDiSetSelectedDriver](#).

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Microsoft Windows SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_NEWDEVICEWIZARD\\_PREANALYZE](#)

[DIF\\_NEWDEVICEWIZARD\\_PRESELECT](#)

[DIF\\_NEWDEVICEWIZARD\\_POSTANALYZE](#)

[DIF\\_SELECTDEVICE](#)

[SetupDiSetSelectedDevice](#)

[SetupDiSetSelectedDriver](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_NEWDEVICEWIZARD\\_DATA](#)

# DIF\_POWERMESSAGEWAKE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_POWERMESSAGEWAKE request allows an installer to supply custom text that Windows displays on the power management properties page of the device properties.

## When Sent

When a user clicks on a menu item or tab to display the properties of a device.

Windows only sends this DIF request if the drivers for the device support power management. Otherwise, Windows does not display any power properties for the device.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_POWERMESSAGEWAKE\\_PARAMS](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Class Installation Parameters

An installer can modify the [SP\\_POWERMESSAGEWAKE\\_PARAMS](#) to supply custom text for a device's power properties page.

## Installer Return Value

A co-installer typically returns NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

A class installer returns NO\_ERROR if it successfully supplies power properties text. Otherwise, a class installer returns ERROR\_DI\_DO\_DEFAULT or a Win32 error code.

## Default DIF Code Handler

None

## Installer Operation

A DIF\_POWERMESSAGEWAKE request allows an installer to supply text that Windows displays on the power properties page for a device.

If a co-installer supplies power-properties text, it should do so in its postprocessing phase. A co-installer should be careful when overwriting any power-properties text supplied by an installer that handled the request before the co-installer.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_POWERMESSAGEWAKE\\_PARAMS](#)

# DIF\_PROPERTIES

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

To supply custom property pages for a device, an installer handles the [DIF\\_ADDPROPERTYPAGE\\_ADVANCED](#) request.

# DIF\_PROPERTYCHANGE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_PROPERTYCHANGE request notifies the installer that the device's properties are changing. The device is being enabled, disabled, started, stopped, or some item on a property page has changed. This DIF request gives the installer an opportunity to participate in the change.

## When Sent

When a device is being enabled, disabled, restarted, stopped, or its properties have changed.

For example, Windows sends this request when a property-page provider sets the DI\_FLAGSEX\_PROPCHANGE\_PENDING flag in the **FlagsEx** field of the **SP\_DEVINSTALL\_PARAMS** structure for the device.

For more information about detecting when a device is started for the first time or subsequently restarted, see the Installer Operation section.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the **device information set** that contains the device.

### *DeviceInfoData*

Supplies a pointer to an **SP\_DEVINFO\_DATA** structure for the device in the device information set.

### Device Installation Parameters

There are device installation parameters (**SP\_DEVINSTALL\_PARAMS**) associated with the *DeviceInfoData*.

### Class Installation Parameters

An **SP\_PROPCHANGE\_PARAMS** structure is associated with the *DeviceInfoData*.

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and **SetupDiCallClassInstaller** should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and **SetupDiCallClassInstaller** will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and `SetupDiCallClassInstaller` will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiChangeState](#)

#### Installer Operation

In response to a `DIF_PROPERTYCHANGE` request an installer can participate in the property-change operation. The class installation parameters (`SP_PROPCHANGE_PARAMS`) indicate which change is taking place.

A property change might require a system restart. For information about how to restart the system, see [SetupDiCallClassInstaller](#).

When Windows sends a `DIF_INSTALLDEVICE` request to install a device for the first time, Windows starts the device but does not send a `DIF_PROPERTYCHANGE` request as part of the installation. If a custom installation operation must be performed when a device is started for the first time and whenever the device is subsequently restarted, an installer or a co-installer should handle the `DIF_INSTALLDEVICE` request that starts the device for the first time and a `DIF_PROPERTYCHANGE` request that indicates that the state change action is that the device is being started.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	<code>Setupapi.h</code> (include <code>Setupapi.h</code> )

## See also

[SetupDiChangeState](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEINSTALL\\_PARAMS](#)

[SP\\_PROPCHANGE\\_PARAMS](#)

# DIF\_REGISTER\_COINSTALLERS

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_REGISTER\_COINSTALLERS request allows an installer to participate in the registration of device co-installers.

## When Sent

Before completing device installation.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device for which co-installers are to be registered.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and [SetupDiCallClassInstaller](#) will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiRegisterCoDeviceInstallers](#)

#### Installer Operation

In response to a DIF\_REGISTER\_COINSTALLERS request an installer might modify the list of co-installers for the device. For example, an installer might programmatically register or remove a device-specific co-installer for the device that is based on the analysis of the device.

Unless the DI\_NOFILECOPY flag is set, an installer that handles this DIF request should copy files that are required for the co-installer(s).

If the DI\_NOFILECOPY flag is clear but the DI\_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

If the installer returns a Win32 error code, Windows stops the installation.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

### [SetupDiRegisterCoDeviceInstallers](#)

### [SP\\_DEVINFO\\_DATA](#)

### [SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_REGISTERDEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DIF\_REGISTERDEVICE request allows an installer to participate in registering a newly created device instance with the PnP manager. Windows sends this DIF request for non-PnP devices.

## When Sent

When an installer reports a previously unknown device in response to a [DIF\\_DETECT](#) request. Windows sends this DIF request in the analyze phase of the Add Hardware Wizard before it installs the device. Windows also sends this request during non-PnP detection.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR or a Win32 error code. A co-installer should not return ERROR\_DI\_POSTPROCESSING\_REQUIRED for this DIF request.

If an installer determines that the device is a duplicate it returns ERROR\_DUPLICATE\_FOUND.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and **SetupDiCallClassInstaller** will not subsequently call the default handler.

If the installer determines that the device is a duplicate, the installer returns ERROR\_DUPLICATE\_FOUND.

### Default DIF Code Handler

#### [SetupDiRegisterDeviceInfo](#)

### Installer Operation

A *device installation application* typically sends this DIF request to register a non-PnP device with the PnP manager. Starting with Microsoft Windows 2000, non-PnP devices must be registered before they can be installed.

An installer typically handles this DIF request to do duplicate detection. Such an installer typically calls the default handler ([SetupDiRegisterDeviceInfo](#)) and specifies its detection routine. If the registration is successful and the installer determines that the device is not a duplicate, the installer returns NO\_ERROR.

A co-installer should perform any operations to handle this DIF request in its preprocessing pass. When the co-installer is called for postprocessing, the device instance has already been registered by either the class installer or the default handler.

If an installer returns an error for this DIF code, typically ERROR\_DUPLICATE\_FOUND, Windows deletes the device from the device information set.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

### [DIF\\_DETECT](#)

### [SetupDiRegisterDeviceInfo](#)

### [SP\\_DEVINFO\\_DATA](#)

### [SP\\_DEWINSTALL\\_PARAMS](#)

# DIF\_REMOVE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_REMOVE request notifies an installer that Windows is about to remove a device and gives the installer an opportunity to prepare for the removal.

## When Sent

When a user removes a device in Device Manager.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device to be removed.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure for the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_REMOVEDevice\\_PARAMS](#) structure might be associated with the *DeviceInfoData*.

There are no class installation parameters for the request if the DI\_CLASSINSTALLPARAMS flag is clear in the [SP\\_DEVINSTALL\\_PARAMS](#). In this case, no hardware profile is specified and the device is to be removed from the system as a whole.

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and `SetupDiCallClassInstaller` will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiRemoveDevice](#)

#### Installer Operation

In response to a `DIF_REMOVE` request, an installer typically performs some clean-up operations. In this case, a co-installer returns `NO_ERROR` and a class installer returns `ERROR_DI_DO_DEFAULT`.

If an installer determines that the device should not be removed, the installer fails the DIF request by returning a Win32 error code. If the `DI QUIETINSTALL` flag is clear, the installer should display a message to the user explaining why the device is not being removed.

Co-installers must not attempt to remove the device themselves by calling `SetupDiRemoveDevice`. Co-installers typically handle this request in postprocessing, after the device is successfully removed.

If a co-installer has to delete information in the registry, for example, the co-installer should do so in postprocessing and only if the previous installers succeeded the removal request. In its preprocessing pass, the co-installer should store the registry information in its context parameter and return `ERROR_DI_POSTPROCESSING_REQUIRED` to request postprocessing. When Windows calls the co-installer for postprocessing of this DIF request, the co-installer should check that the DIF status is `NO_ERROR` and then delete the registry information. If a co-installer deletes registry information in its preprocessing pass and the class installer (or another co-installer) fails the `DIF_REMOVE`, the co-installer could leave the device in an unpredictable state.

Installers should not delete files when handling this DIF request, in case the files are in use by another device.

Windows sends this DIF request before it initiates PnP query-remove and remove processing.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	<code>Setupapi.h</code> (include <code>Setupapi.h</code> )

## See also

[SetupDiRemoveDevice](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_REMOVEDevice\\_PARAMS](#)

# DIF\_SELECTBESTCOMPATDRV

11/2/2020 • 3 minutes to read • [Edit Online](#)

## NOTE

This request was deprecated in Windows 10 version 1703 (Redstone 2). In more recent versions of Windows, this callback is no longer invoked.

A DIF\_SELECTBESTCOMPATDRV request allows an installer to select the best driver from the device information element's compatible driver list.

## When Sent

When the operating system is preparing to install a new PnP device or is performing a change-driver operation on a PnP device.

This DIF request is typically used during a PnP configuration. If a device is being manually installed, Windows sends a [DIF\\_SELECTDEVICE](#) request.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

None

## Installer Output

### Device Installation Parameters

An installer can modify the device installation parameters. However, they typically do not when handling this DIF request.

### *DeviceInfoData*

As a side effect, an installer can modify the driver list associated with the *DeviceInfoData*, in particular, the [SP\\_DRVINSTALL\\_PARAMS](#).

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and [SetupDiCallClassInstaller](#) will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiSelectBestCompatDrv](#)

#### Installer Operation

An installer handles this DIF request to participate in selecting a driver for a PnP device. An installer typically responds to this DIF request in one of the following ways:

- Do nothing.

If an installer has no special selection requirements, it does nothing in response to this DIF request. A class installer returns ERROR\_DI\_DO\_DEFAULT and a co-installer returns NO\_ERROR.

- Modify the parameters of one or more drivers in the driver list.

For example, an installer might remove a driver from consideration for the device by marking it DNF\_BAD\_DRIVER. An installer modifies driver parameters by following these steps:

1. Get the information about the first driver in the list by calling [SetupDiEnumDriverInfo](#) and [SetupDiGetDriverInstallParams](#). If appropriate, modify the driver parameters and apply the change by calling [SetupDiSetDriverInstallParams](#).

If a driver is a worst-case choice, set the driver's rank to 0xFFFF or higher in the driver install parameters. See [How Windows Selects Drivers](#) for more information.

2. Repeat the previous step until you have processed all the drivers in the list. Make sure that you increment the *MemberIndex* parameter to [SetupDiEnumDriverInfo](#) as described in the reference page for that function.

After a class installer modifies the driver list, it returns ERROR\_DI\_DO\_DEFAULT. If a co-installer modifies the driver list, it should do so in preprocessing and return NO\_ERROR.

- Select the best driver for the device.

This action is less common, but an installer might choose the best driver for the device. Such an installer would examine the data for each driver, choose a driver, and call [SetupDiSetSelectedDriver](#) to set the driver. After an installer sets the selected driver, it returns NO\_ERROR.

If a co-installer selects a driver, it should do so in postprocessing.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SetupDiSelectBestCompatDrv](#)

[SetupDiSetSelectedDriver](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

# DIF\_SELECTCLASSDRIVERS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

# DIF\_SELECTDEVICE

11/2/2020 • 5 minutes to read • [Edit Online](#)

A DIF\_SELECTDEVICE request allows an installer to participate in selecting the driver for a device.

## When Sent

When choosing a driver for a newly enumerated device or a new driver for an existing device (change driver). For example, when a user selects Add/Remove Hardware and selects the modem class. Or, a user inserts a PnP device and selects "Choose a Driver From a List" in the Found New Hardware Wizard.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Does not handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device for which a driver is to be selected. There is a [device setup class](#) associated with the *DeviceInfoSet*.

### *DeviceInfoData*

Optionally supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

If *DeviceInfoData* is **NULL**, this request is to select a driver for the [device setup class](#) associated with the *DeviceInfoSet*.

### Device Installation Parameters

If *DeviceInfoData* is not **NULL**, there are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*. If *DeviceInfoData* is **NULL**, there are device installation parameters associated with the *DeviceInfoSet*.

Of particular interest is the **DriverPath**, which contains the location of INF(s) to use when building the driver list.

### Class Installation Parameters

An [SP\\_SELECTDEVICE\\_PARAMS](#) structure is associated with the *DeviceInfoData* if *DeviceInfoData* is not **NULL**. Otherwise, the class installation parameters are associated with the device information set as a whole.

## Installer Output

### Device Installation Parameters

An installer can modify the device installation parameters. However, it should not modify the **DriverPath** field.

### Class Installation Parameters

An installer can modify the [SP\\_SELECTDEVICE\\_PARAMS](#). For example, an installer might specify a title and/or instructions for Windows to use in the dialog box that asks the user to select a driver.

If an installer sets new select-device parameters, versus modifying parameters set by a previous installer, the

installer must zero the fields that it does not set.

### Installer Return Value

If a co-installer does nothing for this DIF code, it returns NO\_ERROR from its preprocessing pass. If a co-installer handles this DIF code, it should do so in its preprocessing pass and return NO\_ERROR or a Win32 error code. By the time that a co-installer is called for postprocessing, the driver has already been selected.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and [SetupDiCallClassInstaller](#) will not subsequently call the default handler.

A class installer returns ERROR\_DI\_BAD\_PATH if the **DriverPath** member of the corresponding [SP\\_DEVINSTALL\\_PARAMS](#) structure is not equal to NULL, but there are no valid drivers at the specified path location. This can occur if there are no drivers at the path location or if there are drivers, but the **Flags** member of the [SP\\_DRVINSTALL\\_PARAMS](#) structure of each driver was set with the DN\_BAD\_DRIVER flag. In response to this error code, Windows displays an error to the user.

### Default DIF Code Handler

#### [SetupDiSelectDevice](#)

##### Installer Operation

In response to a DIF\_SELECTDEVICE request, an installer performs any selection operations required for its device or device class, besides what the default handler does. An installer typically responds to this DIF request in one of the following ways:

- Do nothing.

If an installer has no special selection requirements, it does nothing in response to this DIF code. A class installer returns ERROR\_DI\_DO\_DEFAULT and a co-installer returns NO\_ERROR.

- Supply select strings that Windows will display in the selection UI.

An installer can supply select strings in the class installation parameters ([SP\\_SELECTDEVICE\\_PARAMS](#)). For example, an installer can modify the **Instructions** or the window header **Title**.

A class installer should not supply select strings if a co-installer already supplied select strings. The co-installer probably has more relevant information.

If an installer modifies the [SP\\_SELECTDEVICE\\_PARAMS](#), the installer must also set the DI\_USECI\_SELECTSTRINGS flag in the [SP\\_DEVINSTALL\\_PARAMS](#).

If an installer successfully supplies select strings, Windows still has to call the default handler. Therefore, in this case, a co-installer returns NO\_ERROR and a class installer returns ERROR\_DI\_DO\_DEFAULT.

- Modify the device installation parameters.

An installer can modify the device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)). For example, an installer might set the DI\_SHOWOEM flag to have Windows display the **Have Disk** button.

If a class installer successfully modifies the device installation parameters, the class installer returns **ERROR\_DI\_DO\_DEFAULT**.

- Modify the list of drivers from which the user can select.

This action is less common, but possible. An installer that modifies the driver list might, or might not, also supply select strings.

An installer that modifies the driver list typically marks driver(s) that are inappropriate for the device. An installer marks such drivers with the flag **DNF\_BAD\_DRIVER**. Windows omits these drivers from the list it displays to the user.

An installer marks bad drivers by following these steps:

1. Build the driver list by calling [SetupDiBuildDriverInfoList](#) with a *DriverType* of **SPDIT\_CLASSDRIVER**.
2. Get the information about the first driver in the list by calling [SetupDiEnumDriverInfo](#) and [SetupDiGetDriverInstallParams](#). If the driver is not appropriate for the device, set the **DNF\_BAD\_DRIVER** flag in the **Flags** field of the parameters. Apply the change to the parameters by calling [SetupDiSetDriverInstallParams](#).
3. Repeat the previous step until you have processed all the drivers in the list. Make sure that you increment the *MemberIndex* parameter to [SetupDiEnumDriverInfo](#) as described in the reference page for that function.

An installer might set the **DNF\_BAD\_DRIVER** flag for one or more drivers in the driver list, but an installer must not clear that flag.

If one or more installers successfully modify the driver list, Windows still has to call the default handler. Therefore, in this case, a co-installer returns **NO\_ERROR** and a class installer returns **ERROR\_DI\_DO\_DEFAULT**.

- Display its own driver-selection user interface and set the selected driver.

Only a class installer can display its own driver-selection user interface; co-installers must not. For example, a class installer might display pictures instead of textual lists.

If the class installer successfully sets the selected driver, the class installer returns **NO\_ERROR** and Windows does not call the default handler and therefore does not display the default selection interface.

If the **DI\_ENUMSINGLEINF** flag is set in the device installation parameters, the **DriverPath** is a path of a single INF file instead of a path of a directory. An installer must use only that single INF to build the driver list.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[DIF\\_NEWDEVICEWIZARD\\_SELECT](#)

[SetupDiSelectDevice](#)

[SP\\_DEVINFO\\_DATA](#)

`SP_DEVINSTALL_PARAMS`

`SP_SELECTDEVICE_PARAMS`

# DIF\_TROUBLESHOOTER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The DIF\_TROUBLESHOOTER request allows an installer to start a troubleshooter for a device or to return CHM and HTM troubleshooter files for Windows to start.

**Note** This DIF code is only supported on Windows Server 2003, Windows XP, and Microsoft Windows 2000.

## When Sent

When a user clicks the "Troubleshooter" button for a device in Device Manager.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_TROUBLESHOOTER\\_PARAMS](#) structure is associated with the *DeviceInfoData*.

## Installer Output

### Class Installation Parameters

An installer might modify the [SP\\_TROUBLESHOOTER\\_PARAMS](#), setting a CHM or HTML file.

## Installer Return Value

If a co-installer does not handle this request, it returns NO\_ERROR from its preprocessing pass.

If a co-installer handles this request, it does so in its postprocessing pass. If the co-installer supplies CHM and HTML files, it propagates the status it received (probably ERROR\_DI\_DO\_DEFAULT). If the co-installer runs a troubleshooter and fixes the problem, the co-installer returns NO\_ERROR. If the co-installer runs a troubleshooter but does not fix the problem, it propagates the status it received (ERROR\_DI\_DO\_DEFAULT).

If a class installer supplies a CHM file and an HTML file, or the class installer runs a troubleshooter but does not fix the problem, the class installer returns ERROR\_DI\_DO\_DEFAULT. Windows will subsequently call the default handler.

If a class installer starts its own troubleshooter and fixes the problem, the class installer returns NO\_ERROR. Windows will not subsequently call the default handler.

If the class installer encounters an error, the installer returns an appropriate Win32 error code. Windows will not subsequently call the default handler.

## Default DIF Code Handler

None

There is no default handler for DIF\_TROUBLESHOOTER, but the operating system provides default troubleshooters that attempt to resolve device problems if there are no installer-supplied troubleshooters.

## Installer Operation

An installer calls [CM\\_Get\\_DevNode\\_Status](#) to get the device status and the CM problem code. Depending on the problem, an installer might provide a troubleshooter, a help file, or nothing. A troubleshooter can possibly resolve a problem with a device. If a troubleshooter resolves the problem, it should call [SetupDiCallClassInstaller](#) to send a DIF\_PROPERTYCHANGE request of type DICS\_PROPCHANGE. If an installer does not supply a troubleshooter for a device, it might supply a help file of problem-solving suggestions for the user.

If no installer runs its own troubleshooter, Windows runs HTML Help to display information to the user. If an installer supplied a CHM file in the class installation parameters, Windows displays that file. Otherwise, Windows displays system-supplied troubleshooting information.

The class installation parameters contain at most one **ChmFile** and **HtmlTroubleShooter** pair. If more than one installer specifies these values, Windows uses the values set by the last installer that handled the DIF request.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Windows Server 2003, Windows XP, and Microsoft Windows 2000.
Header	Setupapi.h (include Setupapi.h)

## See also

[CM\\_Get\\_DevNode\\_Status](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_TROUBLESHOOTER\\_PARAMS](#)

# DIF\_UNREMOVE

11/2/2020 • 2 minutes to read • [Edit Online](#)

A DIF\_UNREMOVE request notifies the installer that Windows is about to reinstate a device in a given hardware profile and gives the installer an opportunity to participate in the operation. Windows only sends this request for non-PnP devices.

## When Sent

When a root-enumerated, non-PnP device is reinstated to a hardware profile.

## Who Handles

Class Co-installer	Can handle
Device Co-installer	Can handle
Class Installer	Can handle

## Installer Input

### *DeviceInfoSet*

Supplies a handle to the [device information set](#) that contains the device.

### *DeviceInfoData*

Supplies a pointer to an [SP\\_DEVINFO\\_DATA](#) structure that identifies the device in the device information set.

### Device Installation Parameters

There are device installation parameters ([SP\\_DEVINSTALL\\_PARAMS](#)) associated with the *DeviceInfoData*.

### Class Installation Parameters

An [SP\\_UNREMOVEDevice\\_PARAMS](#) structure is associated with the *DeviceInfoData*. The **Scope** field must be set to DI\_UNREMOVEDevice\_CONFIGSPECIFIC and a hardware profile must be specified in the **HwProfile** field.

## Installer Output

None

## Installer Return Value

A co-installer can return NO\_ERROR, ERROR\_DI\_POSTPROCESSING\_REQUIRED, or a Win32 error code.

If a class installer successfully handles this request and [SetupDiCallClassInstaller](#) should subsequently call the default handler, the class installer returns ERROR\_DI\_DO\_DEFAULT.

If the class installer successfully handles this request, including directly calling the default handler, the class installer should return NO\_ERROR and [SetupDiCallClassInstaller](#) will not subsequently call the default handler again.

**Note** The class installer can directly call the default handler, but the class installer should never attempt to supersede the operations of the default handler.

For more information about calling the default handler, see [Calling Default DIF Code Handlers](#).

If the class installer encounters an error, the installer should return an appropriate Win32 error code and [SetupDiCallClassInstaller](#) will not subsequently call the default handler.

## Default DIF Code Handler

### [SetupDiUnremoveDevice](#)

#### Installer Operation

"Unremoving" a device basically means that Windows clears a flag that previously marked a device as "not present" in a particular hardware profile.

For more information about DIF codes, see [Handling DIF Codes](#).

## Requirements

Version	Supported in Microsoft Windows 2000 and later versions of Windows.
Header	Setupapi.h (include Setupapi.h)

## See also

[SetupDiUnremoveDevice](#)

[SP\\_DEVINFO\\_DATA](#)

[SP\\_DEVINSTALL\\_PARAMS](#)

[SP\\_UNREMOVEDevice\\_PARAMS](#)

# DIF\_UPDATEDRIVER\_UI

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is reserved for system use. Vendor-supplied installers must not handle this request.

# DIF\_VALIDATECLASSDRIVERS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

# DIF\_VALIDATEDRIVER

12/5/2018 • 2 minutes to read • [Edit Online](#)

This DIF code is obsolete and no longer supported in Microsoft Windows 2000 and later versions of Windows.

# DiskClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

DiskClassGuid is an obsolete identifier for the device interface class for hard disk [storage devices](#). Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_DISK](#) class identifier for new instances of this class.

## Remarks

The storage [samples](#) in the WDK include the [disk class driver](#) sample and the [Addfilter Storage Filter Tool](#). The disk class driver sample uses DiskClassGuid to register instances of the GUID\_DEVINTERFACE\_DISK device interface class. The sample Addfilter application uses DiskClassGuid to enumerate instances of the GUID\_DEVINTERFACE\_DISK device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_DISK instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_DISK](#)

# Disks Section of a TxtSetup.ohm File

12/5/2018 • 2 minutes to read • [Edit Online](#)

The **Disks** section identifies the disks in the device installation kit. This section has the following format:

```
[Disks]
diskN = "description",tagfile,directory
...
```

*diskN*

Specifies a key that can be used in subsequent sections to identify the disk.

*description*

Specifies a string containing the name of the disk. Windows uses the description to prompt the user to insert the disk.

*tagfile*

Specifies the name of a verification file on the disk. The filename must be specified as a full path from the root and must not specify a drive. Windows checks for this file to ensure that the user inserted the correct disk.

*directory*

Specifies the directory on the disk where the installation files are located. The directory must be specified as a full path from the root and must not specify a drive.

The following example shows a **Disks** section for an installation kit with two disks:

```
[Disks]
disk1 = "OEM SCSI driver disk 1",\disk1.tag,\
disk2 = "OEM SCSI driver disk 2",\disk2.tag,\
; ...
```

# dpinst XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **dpinst** XML element is the root XML element in a DPInst descriptor file that contains the child elements that customize driver installation.

## Element Tag

```
<dpinst>
```

## XML Attributes

None

## Element Information

Parent elements	None permitted
Child elements	<a href="#">enableNotListedLanguages</a> (zero or one) <a href="#">deleteBinaries</a> (zero or one) <a href="#">forceIfDriverIsNotBetter</a> (zero or one) <a href="#">group</a> (zero or more) <a href="#">headerPath</a> (zero or one) <a href="#">icon</a> (zero or one) <a href="#">installAllOrNone</a> (zero or one) <a href="#">language</a> (zero or more) <a href="#">legacyMode</a> (zero or one) <a href="#">promptIfDriverIsNotBetter</a> (zero or one) <a href="#">quietInstall</a> (zero or one) <a href="#">scanHardware</a> (zero or one) <a href="#">search</a> (zero or more) <a href="#">suppressAddRemovePrograms</a> (zero or one) <a href="#">watermarkPath</a> (zero or one)
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates an XML declaration element, followed by a **dpinst** element, which contains zero or more child elements.

```
<?xml version="1.0" ?>
<dpinst>
  ...
</dpinst>
```

**Note** Because duplicate child elements are not permitted, each **search** child element and **language** element of a **dpinst** element must be unique.

## See also

[deleteBinaries](#)

[enableNotListedLanguages](#)

[forceIfDriverIsNotBetter](#)

[group](#)

[headerPath](#)

[icon](#)

[installAllOrNone](#)

[language](#)

[legacyMode](#)

[promptIfDriverIsNotBetter](#)

[quietInstall](#)

[scanHardware](#)

[search](#)

[suppressAddRemovePrograms](#)

[watermarkPath](#)

# dpinstTitle XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **dpinstTitle** XML element customizes the text that appears on the title bar of all of the DPInst wizard pages.

## Element Tag

```
<dpinstTitle>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the title bar text on all of the wizard pages
Duplicate child elements	None permitted

## Remarks

The following code example of a **dpinstTitle** element customizes the title bar text. The text that specifies the custom DPInst title is shown in bold font style.

```
<dpinst>
  ...
  <language code="0x0409">
    ...
    <dpinstTitle>Toaster Device Installer</dpinstTitle>
    ...
  </language>
  ...
</dpinst>
```

If a **dpinstTitle** element is not specified, DPInst displays the default title bar text that appears on the default welcome page.

## See also

[language](#)

# enableNotListedLanguages XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **enableNotListedLanguages** XML element is an empty element that sets the **enableNotListedLanguages** flag to ON, which configures DPInst to enable all of the supported languages that are not explicitly enabled by **language** XML elements in a *DPInst.xml* file.

## Element Tag

```
<enableNotListedLanguages>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, all of the DPInst-supported languages are enabled if a *DPInst.xml* file does not contain any **language** elements. However, if the *DPInst.xml* file contains **language** elements, only the languages that are explicitly specified by the **language** elements are enabled and all of the other languages are implicitly disabled. To enable all of the languages that are implicitly disabled, set the **enableNotListedLanguages** flag to ON by including an **enableNotListedLanguages** element as a child element of a **dpinst** XML element or use the **/el** command-line switch.

The following code example demonstrates an **enableNotListedLanguages** element.

```
<dpinst>
  ...
  <enableNotListedLanguages/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# eula XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **eula** XML element is an empty XML element that includes two attributes that specify a EULA text file that contains custom text for a DPInst EULA page.

## Element Tag

```
<eula>
```

## XML Attributes

Type	The type of vendor-supplied EULA. The value of this attribute must be set to the string "txt", which indicates a plain-text file.
Path	A string that identifies the name of the file that contains the text for a DPInst EULA page. The EULA text file must be encoded by using UTF-8 encoding. The EULA file must be located in the DPInst root directory, which is the directory that contains the DPInst executable file ( <i>DPInst.exe</i> ), or a subdirectory under the DPInst root directory. If the EULA file is in a subdirectory, specify a fully qualified file name that is relative to the DPInst root directory.

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **eula** element that specifies that *Data\Eula409.txt* contains custom EULA text. The *Eula409.txt* file is in the *Data* directory, which must be a subdirectory under the DPInst root directory. The text that specifies the custom EULA file is shown below using the `<eula>` tag.

```
<dpinst>
...
<language code="0x0409">
...
<eula type="txt" path="Data\Eula409.txt"/>
...
</language>
...
</dpinst>
```

## See also

[language](#)

# eulaHeaderTitle XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **eulaHeaderTitle** XML element customizes the text of the EULA header title that appears directly below the title bar on a DPInst EULA page.

## Element Tag

```
<eulaHeaderTitle>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the title on a DPInst EULA page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **eulaHeaderTitle** element that customizes the header title of a DPInst EULA page. The text that specifies the custom header title is shown in bold font style.

```
<dpinst>
  ...
  <language code="0x0409">
    ...
    <eulaHeaderTitle>End User License Agreement</eulaHeaderTitle>
    ...
  </language>
  ...
</dpinst>
```

If a **eulaHeaderTitle** element is not specified, DPInst displays the default EULA header title that is shown on the default DPInst EULA page.

## See also

[language](#)

# eulaNoButton XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **eulaNoButton** XML element customizes the text that is associated with the do-not-accept option button on a DPInst EULA page.

## Element Tag

```
<eulaNoButton>
```

## XML Attributes

None

## Element Information

Parent elements	<b>language</b>
Child elements	None permitted
Data contents	String that customizes the text that is associated with the do-not-accept option button on a DPInst EULA page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **eulaNoButton** element that customizes the text of the do-not-accept option button on a DPInst EULA page. The text that specifies the custom text of the do-not-accept option button is shown in bold font style.

```
<dpinst>
  ...
<language code="0x0409">
  ...
    <eulaNoButton>I do n&ot accept this EULA</eulaNoButton>
  ...
</language>
  ...
</dpinst>
```

If a **eulaNoButton** element is not specified, DPInst displays the default button text that is shown on the default DPInst EULA page.

## See also

[eula](#)

[eulaYesButton](#)

language

# eulaYesButton XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **eulaYesButton** XML element customizes the text that is associated with the accept option button on a DPInst EULA page.

## Element Tag

```
<eulaYesButton>
```

## XML Attributes

None

## Element Information

Parent elements	<b>language</b>
Child elements	None permitted
Data contents	String that customizes the text that is associated with the accept option button on a DPInst EULA page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **eulaYesButton** element that customizes the accept option button text on a DPInst EULA page. The text that specifies the custom text of the accept option button is shown in bold font style.

```
<dpinst>
  ...
<language code="0x0409">
  ...
  <eulaYesButton>I &accept this EULA</eulaYesButton>
  ...
</language>
...
</dpinst>
```

If a **eulaYesButton** element is not specified, DPInst displays the default option button text that is shown on the default DPInst EULA page.

## See also

[eula](#)

[eulaNoButton](#)

language

# Files.HwComponent.ID Section of a TxtSetup.oem File

12/5/2018 • 2 minutes to read • [Edit Online](#)

A **Files.HwComponent.ID** section lists the files to be copied if the user selects a particular component option. One of these sections must be present for each option listed in each **HwComponent** section.

```
[Files.HwComponent.ID]
filetype = diskN,filename[,DriverKey]
...
```

## Files.HwComponent.ID

*HwComponent* corresponds to the name of a **HwComponent** section in the file. *ID* corresponds to an *ID* entry in that **HwComponent** section.

### filetype

Identifies the type of the file to be copied. One of these entries is present for each file to be copied for this **HwComponent.ID**.

The *filetype* is one of the following system-defined values:

#### driver

Valid for all components. Windows copies the file to `%systemroot%\system32\drivers`.

#### dll

Valid for all components. Useful for the GDI portion of a display driver. Windows copies the file to `%systemroot%\system32`.

#### hal

Valid only for the **computer** component. Windows copies the file to `%systemroot%\system32\hal.dll` (for x86) or to `\os\winnt\hal.dll` on the system partition (for non-x86).

#### inf

Valid for all components. Specifies the regular INF file for the device. This file is used during GUI-mode setup and for other device maintenance operations. The file is copied to `%systemroot%\system32`.

#### catalog

Valid for drivers. Specifies a catalog file for the device. Not required for any component. For example, `catalog = d1, mydriver.cat`. See the WHQL guidelines for more information about catalog files.

#### detect

Valid for the **computer** component (x86 only). If specified, replaces the standard x86 hardware recognizer. Windows copies the file to `c:\ntdetect.com`.

#### diskN

Identifies the disk from which to copy the file. This value must match an entry in the **Disks** section.

#### filename

Specifies the name of the file, not including the directory path or drive. To form the full file name, Windows appends the *filename* to the directory specified for the disk in the **Disks** section. File names must not exceed eight characters, and extensions must not exceed three characters.

#### DriverKey

Specifies the name of the key to be created in the registry services tree for this file, if the file is of type **driver**. This

value is used to form **Config.DriverKey** section names. This value is required for components of type **scsi**.

The following example shows a **Files.HwComponent.ID** section in a *TxtSetup.oem* file:

```
; ...
[Files.SCSI.oemscsi]
driver = d1,oemfs2.sys,OEMSCSI
inf = d1,oemsetup.inf
dll = d1, oemdrv.dll
catalog = d1, oemdrv.cat
; ...
```

# finishText XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **finishText** XML element customizes the main text that DPInst displays on a DPInst finish page.

## Element Tag

```
<finishText>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the main text on a finish page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **finishText** element that customizes the main text on a finish page that DPInst displays for a successful installation. The text that specifies the custom finish text is shown in bold font style.

```
dpinst>
...
<language code="0x0409">
...
  <finishText>Enjoy using the Toaster.</finishText>
...
</language>
...
</dpinst>
```

If a **finishText** element is not specified, DPInst displays default finish text that indicates whether the installation was a success or a failure.

## See also

[finishTitle](#)

[language](#)

# finishTitle XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **finishTitle** XML element customizes the text of the finish title that appears at the top of a DPInst finish page.

## Element Tag

```
<finishTitle>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the title text at the top of a finish page.
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **finishTitle** element that customizes the title text at the top of a finish page. The text that specifies the custom title text is shown in bold font style.

```
dpinst>
...
<language code="0x0409">
...
  <finishTitle>Congratulations! You are finished installing your Toaster device.</finishTitle>
...
</language>
...
</dpinst>
```

If a **finishTitle** element is not specified, DPInst displays the default title text that is shown on the default finish page.

## See also

[finishText](#)

[language](#)

# FloppyClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

FloppyClassGuid is an obsolete identifier for the device interface class for floppy disk **storage devices**. Starting Microsoft Windows 2000, use the **GUID\_DEVINTERFACE\_FLOPPY** class identifier for new instances of this class.

## Remarks

The storage **samples** in the WDK include a **floppy driver** sample that uses FloppyClassGuid to register instances of this device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_FLOPPY instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_FLOPPY](#)

# forceIfDriverIsNotBetter XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **forceIfDriverIsNotBetter** XML element is an empty element that sets the **forceIfDriverIsNotBetter** flag to ON, which configures DPInst to install a driver on a device even if the driver that is currently installed on the device is a better match than the new driver.

## Element Tag

```
<forceIfDriverIsNotBetter>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **forceIfDriverIsNotBetter** flag is set to OFF. You can set the **forceIfDriverIsNotBetter** flag to ON by including a **forceIfDriverIsNotBetter** element as a child element of a [dpinst XML element](#) in a DPinst descriptor file or by using the /f command-line switch.

The following code example demonstrates a **forceIfDriverIsNotBetter** element.

```
<dpinst>
  ...
  <forceIfDriverIsNotBetter/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# group XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **group** XML element specifies an ordered collection of [driver packages](#) that DPInst handles as a driver package group.

## Element Tag

```
<group>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a>
Child elements	<a href="#">package</a> (zero or more) <a href="#">installAllOrNone</a> (zero or one) <a href="#">suppressAddRemovePrograms</a> (zero or one)
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **group** element that includes two [package XML elements](#) and an [installAllOrNone XML element](#). The example **group** element configures DPInst to handle the "Abc" [driver package](#) and the "Def" driver package as a group. The [installAllOrNone](#) XML element configures DPInst to install the driver packages in the driver package group only if both drivers can be installed.

```
<dpinst>
  ...
  <group>
    <package path="DirAbc\Abc.inf" />
    <package path="DirDef\Def.inf" />
    <installAllOrNone/>
  </group>
  ...
</dpinst>
```

## See also

[dpinst](#)

[installAllOrNone](#)

**package**

**suppressAddRemovePrograms**

# GUID\_61883\_CLASS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_61883\_CLASS [device interface class](#) is defined for devices in the 61883 [device setup class](#).

ATTRIBUTE	SETTING
Identifier	GUID_61883_CLASS
Class GUID	{7EBEFBC0-3200-11d2-B4C2-00A0C9697D07}

## Remarks

Drivers for devices in the 61883 device setup class register instances of this device interface class to notify the operating system and applications of the presence of 61883 devices. The 61883 device interface class includes IEEE 1394 devices that support the IEC-61883 protocol. For information about 61883 devices and drivers, see [IEC-61883 Client Drivers](#).

For information about the device setup class for 1394 bus devices, see [BUS1394\\_CLASS\\_GUID](#).

## Requirements

Version	Available in Windows XP and later versions of Windows.
Header	61883.h (include 61883.h)

## See also

[BUS1394\\_CLASS\\_GUID](#)

# GUID\_AVC\_CLASS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_AVC\_CLASS [device interface class](#) is defined for audio video control (AV/C) devices that are supported by the [AVStream](#) architecture.

ATTRIBUTE	SETTING
Identifier	GUID_AVC_CLASS
Class GUID	{095780C3-48A1-4570-BD95-46707F78C2DC}

## Remarks

The system-supplied [AV/C client driver Avc.sys](#) registers an instance of GUID\_AVC\_CLASS to represent an external AV/C unit on a 1394 bus.

For information about the device interface class for virtual AV/C devices, see [GUID\\_VIRTUAL\\_AVC\\_CLASS](#).

## Requirements

Version	Available in Windows Vista, Windows Server 2003, Windows XP and later versions of Windows.
Header	Avc.h (include Avc.h)

## See also

[GUID\\_VIRTUAL\\_AVC\\_CLASS](#)

# GUID\_BTHPORT\_DEVICE\_INTERFACE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_BTHPORT\_DEVICE\_INTERFACE [device interface class](#) is defined for [Bluetooth radios](#).

ATTRIBUTE	SETTING
Identifier	GUID_BTHPORT_DEVICE_INTERFACE
Class GUID	{0850302A-B344-4fda-9BE9-90576B8D46F0}

## Remarks

Drivers for [Bluetooth radios](#) register instances of this device interface class to notify the operating system and applications of the presence of Bluetooth radios.

## Requirements

Version	Available in Windows Vista, Windows XP SP2, and later versions of Windows.
Header	Bthdef.h (include Bthdef.h)

# GUID\_CLASS\_COMPORT

12/5/2018 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_COMPORT is an obsolete identifier for the device interface class for devices that support a 16550 UART-compatible hardware interface. Use the [GUID\\_DEVINTERFACE\\_COMPORT](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Use <a href="#">GUID_DEVINTERFACE_COMPORT</a> instead.
Header	Ntddser.h (include Ntddser.h)

## See also

[GUID\\_DEVINTERFACE\\_COMPORT](#)

# GUID\_CLASS\_INPUT

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_INPUT is an obsolete identifier for the [device interface class](#) for [HID collections](#). Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_HID](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_HID instead.
Header	Hidclass.h (include Hidclass.h)

## See also

[GUID\\_DEVINTERFACE\\_HID](#)

# GUID\_CLASS\_KEYBOARD

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_KEYBOARD is an obsolete identifier for the [device interface class](#) for keyboard devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_KEYBOARD](#) class identifier for new instances of this class.

## Remarks

The HID samples that are provided in the WDK include the keyboard class driver. The keyboard class driver uses GUID\_CLASS\_KEYBOARD to register instances of this device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_KEYBOARD instead.
Header	Ntddkbd.h (include Ntddkbd.h)

## See also

[GUID\\_DEVINTERFACE\\_KEYBOARD](#)

# GUID\_CLASS\_MODEM

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_MODEM is an obsolete identifier for the [device interface class](#) for modem devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_MODEM](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_MODEM instead.
Header	Ntddmodm.h (include Ntddmodm.h)

## See also

[GUID\\_DEVINTERFACE\\_MODEM](#)

# GUID\_CLASS\_MOUSE

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_MOUSE is an obsolete identifier for the [device interface class](#) for mouse devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_MOUSE](#) class identifier for new instances of this class.

## Remarks

The HID samples that are provided in the WDK include the mouse class driver. The mouse class driver uses GUID\_CLASS\_MOUSE to register instances of this device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_MOUSE instead.
Header	Ntddmou.h (include Ntddmou.h)

## See also

[GUID\\_DEVINTERFACE\\_MOUSE](#)

# GUID\_CLASS\_USB\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_USB\_DEVICE is an obsolete identifier for the [device interface class](#) for [USB](#) devices that are attached to a USB hub. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_USB\\_DEVICE](#) class identifier for new instances of this class.

## Remarks

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The USBVIEW sample uses GUID\_CLASS\_USB\_DEVICE to register to be notified if instances of the GUID\_CLASS\_USB\_DEVICE interface class are present.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_USB_DEVICE instead.
Header	Usbiodef.h (include Usbiodef.h)

Previously, this identifier was dependent on `Usbioctl.h`. Note that you now need to include `Usbiodef.h` instead.

## See also

[GUID\\_DEVINTERFACE\\_USB\\_DEVICE](#)

# GUID\_CLASS\_USB\_HOST\_CONTROLLER

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_USB\_HOST\_CONTROLLER is an obsolete identifier for the [device interface class](#) for [USB](#) host controller devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_USB\\_HOST\\_CONTROLLER](#) class identifier for new instances of this class.

## Remarks

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The USBVIEW sample uses GUID\_CLASS\_USB\_HOST\_CONTROLLER to enumerate instances of the GUID\_CLASS\_USB\_HOST\_CONTROLLER device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use <a href="#">GUID_DEVINTERFACE_USB_HOST_CONTROLLER</a> instead.
Header	Usbiodef.h (include Usbiodef.h)

## See also

[GUID\\_DEVINTERFACE\\_USB\\_HOST\\_CONTROLLER](#)

# GUID\_CLASS\_USBHUB

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_CLASS\_USBHUB is an obsolete identifier for the [device interface class](#) for [USB](#) hub devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_USB\\_HUB](#) class identifier for new instances of this class.

## Remarks

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The USBVIEW sample uses GUID\_CLASS\_USBHUB to be notified if the instances of the GUID\_CLASS\_USBHUB device interface class are present.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_USB_HUB instead.
Header	Usbiodef.h (include Usbiodef.h)

## See also

[GUID\\_DEVINTERFACE\\_USB\\_HUB](#)

# GUID\_DEVICE\_APPLICATIONLAUNCH\_BUTTON

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_APPLICATIONLAUNCH\_BUTTON [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) application start buttons.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_APPLICATIONLAUNCH_BUTTON
Class GUID	{629758EE-986E-4D9E-8E47-DE27F8AB054D}

## Remarks

The system-supplied [ACPI driver](#) registers an instance of this device interface class to notify the operating system and applications of the presence of ACPI application start buttons.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_BATTERY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_BATTERY device interface class is defined for [battery devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_BATTERY
Class GUID	{72631E54-78A4-11D0-BCF7-00AA00B7B32A}

## Remarks

The system-supplied [battery class driver](#) registers an instance of this device interface class for a battery device on behalf of a battery miniclass driver.

For information about battery devices and drivers, see [Overview of System Battery Management](#).

## Requirements

Header	Batclass.h (include Batclass.h)
--------	---------------------------------

# GUID\_DEVICE\_LID

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_LID [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) lid devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_LID
Class GUID	{4AFA3D52-74A7-11d0-be5e-00A0C9062857}

## Remarks

The system-supplied [ACPI driver](#) registers instances of this device interface class to notify the operating system and applications of the presence of ACPI lid devices.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_MEMORY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_MEMORY [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) memory devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_MEMORY
Class GUID	{3FD0F03D-92E0-45FB-B75C-5ED8FFB01021}

## Remarks

The system-supplied [ACPI driver](#) registers instances of this device interface class to notify the operating system and applications of the presence of ACPI memory devices.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_MESSAGE\_INDICATOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_MESSAGE\_INDICATOR [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) message indicator devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_MESSAGE_INDICATOR
Class GUID	{CD48A365-FA94-4CE2-A232-A1B764E5D8B4}

## Remarks

The system-supplied [ACPI driver](#) registers an instance of this device interface class to notify the operating system and applications of the presence of ACPI message indicator devices.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_PROCESSOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_PROCESSOR [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) processor devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_PROCESSOR
Class GUID	{97FADB10-4E33-40AE-359C-8BEF029DBDD0}

## Remarks

The system-supplied [ACPI driver](#) registers an instance of this device interface class to notify the operating system and applications of the presence of processor devices.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_SYS\_BUTTON

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVICE\_SYS\_BUTTON [device interface class](#) is defined for Advanced Configuration and Power Interface (ACPI) system power button devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_SYS_BUTTON
Class GUID	{4AFA3D53-74A7-11d0-be5e-00A0C9062857}

## Remarks

The system-supplied [ACPI driver](#) registers an instance of this device interface class to notify the operating system and applications of the presence of system power button devices. I8042prt, the system-supplied driver for PS/2-style keyboard and mouse devices, also registers an instance of this class for a keyboard that supports a system power button.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

For information about PS/2-style keyboard and mouse devices, see [Non-HID Class Keyboard and Mouse Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVICE\_THERMAL\_ZONE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVICE\\_THERMAL\\_ZONE](#) device interface class is defined for Advanced Configuration and Power Interface (ACPI) thermal zone devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVICE_THERMAL_ZONE
Class GUID	{4AFA3D51-74A7-11d0-be5e-00A0C9062857}

## Remarks

The system-supplied [ACPI driver](#) registers an instance of this device interface class to notify the operating system and applications of the presence of thermal zone devices.

For information about supplying WDM [function drivers](#) for ACPI devices, see [Supporting ACPI Devices](#).

## Requirements

Header	Poclass.h (include Poclass.h)
--------	-------------------------------

# GUID\_DEVINTERFACE\_BRIGHTNESS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_BRIGHTNESS device interface class](#) is defined for display adapter drivers that operate in the context of the [Windows Vista Display Driver Model](#) and support brightness control of monitor child devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_BRIGHTNESS
Class GUID	{FDE5BBA4-B3F9-46FB-BDAA-0728CE3100B4}

## Remarks

Drivers register instances of this device interface class to notify the operating system and applications of the presence of brightness control interfaces for monitor child devices.

If the display miniport driver supports a direct-call brightness control interface for this [device setup class](#), a kernel-mode component can retrieve the direct-call interface by calling the miniport driver's [DxgkDdiQueryInterface](#) function and supplying [GUID\\_DEVINTERFACE\\_BRIGHTNESS](#) to specify the interface type.

For information about brightness devices, see [Supporting Brightness Controls on Integrated Display Panels](#) and [Brightness Control Interface](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Dispmprt.h (include Dispmprt.h)

# GUID\_DEVINTERFACE\_CDCHANGER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_CDCHANGER [device interface class](#) is defined for CD-ROM changer devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_CDCHANGER
Class GUID	{53F56312-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied CD-ROM [changer driver](#) registers instances of GUID\_DEVINTERFACE\_CDCHANGER to notify the operating system and applications of the presence of CD-ROM changer devices.

For information about the device interface class for CD-ROM devices, see [GUID\\_DEVINTERFACE\\_CDROM](#).

For information about storage devices, see [Storage Drivers](#).

**CdChangerClassGuid** is an obsolete identifier for the GUID\_DEVINTERFACE\_CDCHANGER device interface class; for new instances of this class, use GUID\_DEVINTERFACE\_CDCHANGER instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[CdChangerClassGuid](#)

[GUID\\_DEVINTERFACE\\_CDROM](#)

# GUID\_DEVINTERFACE\_CDROM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_CDROM device interface class is defined for CD-ROM storage devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_CDROM
Class GUID	{53F56308-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied class driver for CD-ROM storage devices registers instances of this device interface class to notify the operating system and applications of the presence of a CD-ROM device.

The storage samples in the WDK include the [CDROM class driver](#) sample and the [Addfilter Storage Filter Tool](#). The CD-ROM class driver sample uses the obsolete identifier [CdRomClassGuid](#) to register instances of the GUID\_DEVINTERFACE\_CDROM device interface class. The sample Addfilter application uses CdRomClassGuid to enumerate instances of the GUID\_DEVINTERFACE\_CDROM device interface class.

For information about the device interface class for CD-ROM changer devices, see [GUID\\_DEVINTERFACE\\_CDCHANGER](#).

For information about storage devices, see [Storage Drivers](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[CdRomClassGuid](#)

[GUID\\_DEVINTERFACE\\_CDCHANGER](#)

# GUID\_DEVINTERFACE\_COMPORT

12/1/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_COMPORT device interface class is defined for COM ports.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_COMPORT
Class GUID	{86E0D1E0-8089-11D0-9CE4-08003E301F73}

## Remarks

Drivers for serial ports register instances of this device interface class to notify the operating system and applications of the presence of COM ports.

The system-supplied function driver for serial ports registers an instance of this device interface class for a [serial port](#).

The following samples (on Github) register an instance of this class for a serial port:

- [The Serial sample](#)
- [The Virtual serial driver sample](#) (UMDF 1.0)
- [The Virtual serial2 driver sample](#) (KMDF)

**GUID\_CLASS\_COMPORT** is an obsolete identifier for this device interface class; for new instances of this class, use GUID\_DEVINTERFACE\_COMPORT instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddser.h (include Ntddser.h)

## See also

[GUID\\_CLASS\\_COMPORT](#)

# GUID\_DEVINTERFACE\_DISK

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_DISK [device interface class](#) is defined for hard disk storage devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_DISK
Class GUID	{53F56307-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied storage class drivers register an instance of GUID\_DEVINTERFACE\_DISK for a hard disk storage device.

The storage [samples](#) in the WDK include the [disk class driver](#) sample and the [Addfilter Storage Filter Tool](#). The disk class driver sample uses the obsolete identifier [DiskClassGuid](#) to register instances of the GUID\_DEVINTERFACE\_DISK device interface class. The sample Addfilter application uses DiskClassGuid to enumerate instances of GUID\_DEVINTERFACE\_DISK device interface class.

For information about storage drivers, see [Storage Drivers](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[DiskClassGuid](#)

# GUID\_DEVINTERFACE\_DISPLAY\_ADAPTER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `GUID_DEVINTERFACE_DISPLAY_ADAPTER` device interface class is defined for display views that are supported by display adapters.

ATTRIBUTE	SETTING
Identifier	<code>GUID_DEVINTERFACE_DISPLAY_ADAPTER</code>
Class GUID	<code>{5B45201D-F2F2-4F3B-85BB-30FF1F953599}</code>

## Remarks

The system-supplied display drivers register an instance of this device interface class to notify the operating system and applications of the presence of a display view.

For information about display devices, see [Windows Vista Display Driver Model](#) and [Windows 2000 Display Driver Model](#).

For information about the device interface class for display adapters, see [GUID\\_DISPLAY\\_DEVICE\\_ARRIVAL](#).

## Requirements

Header	<code>Ntddvdeo.h</code> (include <code>Ntddvdeo.h</code> )
--------	--

## See also

[GUID\\_DISPLAY\\_DEVICE\\_ARRIVAL](#)

# GUID\_DEVINTERFACE\_FLOPPY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_FLOPPY [device interface class](#) is defined for floppy disk [storage devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_FLOPPY
Class GUID	{53F56311-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied storage class driver for floppy disk storage devices registers an instance of GUID\_DEVINTERFACE\_FLOPPY for a floppy disk storage device.

The storage [samples](#) in the WDK include a [floppy driver](#) sample that uses the obsolete identifier [FloppyClassGuid](#) to register an instance of the GUID\_DEVINTERFACE\_FLOPPY device interface class.

For information about storage drivers, see [Storage Drivers](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[FloppyClassGuid](#)

# GUID\_DEVINTERFACE\_HID

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_HID device interface class is defined for [HID collections](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_HID
Class GUID	{4D1E55B2-F16F-11CF-88CB-001111000030}

## Remarks

Drivers for HID collections register instances of this device interface class to notify the operating system and applications of the presence of HID collections.

The system-supplied [HID class driver](#) registers an instance of this device interface class for a HID collection. For example, the HID class driver registers an interface for a USB keyboard or mouse device. Access a HID collection by using the I/O interface supported by the HID class driver.

For information about HID devices and drivers, see [HIDClass Devices](#).

For information about the device interface class for keyboard devices, see [GUID\\_DEVINTERFACE\\_KEYBOARD](#).

For information about the device interface class for mouse devices, see [GUID\\_DEVINTERFACE\\_MOUSE](#).

The [GUID\\_CLASS\\_INPUT](#) is an obsolete identifier for this device interface class; use GUID\_DEVINTERFACE\_HID instead.

## Requirements

Header	Hidclass.h (include Hidclass.h)
--------	---------------------------------

## See also

[GUID\\_CLASS\\_INPUT](#)

[GUID\\_DEVINTERFACE\\_KEYBOARD](#)

[GUID\\_DEVINTERFACE\\_MOUSE](#)

# GUID\_DEVINTERFACE\_I2C

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_I2C device interface class](#) is defined for display adapter drivers that operate in the context of the [Windows Vista Display Driver Model](#) and perform I2C transactions with monitor child devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_I2C
Class GUID	{2564AA4F-DDDB-4495-B497-6AD4A84163D7}

## Remarks

Drivers register instances of this device interface class to notify the operating system and applications of the presence of I2C interfaces that perform transactions with monitor child devices.

If a display miniport driver supports a direct-call I2C interface for this [device setup class](#), a kernel-mode component can retrieve the direct-call interface by calling the miniport driver's [DxgkDdiQueryInterface](#) function and supplying GUID\_DEVINTERFACE\_I2C to specify the interface type.

For information about the I2C bus, see [I2C Bus and Child Devices of the Display Adapter](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Dispmprt.h (include Dispmprt.h)

# GUID\_DEVINTERFACE\_IMAGE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **GUID\_DEVINTERFACE\_IMAGE** device interface class is defined for [WIA devices and Still Image \(STI\) devices](#), including digital cameras and scanners.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_IMAGE
Class GUID	{6BDD1FC6-810F-11D0-BEC7-08002BE2092F}

## Remarks

The system-supplied kernel-mode drivers for WIA devices register an instance of this device interface class to notify the operating system and applications of the presence of WIA devices.

For information about WIA drivers and STI drivers, see [Windows Image Acquisition Drivers](#).

## Requirements

Version	Available in Windows XP and later versions of Windows.
Header	Wiaintfc.h (include Wiaintfc.h)

# GUID\_DEVINTERFACE\_KEYBOARD

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_KEYBOARD device interface class](#) is defined for keyboard devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_KEYBOARD
Class GUID	{884b96c3-56ef-11d1-bc8c-00a0c91405dd}

## Remarks

Drivers for keyboard devices register instances of this device interface class to notify the system and applications of the presence of keyboard devices.

The system-supplied [keyboard class driver](#) registers an instance of this device interface class for a keyboard device. Access an instance of this device interface class by using the I/O interface supported by the keyboard class driver.

For general information about supporting keyboard devices, see [HID Architecture](#) and [Features of the Kbdclass and Mouclass Drivers](#).

The WDK includes sample code for the system-supplied keyboard class driver. The keyboard class driver uses the obsolete identifier [GUID\\_CLASS\\_KEYBOARD](#) to register an instance of this [device setup class](#).

For information about the device interface class for mouse devices, see [GUID\\_DEVINTERFACE\\_MOUSE](#).

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddkbd.h (include Ntddkbd.h)

## See also

[GUID\\_CLASS\\_KEYBOARD](#)

[GUID\\_DEVINTERFACE\\_MOUSE](#)

# GUID\_DEVINTERFACE\_MEDIUMCHANGER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_MEDIUMCHANGER device interface class is defined for medium changer devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_MEDIUMCHANGER
Class GUID	{53F56310-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied medium [changer drivers](#) register an instance of GUID\_DEVINTERFACE\_MEDIUMCHANGER to notify the operating system and applications of the presence of medium changer devices.

For more information about storage drivers, see [Storage Drivers](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[MediumChangerClassGuid](#)

# GUID\_DEVINTERFACE\_MODEM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_MODEM device interface class is defined for [modem devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_MODEM
Class GUID	{2C7089AA-2E0E-11D1-B114-00C04FC2AAE4}

## Remarks

Drivers for modem devices register instances of this device interface class to notify the operating system and applications of the presence of modem devices.

GUID\_DEVINTERFACE\_MODEM in *Ntddmodm.h* will be defined correctly only if the correct versions of the INITGUID and DEFINE\_GUID macros are defined before the inclusion of *Ntddmodm.h*. The DEFINE\_GUID macro is defined in the *Guiddef.h*. To make sure that INITGUID, DEFINE\_GUID, and GUID\_DEVINTERFACE\_MODEM are correctly defined, include the following code in a header file:

```
#ifndef INITGUID
#define INITGUID
#include <guiddef.h>
#undef INITGUID
#else
#include <guiddef.h>
#endif

#include <ntddmodm.h>
...
```

For information about modem devices, see [Modem Devices Design Guide](#).

For an example of using this device interface class, see the [FakeModem - Unimodem controller-less modem sample driver](#) sample that is provided in the WDK.

**GUID\_CLASS\_MODEM** is an obsolete identifier for this device interface class; for new instances of this class, use GUID\_DEVINTERFACE\_MODEM instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddmodm.h (include Ntddmodm.h)

## See also

**GUID\_CLASS\_MODEM**

# GUID\_DEVINTERFACE\_MONITOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_MONITOR device interface class is defined for monitor devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_MONITOR
Class GUID	{E6F07B5F-EE97-4a90-B076-33F57BF4EAA7}

## Remarks

Windows registers a device interface for each monitor that is configured in the operating system.

For information about display adapter and monitors, see [Display Devices Design Guide](#) and [Monitor Drivers](#).

## Requirements

Header	Ntddvdeo.h (include Ntddvdeo.h)
--------	---------------------------------

# GUID\_DEVINTERFACE\_MOUSE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_MOUSE device interface class is defined for mouse devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_MOUSE
Class GUID	{378DE44C-56EF-11D1-BC8C-00A0C91405DD}

## Remarks

Drivers for mouse devices register instances of this device interface class to notify the operating system and applications of the presence of mouse devices.

The system-supplied [mouse class driver](#) registers an instance of this device interface class for a mouse device. Access an instance of this device interface class by using the I/O interface supported by the mouse class driver.

For general information about supporting mouse devices, see [HID Architecture](#) and [Features of the Kbdclass and Mouclass Drivers](#).

The WDK includes sample code for the system-supplied mouse class driver. The mouse class driver uses the obsolete identifier [GUID\\_CLASS\\_MOUSE](#) to register an instance of this [device setup class](#).

For information about the device interface class for keyboard devices, see [GUID\\_DEVINTERFACE\\_KEYBOARD](#).

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddmou.h (include Ntddmou.h)

## See also

[GUID\\_CLASS\\_MOUSE](#)

[GUID\\_DEVINTERFACE\\_KEYBOARD](#)

# GUID\_DEVINTERFACE\_NET

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_NET device interface class is defined for network devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_NET
Class GUID	{CAC88484-7515-4C03-82E6-71A87ABAC361}

## Remarks

Drivers of network devices register instances of this device interface class to notify other network components and applications of the presence of network devices.

NDIS registers instances of this interface class for NDIS miniport drivers.

For information about network devices and drivers, see [Network Design Guide](#).

## Requirements

Version	Available in Windows Vista, Windows Server 2003 Scalable Networking Pack (SNP), and later versions of Windows.
Header	Ndisguid.h (include Ndisguid.h)

# GUID\_DEVINTERFACE\_OPM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_OPM device interface class](#) is defined for display adapter drivers that operate in the context of the [Windows Vista Display Driver Model](#) and support output protection management (OPM) for monitor child devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_OPM
Class GUID	{BF4672DE-6B4E-4BE4-A325-68A91EA49C09}

## Remarks

Drivers register instances of this device interface class to notify the operating system and applications of the presence of OPM device interfaces.

If a display miniport driver supports a direct-call OPM interface for this [device setup class](#), a kernel-mode component can retrieve the direct-call interface by calling the miniport driver's [DxgkDdiQueryInterface](#) function and supplying GUID\_DEVINTERFACE\_OPM to specify the interface type.

For information about OPM, see [Supporting Output Protection Manager](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Dispmprt.h (include Dispmprt.h)

# GUID\_DEVINTERFACE\_PARALLEL

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **GUID\_DEVINTERFACE\_PARALLEL** device interface class is defined for [parallel ports](#) that support an IEEE 1284-compatible hardware interface.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_PARALLEL
Class GUID	{97F76EF0-F883-11D0-AF1F-0000F800845C}

## Remarks

Drivers for parallel ports register instances of **GUID\_DEVINTERFACE\_PARALLEL** to notify the operating system and applications of the presence of parallel ports.

The system-supplied function driver for parallel ports registers an instance of this device class for a parallel port.

For information about parallel devices and drivers, see [Introduction to Parallel Ports and Devices](#).

For information about the device interface class for devices that are attached to a parallel port, see [GUID\\_DEVINTERFACE\\_PARCLASS](#).

[GUID\\_PARALLEL\\_DEVICE](#) is an obsolete identifier for this device interface class; for new instances of this class, use **GUID\_DEVINTERFACE\_PARALLEL** instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddpar.h (include Ntddpar.h)

## See also

[GUID\\_DEVINTERFACE\\_PARCLASS](#)

[GUID\\_PARALLEL\\_DEVICE](#)

# GUID\_DEVINTERFACE\_PARCLASS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_PARCLASS](#) device interface class is defined for devices that are attached to a [parallel port](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_PARCLASS
Class GUID	{811FC6A5-F728-11D0-A537-0000F8753ED1}

## Remarks

Drivers for parallel devices that are attached to parallel ports register instances of [GUID\\_DEVINTERFACE\\_PARCLASS](#) to notify the operating system and applications of the presence of parallel devices.

The system-supplied bus driver for parallel ports creates an instance of this device interface class for each hardware device that is attached to a parallel port.

For information about parallel devices and drivers, see [Parallel Devices Design Guide](#).

For information about the device interface class for parallel ports, see [GUID\\_DEVINTERFACE\\_PARALLEL](#).

[GUID\\_PARCLASS\\_DEVICE](#) is an obsolete identifier for this device interface class; for new instances of this class, use [GUID\\_DEVINTERFACE\\_PARCLASS](#) instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddpar.h (include Ntddpar.h)

## See also

[GUID\\_DEVINTERFACE\\_PARALLEL](#)

[GUID\\_PARCLASS\\_DEVICE](#)

# GUID\_DEVINTERFACE\_PARTITION

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_PARTITION device interface class](#) is defined for partition devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_PARTITION
Class GUID	{53F5630A-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied [storage drivers](#) register an instance of [GUID\\_DEVINTERFACE\\_PARTITION](#) for a partition that is a child device of a [storage device](#).

[PartitionClassGuid](#) is an obsolete identifier for the [GUID\\_DEVINTERFACE\\_PARTITION](#) device interface class. For new instances of this class, use [GUID\\_DEVINTERFACE\\_PARTITION](#) instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[PartitionClassGuid](#)

# GUID\_DEVINTERFACE\_SERENUM\_BUS\_ENUMERATOR

12/1/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_SERENUM\\_BUS\\_ENUMERATOR](#) device interface class is defined for Plug and Play (PnP) serial ports.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_SERENUM_BUS_ENUMERATOR
Class GUID	{4D36E978-E325-11CE-BFC1-08002BE10318}

## Remarks

The system-supplied enumerator for serial port devices registers instances of [GUID\\_DEVINTERFACE\\_SERENUM\\_BUS\\_ENUMERATOR](#) to notify the operating system and applications of the presence of serial port devices.

The WDK includes the serial enumerator sample [serenum](#). The serenum sample uses the obsolete identifier [GUID\\_SERENUM\\_BUS\\_ENUMERATOR](#) to register instances of this device interface class. The serenum sample is located in the *src\kernel* directory of the WDK.

For information about serial devices and drivers, see [Serial Devices and Drivers](#).

For information about the device interface class for serial port devices, see [GUID\\_DEVINTERFACE\\_COMPORT](#).

## Requirements

Header	Ntddser.h (include Ntddser.h)
--------	-------------------------------

## See also

[GUID\\_DEVINTERFACE\\_COMPORT](#)

[GUID\\_SERENUM\\_BUS\\_ENUMERATOR](#)

# GUID\_DEVINTERFACE\_SIDESHOW

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_SIDESHOW device interface class is defined for [Windows SideShow devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_SIDESHOW
Class GUID	{152E5811-FEB9-4B00-90F4-D32947AE1681}

## Remarks

Drivers for Windows Sideshow-compatible devices register instances of GUID\_DEVINTERFACE\_SIDESHOW to notify the system and applications of the presence of Windows SideShow devices.

## Requirements

Version	Available in Windows Vista and later versions of the Microsoft Windows operating systems.
Header	WindowsSideShowDriverEvents.h (include WindowsSideShowDriverEvents.h)

# GUID\_DEVINTERFACE\_STORAGEPORT

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_STORAGEPORT [device interface class](#) is defined for storage port devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_STORAGEPORT
Class GUID	{2ACCFE60-C130-11D2-B082-00A0C91EFB8B}

## Remarks

The system-supplied drivers for storage port devices register instances of GUID\_DEVINTERFACE\_STORAGEPORT to notify the operating system and applications of the presence of storage device adapters.

For more information about storage drivers, see [Storage Drivers](#).

**StoragePortClassGuid** is an obsolete identifier for the GUID\_DEVINTERFACE\_STORAGEPORT device interface class. For new instances of this class, use GUID\_DEVINTERFACE\_STORAGEPORT instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[StoragePortClassGuid](#)

# GUID\_DEVINTERFACE\_TAPE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_TAPE [device interface class](#) is defined for tape [storage devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_TAPE
Class GUID	{53F5630B-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied [tape class driver](#) registers an instance of GUID\_DEVINTERFACE\_TAPE to notify the operating system and applications of the presence of tape storage devices.

For more information about storage drivers, see [Storage Drivers](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[TapeClassGuid](#)

# GUID\_DEVINTERFACE\_USB\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `GUID_DEVINTERFACE_USB_DEVICE` device interface class is defined for `USB` devices that are attached to a `USB` hub.

ATTRIBUTE	SETTING
Identifier	<code>GUID_DEVINTERFACE_USB_DEVICE</code>
Class GUID	<code>{A5DCBF10-6530-11D2-901F-00C04FB951ED}</code>

## Remarks

The system-supplied `USB` hub driver registers instances of `GUID_DEVINTERFACE_USB_DEVICE` to notify the system and applications of the presence of `USB` devices that are attached to a `USB` hub.

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The `USBVIEW` sample uses the obsolete identifier `GUID_CLASS_USB_DEVICE` to register to be notified of the arrival of instances of this device interface class.

You must include `initguid.h` before including any header that declares a `GUID` by using the `DEFINE_GUID` macro.

## Requirements

Header	<code>Usbiodef.h</code> (include <code>Usbiodef.h</code> , <code>initguid.h</code> )
--------	--

## See also

[GUID\\_CLASS\\_USB\\_DEVICE](#)

[GUID\\_DEVINTERFACE\\_USB\\_HOST\\_CONTROLLER](#)

[GUID\\_DEVINTERFACE\\_USB\\_HUB](#)

# GUID\_DEVINTERFACE\_USB\_HOST\_CONTROLLER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `GUID_DEVINTERFACE_USB_HOST_CONTROLLER` device interface class is defined for `USB` host controller devices.

ATTRIBUTE	SETTING
Identifier	<code>GUID_DEVINTERFACE_USB_HOST_CONTROLLER</code>
Class GUID	<code>{3ABF6F2D-71C4-462A-8A92-1E6861E6AF27}</code>

## Remarks

The system-supplied port driver for a USB host controller registers instances of `GUID_DEVINTERFACE_USB_HOST_CONTROLLER` to notify the operating system and applications of the presence of USB host controllers.

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The USBVIEW uses the obsolete identifier `GUID_CLASS_USB_HOST_CONTROLLER` to enumerate instances of this device interface class.

You must include `initguid.h` before including any header that declares a GUID by using the `DEFINE_GUID` macro.

## Requirements

Header	<code>Usbiodef.h</code> (include <code>Usbiodef.h</code> , <code>initguid.h</code> )
--------	--

## See also

[GUID\\_CLASS\\_USB\\_HOST\\_CONTROLLER](#)

[GUID\\_DEVINTERFACE\\_USB\\_DEVICE](#)

[GUID\\_DEVINTERFACE\\_USB\\_HUB](#)

# GUID\_DEVINTERFACE\_USB\_HUB

11/2/2020 • 2 minutes to read • [Edit Online](#)

The [GUID\\_DEVINTERFACE\\_USB\\_HUB device interface class](#) is defined for [USB hub devices](#).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_USB_HUB
Class GUID	{F18A0E88-C30C-11D0-8815-00A0C906BED8}

## Remarks

The system-supplied USB port driver registers instances of `GUID_DEVINTERFACE_USB_HUB` to notify the operating system and applications of the presence of the root hub of host controller devices. The system-supplied USB hub driver registers instances of this class for additional hub devices, if any, that are supported by the host controller.

The Microsoft Windows Driver Kit (WDK) includes the [USBVIEW sample application](#). The USBVIEW sample uses the obsolete identifier [GUID\\_CLASS\\_USBHUB](#) to be notified of instances of this device interface class.

You must include `initguid.h` before including any header that declares a GUID by using the `DEFINE_GUID` macro.

## Requirements

Header	<code>Usbiodef.h</code> (include <code>Usbiodef.h</code> , <code>initguid.h</code> )
--------	--

## See also

[GUID\\_CLASS\\_USBHUB](#)

[GUID\\_DEVINTERFACE\\_USB\\_DEVICE](#)

[GUID\\_DEVINTERFACE\\_USB\\_HOST\\_CONTROLLER](#)

# GUID\_DEVINTERFACE\_VIDEO\_OUTPUT\_ARRIVAL

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `GUID_DEVINTERFACE_VIDEO_OUTPUT_ARRIVAL` device interface class is defined for child devices of [display devices](#).

ATTRIBUTE	SETTING
Identifier	<code>GUID_DEVINTERFACE_VIDEO_OUTPUT_ARRIVAL</code>
Class GUID	{1AD9E4F0-F88D-4360-BAB9-4C2D55E564CD}

## Remarks

This Windows operating system does not use this device interface class.

For information about display devices, see [Windows Vista Display Driver Model](#) and [Windows 2000 Display Driver Model](#).

For information about the device interface class for display adapters, see [GUID\\_DISPLAY\\_DEVICE\\_ARRIVAL](#).

## Requirements

Header	Ntddvdeo.h (include Ntddvdeo.h)
--------	---------------------------------

## See also

[GUID\\_DISPLAY\\_DEVICE\\_ARRIVAL](#)

# GUID\_DEVINTERFACE\_VOLUME

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_VOLUME device interface class is defined for volume devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_VOLUME
Class GUID	{53F5630D-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied [storage class drivers](#) register instances of GUID\_DEVINTERFACE\_VOLUME to notify the operating system and applications of the presence of volume devices. The mount manager uses the Plug and Play (PnP) device interface notification mechanism to signal the arrival or removal of a volume interface.

The storage [samples](#) in the WDK include an [Addfilter Storage Filter Tool](#) that uses the obsolete identifier [VolumeClassGuid](#) to enumerate instances of the GUID\_DEVINTERFACE\_VOLUME device interface class.

For more information about storage drivers, see [Storage Drivers](#) and [Supporting Mount Manager Requests in a Storage Class Driver](#).

## Requirements

Header	Ntddstor.h (include Ntddstor.h)
--------	---------------------------------

## See also

[VolumeClassGuid](#)

# GUID\_DEVINTERFACE\_WPD

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_WPD [device interface class](#) is defined for [Windows Portable Devices](#) (WPD).

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_WPD
Class GUID	{6AC27878-A6FA-4155-BA85-F98F491D4F33}

## Remarks

Drivers for devices that support the WPD device-driver interface (DDI) register instances of GUID\_DEVINTERFACE\_WPD to notify the operating system and applications that WPD devices are present.

The WPD class extension component registers an instance of this device interface for a WPD driver that uses the WPD class extension.

Clients that use WPD devices should register to be notified of the arrival of instances of this device interface class.

Clients can enumerate WPD devices that register this interface by calling [IPortableDeviceManager::GetDevices](#) (documented in Windows SDK).

For information about the device interface class for private WPD devices, see [GUID\\_DEVINTERFACE\\_WPD\\_PRIVATE](#).

## Requirements

Version	Available in Windows Vista, Windows XP, and later versions of Windows.
Header	Portabledevice.h (include Portabledevice.h)

## See also

[GUID\\_DEVINTERFACE\\_WPD\\_PRIVATE](#)

# GUID\_DEVINTERFACE\_WPD\_PRIVATE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The `GUID_DEVINTERFACE_WPD_PRIVATE` device interface class is defined for specialized Windows Portable Devices (WPD).

ATTRIBUTE	SETTING
Identifier	<code>GUID_DEVINTERFACE_WPD_PRIVATE</code>
Class GUID	<code>{BA0C718F-4DED-49B7-BDD3-FABE28661211}</code>

## Remarks

`GUID_DEVINTERFACE_WPD_PRIVATE` should be used only for private devices that are used by custom WPD applications. Generic WPD drivers and clients of WPD devices should not use instances of this device interface class.

Custom applications can enumerate private devices that register this interface by calling `IPortableDeviceManager::GetPrivateDevices` (documented in Windows SDK).

For information about the device interface class for generic WPD devices, see [GUID\\_DEVINTERFACE\\_WPD](#).

## Requirements

Version	Available in Windows Vista, Windows XP, and later versions of Windows.
Header	<code>Portabledevice.h</code> (include <code>Portabledevice.h</code> )

## See also

[GUID\\_DEVINTERFACE\\_WPD](#)

# GUID\_DEVINTERFACE\_WRITEONCEDISK

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DEVINTERFACE\_WRITEONCEDISK [device interface class](#) is defined for write-once disk devices.

ATTRIBUTE	SETTING
Identifier	GUID_DEVINTERFACE_WRITEONCEDISK
Class GUID	{53F5630C-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The system-supplied [storage drivers](#) register instances of GUID\_DEVINTERFACE\_WRITEONCEDISK to notify the operating system and application of the presence of write-once disks, such as a CD-R.

[WriteOnceDiskClassGuid](#) is an obsolete identifier for the GUID\_DEVINTERFACE\_WRITEONCEDISK device interface class. For new instances of this class, use GUID\_DEVINTERFACE\_WRITEONCEDISK instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[WriteOnceDiskClassGuid](#)

# GUID\_DISPLAY\_ADAPTER\_INTERFACE

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_DISPLAY\_ADAPTER\_INTERFACE is an obsolete identifier for the device interface class for display adapter devices; for new instances of this class, use the [GUID\\_DEVINTERFACE\\_DISPLAY\\_ADAPTER](#) class identifier instead.

## Requirements

Header	Ntddvdeo.h (include Ntddvdeo.h)
--------	---------------------------------

## See also

[GUID\\_DEVINTERFACE\\_DISPLAY\\_ADAPTER](#)

# GUID\_DISPLAY\_DEVICE\_ARRIVAL

11/2/2020 • 2 minutes to read • [Edit Online](#)

The GUID\_DISPLAY\_DEVICE\_ARRIVAL device interface class is defined for [display adapters](#).

ATTRIBUTE	SETTING
Identifier	GUID_DISPLAY_DEVICE_ARRIVAL
Class GUID	{1CA05180-A699-450A-9A0C-DE4FBE3DDD89}

## Remarks

The system-supplied components of the [Windows Vista Display Driver Model](#) register instances of this device interface class to notify the operating system and applications of the presence of display adapters.

For information about the device interface class for display views that are supported by display adapters, see [GUID\\_DEVINTERFACE\\_DISPLAY\\_ADAPTER](#).

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Ntddvdeo.h (include Ntddvdeo.h)

## See also

[GUID\\_DEVINTERFACE\\_DISPLAY\\_ADAPTER](#)

# GUID\_IO\_VOLUME\_DEVICE\_INTERFACE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **GUID\_IO\_VOLUME\_DEVICE\_INTERFACE** [device interface class](#) is defined for volume devices.

ATTRIBUTE	SETTING
Identifier	GUID_IO_VOLUME_DEVICE_INTERFACE
Class GUID	{53F5630D-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The **GUID\_IO\_VOLUME\_DEVICE\_INTERFACE** identifier for this device interface class is an alias for the [GUID\\_DEVINTERFACE\\_VOLUME](#) device interface class.

## Requirements

Header	loevent.h (include loevent.h)
--------	-------------------------------

## See also

[GUID\\_DEVINTERFACE\\_VOLUME](#)

# GUID\_PARALLEL\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_PARALLEL\_DEVICE is an obsolete identifier for the [device interface class](#) for parallel ports that support an IEEE 1284-compatible hardware interface. For new instances of this class, use the [GUID\\_DEVINTERFACE\\_PARALLEL](#) class identifier instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddpar.h (include Ntddpar.h)

## See also

[GUID\\_DEVINTERFACE\\_PARALLEL](#)

# GUID\_PARCLASS\_DEVICE

12/5/2018 • 2 minutes to read • [Edit Online](#)

GUID\_PARCLASS\_DEVICE is an obsolete identifier for the device interface class for devices that are attached to a parallel port. For new instances of this class, use the [GUID\\_DEVINTERFACE\\_PARCLASS](#) class identifier instead.

## Requirements

Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	Ntddpar.h (include Ntddpar.h)

## See also

[GUID\\_DEVINTERFACE\\_PARCLASS](#)

# GUID\_SERENUM\_BUS\_ENUMERATOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

GUID\_SERENUM\_BUS\_ENUMERATOR is an obsolete identifier for the [device interface class](#) for Plug and Play (PnP) serial ports. Starting with Microsoft Windows 2000, use the

[GUID\\_DEVINTERFACE\\_SERENUM\\_BUS\\_ENUMERATOR](#) class identifier for new instances of this class.

## Remarks

The WDK includes the serial enumerator sample ([serenum](#)). The serial enumerator uses GUID\_SERENUM\_BUS\_ENUMERATOR to register instances of this device interface class. The serenum sample is included in the `src\kernel` directory of the WDK.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_SERENUM_BUS_ENUMERATOR instead.
Header	Ntddser.h (include Ntddser.h)

## See also

[GUID\\_DEVINTERFACE\\_SERENUM\\_BUS\\_ENUMERATOR](#)

# GUID\_VIRTUAL\_AVC\_CLASS

11/2/2020 • 2 minutes to read • [Edit Online](#)

The **GUID\_VIRTUAL\_AVC\_CLASS** [device interface class](#) is defined for virtual audio video control (AV/C) devices that are supported by the [AVStream](#) architecture.

ATTRIBUTE	SETTING
Identifier	GUID_VIRTUAL_AVC_CLASS
Class GUID	{616EF4D0-23CE-446D-A568-C31EB01913D0}

## Remarks

The system-supplied [AV/C client driver Avc.sys](#) registers an instance of **GUID\_VIRTUAL\_AVC\_CLASS** to represent a virtual AV/C device.

For information about the device interface class for AV/C units on a 1394 bus, see [GUID\\_AVC\\_CLASS](#).

## Requirements

Version	Available in Windows Vista, Microsoft Windows Server 2003, Windows XP and later versions of Windows.
Header	Avc.h (include Avc.h)

## See also

[GUID\\_AVC\\_CLASS](#)

# HardwareIds.scsi.ID Section of a TxtSetup.oem File

12/5/2018 • 2 minutes to read • [Edit Online](#)

A **HardwareIds.scsi.ID** section specifies the hardware IDs of the devices that a particular mass storage driver supports.

```
[HardwareIds.scsi.ID]
id = "deviceID","service"
...
```

## HardwareIds.scsi.ID

*ID* corresponds to an *ID* entry in the *HwComponent* section.

### *deviceID*

Specifies the device ID for a mass storage device.

### *service*

Specifies the service to be installed for the device. The service is specified by the file name of its executable image without a *.sys* extension.

The following example is an excerpt from a **HardwareIds.scsi.ID** section for a disk device:

```
; ...
[HardwareIds.scsi.oemscsi]
id = "PCI\VEN_9004&DEV_8111","oemscsi"

; ...
```

# headerPath XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **headerPath** XML element specifies the source file name for a custom header bitmap that DPInst displays in the upper-right corner of the DPInst EULA page and DPInst installation page.

## Element Tag

```
<headerPath>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b> or <b>language</b>
Child elements	None permitted
Data contents	String that specifies the file name of the header bitmap that DPInst displays in the upper-right corner of the DPInst EULA and installation pages. The header bitmap file must be located in the DPInst root directory, which is the directory that contains the DPInst executable file ( <i>DPInst.exe</i> ), or in a subdirectory under the DPInst root directory. If the header bitmap file is in a subdirectory, specify a fully qualified file name that is relative to the DPInst root directory.
Duplicate child elements	None permitted

## Remarks

A **headerPath** element is customized, but not localized, if it is a child element of a **dpinst** XML element. A **headerPath** element is customized and localized if it is a child element of a **language** XML element.

The following code example demonstrates a **headerPath** element that specifies *Data\Header.bmp* as the header bitmap file that DPInst displays in the upper-right corner of a DPInst EULA and installation pages and an installation page. The text that specifies the custom header bitmap file is shown in bold font style.

```
<dpinst>
  ...
  <headerPath>Data\Header.bmp</headerPath>
  ...
</dpinst>
```

If a **headerPath** element is not specified, DPInst uses a default header bitmap.

## See also

[dpinst](#)

[language](#)

# HwComponent Section of a *TxtSetup.oem* File

12/5/2018 • 2 minutes to read • [Edit Online](#)

A *HwComponent* section lists the drivers available for a particular component. There is a *HwComponent* section for each type of component supported by the file.

```
[HwComponent]
ID = description
...
```

## *HwComponent*

The name of the section must be one of the following system-defined values: **computer** or **scsi**.

### *ID*

Specifies a string, unique within this section, that identifies the option.

For each entry in this section, there must be a corresponding **Files.HwComponent.ID** section in the file.

For the **computer** component, the last three characters of the string determine which kernel Windows copies. If this string ends in **\_up**, Windows copies the uniprocessor kernel. If this string ends in **\_mp**, Windows copies the multiprocessor kernel. If the string does not end in **\_Xp**, Windows copies one or the other kernel, but does not guarantee which one.

### *description*

Specifies a string that Windows presents to the user in the menu of driver choices.

The following example shows a *HwComponent* section for a *TxtSetup.oem* file that supports one component (**scsi**) and offers two options:

```
; ...
[SCSI]           ; HwComponent section
oemscsi = "OEM Fast SCSI Controller"
oemscsi2 = "OEM Fast SCSI Controller 2"

; ...
```

# icon XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **icon** XML element specifies the source file for a custom icon that DPInst displays on the DPInst EULA page. DPInst uses this icon to represent DPInst on the Microsoft Windows taskbar and desktop. DPInst also uses this icon for the entry that represents a [driver package](#), which DPInst adds to **Programs and Features** in Control Panel

**Note** Prior to Windows Vista, DPInst added the entry for the driver package to **Add or Remove Programs** in Control Panel.

## Element Tag

```
<icon>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a> or <a href="#">language</a>
Child elements	None permitted
Data contents	String that specifies the source file that contains the icon that DPInst displays on a DPInst EULA page. The icon file must be located in the DPInst root directory, which is the directory that contains the DPInst executable file ( <i>DPInst.exe</i> ), or in a subdirectory under the DPInst root directory. If the icon file is in a subdirectory, specify a fully qualified file name that is relative to the DPInst root directory.
Duplicate child elements	None permitted

## Remarks

An **icon** element is customized, but not localized, if it is a child element of a [dpinst](#) XML element. An **icon** element is customized and localized if it is a child element of a [language](#) XML element.

The following code example demonstrates an **icon** element that specifies Data\Small.ico as the source of a custom icon that DPInst displays on the DPInst EULA page. The text that specifies the custom icon file is shown in bold font style.

```
<dpinst>
  ...
  <icon>Data\Eula.ico</icon>
  ...
</dpinst>
```

If an **icon** element is not specified, DPInst displays a default icon. The position of this icon cannot be changed.

## See also

[dpinst](#)

[language](#)

# installAllOrNone XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **installAllOrNone** XML element is an empty element that sets the **installAllOrNone** flag to ON, which configures DPInst to install drivers in a [driver package](#) only if all of the driver packages in the installation package can be installed or if all of driver packages in the driver package group can be installed.

## Element Tag

```
<installAllOrNone>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a> or <a href="#">group</a>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **installAllOrNone** flag is set to OFF. To set the **installAllOrNone** flag to ON for all of the [driver packages](#), including those in driver package groups, include an **installAllOrNone** element as a child element of a **dpinst** XML element, or use the /a command-line switch. To set the **installAllOrNone** flag to ON only for a specific driver package group, include an **installAllOrNone** element as child element of the corresponding **group** XML element.

The following code example demonstrates an **installAllOrNone** element that is a child element of a **dpinst** element.

```
<dpinst>
  ...
  <installAllOrNone/>
  ...
</dpinst>
```

The following code example demonstrates an **installAllOrNone** element that is a child element of a **group** element.

```
<dpinst>
  ...
  <group>
    <package path="DirAbc\Abc.inf" />
    <package path="DirDef\Def.inf" />
    <installAllOrNone/>
  <group/>
  ...
</dpinst>
```

## See also

[dpinst](#)

[group](#)

# installHeaderTitle XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **installHeaderTitle** XML element customizes the bold text of the installation header title that appears on a DPInst installation page.

## Element Tag

```
<installHeaderTitle>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the title text of an installation page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates an **installHeaderTitle** element that customizes the title text of an installation page. The text that specifies the custom install header title is shown in bold font style.

```
<dpinst>
  ...
  <language code="0x0409">
    ...
    <installHeaderTitle>Installing the software for your Toaster device...</installHeaderTitle>
    ...
  </language>
  ...
</dpinst>
```

If an **installHeaderTitle** element is not specified, DPInst displays the default title text that is shown on the default installation page.

## See also

[language](#)

# InstallSelectedDriver function

11/2/2020 • 4 minutes to read • [Edit Online](#)

The **InstallSelectedDriver** function is deprecated. For Windows Vista and later, use [DiInstallDevice](#) instead.

## Syntax

```
BOOL WINAPI InstallSelectedDriver(
    _In_     HWND      hwndParent,
    _In_     HDEVINFO  DeviceInfoSet,
    _In_     LPCTSTR   Reserved,
    _In_     BOOL      Backup,
    _Out_    PDWORD    bReboot
);
```

## Parameters

### *hwndParent* [in]

A handle to the top-level window that the **InstallSelectedDriver** function uses to display user interface components that are associated with installing the driver.

### *DeviceInfoSet* [in]

A handle to a [device information set](#) that contains a device information element that represents a selected device and a selected driver for the device. For more information about how to select a device and a driver for a device, see the following **Remarks** section.

### *Reserved* [in]

This parameter must be set to **NULL**.

### *Backup* [in]

A value of type **BOOL** that determines whether **InstallSelectedDriver** backs up the currently installed drivers for the selected device before the function installs the selected driver for the device. If the currently installed drivers are backed up and the user encounters a problem with the new driver, the user can roll back the installation of the new driver to the backed up driver. If the currently installed drivers are not backed up, the user cannot roll back the installation of the new driver to the previously installed driver. If *Backup* is set to **TRUE**, **InstallSelectedDriver** backs up the currently installed drivers; otherwise, the function does not back up the currently installed drivers. For more information about driver back up, see [DiRollbackDriver](#).

### *bReboot* [out]

A pointer to a variable of type **DWORD** that **InstallSelectedDriver** sets to indicate whether a system restart is required to complete the installation. If the variable is set to **DI\_NEEDREBOOT**, a system restart is required; otherwise, a system restart is not required. The caller is responsible for restarting the system.

## Return value

**InstallSelectedDriver** returns **TRUE** if the selected driver was installed on the selected device; otherwise, the function returns **FALSE** and the logged error can be retrieved by making a call to [GetLastError](#).

Some of the more common error values that [GetLastError](#) might return are as follows:

RETURN CODE	DESCRIPTION
NO_ERROR	The selected driver was a better match to the driver than the driver that was previously installed on the device.
ERROR_IN_WOW64	The calling application is a 32-bit application that is attempting to execute in a 64-bit environment, which is not allowed. For more information, see <a href="#">Installing Devices on 64-Bit Systems</a> .

## Remarks

To access **InstallSelectedDriver**, call **LoadLibrary** to load *Newdev.dll* and then call **GetProcAddress** to obtain a function pointer to **InstallSelectedDriver**.

You should only call **InstallSelectedDriver** if it is necessary to install a specific driver on a specific device.

**Important** For Windows Vista and later versions of Windows, call **DlInstallDevice** instead of **InstallSelectedDriver** to perform this type of operation.

Other than the special applications that require the installation of a specific driver on a specific device, an installation application should install the driver that is the best match for a device. To install the driver that is the best match for a device, call **DlInstallDriver** or **UpdateDriverForPlugAndPlayDevices**. For more information about which of these functions to call to install a driver on a device, see [SetupAPI Functions that Simplify Driver Installation](#).

Before calling **InstallSelectedDriver**, the caller must obtain a device information set that contains the device, select the device in the set, and select a driver for the device.

To create a device information set that contains the device, do one of the following:

- Call **SetupDiGetClassDevs** to retrieve a device information set that contains the device and then call **SetupDiEnumDeviceInfo** to enumerate the devices in the device information set. On each call, **SetupDiEnumDeviceInfo** returns an **SP\_DEVINFO\_DATA** structure that represents the enumerated device in the device information set. To obtain specific information about the enumerated device, call **SetupDiGetDeviceRegistryProperty** and supply the **SP\_DEVINFO\_DATA** structure that was returned by **SetupDiEnumDeviceInfo**.
  - OR -
- Call **SetupDiOpenDeviceInfo** to add a device with a known device instance ID to the device information set. **SetupDiOpenDeviceInfo** returns an **SP\_DEVINFO\_DATA** structure that represents the device in the device information set.

After obtaining the **SP\_DEVINFO\_DATA** structure for a device, call **SetupDiSetSelectedDevice** to select the device in the device information set.

To retrieve a driver for a device, call **SetupDiBuildDriverInfoList** to build a list of compatible drivers for the device and then call **SetupDiEnumDriverInfo** to enumerate the elements of the driver list for the device. For each enumerated driver, **SetupDiEnumDriverInfo** retrieves an **SP\_DRVINFO\_DATA** structure that represents the driver. **SetupDiGetDriverInfoDetail** can be called to retrieve additional details about an enumerated driver.

After obtaining an **SP\_DRVINFO\_DATA** structure for the driver, call **SetupDiSetSelectedDriver** to select the driver for the device.

## Requirements

Target platform	Desktop
Version	Available in Microsoft Windows 2000 and later versions of Windows.
Header	None (The <b>InstallSelectedDriver</b> function is not defined in a public header file. For more information, see the <b>Remarks</b> section.)
Library	Newdev.lib
DLL	Newdev.dll

## See also

[DiInstallDevice](#)

[DiInstallDriver](#)

[SetupDiBuildDriverInfoList](#)

[SetupDiEnumDeviceInfo](#)

[SetupDiEnumDriverInfo](#)

[SetupDiGetClassDevs](#)

[SetupDiGetDeviceRegistryProperty](#)

[SetupDiGetDriverInfoDetail](#)

[SetupDiOpenDeviceInfo](#)

[SetupDiSetSelectedDevice](#)

[SetupDiSetSelectedDriver](#)

[UpdateDriverForPlugAndPlayDevices](#)

# KSCATEGORY\_ACOUSTIC\_ECHO\_CANCEL

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_ACOUSTIC\_ECHO\_CANCEL device interface class is defined for the [kernel streaming \(KS\)](#) functional category that performs acoustic echo cancellation.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_ACOUSTIC_ECHO_CANCEL
Class GUID	{BF963D80-C559-11D0-8A2B-00A0C9255AC1}

## Remarks

Drivers for KS audio devices register instances of KSCATEGORY\_ACOUSTIC\_ECHO\_CANCEL to indicate to the operating system that the devices support the KS functional category that performs acoustic echo cancellation.

For information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_AUDIO

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_AUDIO device interface class is defined for the [kernel streaming](#) (KS) functional category for an audio device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_AUDIO
Class GUID	{6994AD04-93EF-11D0-A3CC-00A0C9223196}

## Remarks

Drivers for KS audio devices register instances of this device interface class to indicate to the operating system that the devices support the KSCATEGORY\_AUDIO functional category.

For information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

For information about how to register this functional category in an INF file, see the Help files */NFViewer.htm* and *ac97smpl.inf*, which are included with the [AC'97 sample driver](#) in the WDK.

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_AUDIO\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_AUDIO\_DEVICE [device interface class](#) represents a [kernel streaming](#) (KS) functional category that is reserved for exclusive use by the system-supplied [WDM audio components](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_AUDIO_DEVICE
Class GUID	{FBF6F530-07B9-11D2-A71E-0000F8004788}

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_AUDIO\_GFX

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_AUDIO\_GFX device interface class is defined for the [kernel streaming \(KS\)](#) functional category that supports a [global effects \(GFX\)](#) filter.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_AUDIO_GFX
Class GUID	{9BAF9572-340C-11D3-ABDC-00A0C90AB16F}

## Remarks

Drivers for KS audio adapter devices register instances of KSCATEGORY\_AUDIO\_GFX to indicate to the operating system that the devices support the KSCATEGORY\_AUDIO\_GFX functional category.

For information about other device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

## Requirements

Version	Available in Windows Server 2003, Windows XP, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_AUDIO\_SPLITTER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_AUDIO\_SPLITTER device interface class is designed for a [kernel streaming](#) (KS) functional category that is reserved for exclusive use by the system-supplied [WDM audio components](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_AUDIO_SPLITTER
Class GUID	{9EA331FA-B91B-45F8-9285-BD2BC77AFCDE}

## Requirements

Version	Available in Windows Server 2003, Windows XP, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_BDA\_IP\_SINK

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_IP\_SINK device interface class is defined for the [kernel streaming \(KS\)](#) functional category for a sink filter in the [broadcast driver architecture \(BDA\)](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_IP_SINK
Class GUID	{71985F4A-1CA1-11d3-9CC8-00C04F7971E0}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_IP\_SINK to indicate that the devices support a BDA IP sink filter.

For more information, see [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000 with DirectX 9.0A installed, and later versions of Windows.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BDA\_NETWORK\_EPG

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_NETWORK\_EPG device interface class is defined for the **kernel streaming** (KS) functional category for an electronic program guide (EPG) in the **broadcast driver architecture** (BDA).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_NETWORK_EPG
Class GUID	{71985F49-1CA1-11d3-9CC8-00C04F7971E0}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_NETWORK\_EPG to indicate to the operating system that the devices support a BDA EPG filter.

For more information, see [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows XP, Windows 2000 with DirectX 9.0A installed, and later versions of Windows.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BDA\_NETWORK\_PROVIDER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_NETWORK\_PROVIDER device interface class is defined for the [kernel streaming](#) (KS) functional category for a network provider in the [broadcast driver architecture](#) (BDA).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_NETWORK_PROVIDER
Class GUID	{71985F4B-1CA1-11d3-9CC8-00C04F7971E0}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_NETWORK\_PROVIDER to indicate to the operating system that the devices support a BDA network provider filter.

For more information, see [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows XP and later versions of Windows. Available in Windows 2000 with DirectX 9.0A installed.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BDA\_NETWORK\_TUNER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_NETWORK\_TUNER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for a network tuner in the [broadcast driver architecture](#) (BDA).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_NETWORK_TUNER
Class GUID	{71985F48-1CA1-11d3-9CC8-00C04F7971E0}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_NETWORK\_TUNER to indicate to the operating system that the devices support a BDA network tuner filter.

For an example of how to register this functional category in an INF file, see the INF file *BDASwTunerATSC.inf*. *BDASwTunerATSC.inf* is included with the BDA sample generic tuner in the *src\swtuner\BDA\Atuner\gentuner* subdirectory of the WDK.

For more information about the KS functional category for the network tuner filters, see [Common Control Nodes and Filters](#) and [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows XP, Windows 2000 with DirectX 9.0A installed, and later versions of Windows.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BDA\_RECEIVER\_COMPONENT

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_RECEIVER\_COMPONENT device interface class is defined for the [kernel streaming](#) (KS) functional category for a receiver in the [broadcast driver architecture](#) (BDA).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_RECEIVER_COMPONENT
Class GUID	{FD0A5AF4-B41D-11d2-9C95-00C04F7971E0}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_RECEIVER\_COMPONENT to indicate to the operating system that the devices support a BDA receiver filter.

For more information about the KS functional category for a BDA receiver filters, see [Common Control Nodes and Filters](#), [Starting a BDA Minidriver](#), and [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows XP, Windows 2000 with DirectX 9.0A installed, and later versions of Windows.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BDA\_TRANSPORT\_INFORMATION

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BDA\_TRANSPORT\_INFORMATION [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for a transport information filter (TIF) in the [broadcast driver architecture](#) (BDA).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BDA_TRANSPORT_INFORMATION
Class GUID	{A2E3074F-6C3D-11d3-B653-00C04F79498E}

## Remarks

Drivers for BDA devices register instances of KSCATEGORY\_BDA\_TRANSPORT\_INFORMATION to indicate to the operating system that the devices support a BDA transport information filter.

For more information about the KS functional category for BDA transport information filters, see [BDA Filter Category GUIDs](#).

## Requirements

Version	Available in Windows XP, Windows 2000 with DirectX 9.0A installed, and later versions of Windows.
Header	Bdamedia.h (include Bdamedia.h)

# KSCATEGORY\_BRIDGE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_BRIDGE device interface class is defined for the [kernel streaming](#) (KS) functional category that supports a software interface between the KS subsystem and another software component.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_BRIDGE
Class GUID	{085AFF00-62CE-11CF-A5D6-28DB04C10000}

## Remarks

Drivers for KS audio adapter devices register instances of KSCATEGORY\_BRIDGE to indicate to the operating system that the devices support the KSCATEGORY\_BRIDGE functional category.

For more information about KSCATEGORY\_BRIDGE functional category, see [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_CAPTURE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_CAPTURE device interface class is defined for the kernel streaming (KS) functional category that captures wave or MIDI data streams.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_CAPTURE
Class GUID	{65E8773D-8F56-11D0-A3B9-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_CAPTURE to indicate that the devices support the KSCATEGORY\_CAPTURE functional category.

For information about how to register this functional category in an INF file, see the *Ac97smpl.inf* INF file that is included with the [AC'97 sample driver](#) that is provided in the WDK.

For information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_CLOCK

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_CLOCK device interface class is defined for the [kernel streaming](#) (KS) functional category for a clock device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_CLOCK
Class GUID	{53172480-4791-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_CLOCK to indicate to the operating system that the devices support the KSCATEGORY\_CLOCK functional category.

For more information about kernel streaming clocks, see [KS Minidriver Architecture](#), [KS Clocks](#), and [AVStream Clocks](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_COMMUNICATIONSTRANSFORM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_COMMUNICATIONSTRANSFORM [device interface class](#) is defined for the [kernel streaming \(KS\)](#) functional category for a communication transform device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_COMMUNICATIONSTRANSFORM
Class GUID	{CF1DDA2C-9743-11D0-A3EE-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_COMMUNICATIONSTRANSFORM to indicate to the operating system that the devices support the KSCATEGORY\_COMMUNICATIONSTRANSFORM functional category.

The KSCATEGORY\_COMMUNICATIONSTRANSFORM functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_CROSSBAR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_CROSSBAR [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for a crossbar device that routes video and audio streams.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_CROSSBAR
Class GUID	{A799A801-A46D-11D0-A18C-00A02401DCD4}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_CROSSBAR to indicate to the operating system that the devices support the KSCATEGORY\_CROSSBAR functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src\swtuner\algtuner* directory of the WDK.

For information about crossbar devices for audio and video, see [Filters Used with the Video Capture Devices and Analog Video Category](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_DATACOMPRESSOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_DATACOMPRESSOR [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that compresses a data stream.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_DATACOMPRESSOR
Class GUID	{1E84C900-7E70-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_DATACOMPRESSOR to indicate to the operating system that the devices support the KSCATEGORY\_DATACOMPRESSOR functional category.

The KSCATEGORY\_DATACOMPRESSOR functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

For information about the device interface class that is defined for the KS functional category that decompresses a data stream, see [KSCATEGORY\\_DATADECOMPRESSOR](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSCATEGORY\\_DATADECOMPRESSOR](#)

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_DATADECOMPRESSOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_DATADECOMPRESSOR [device interface class](#) is defined for the [kernel streaming \(KS\)](#) functional category that decompresses a data stream.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_DATADECOMPRESSOR
Class GUID	{2721AE20-7E70-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_DATADECOMPRESSOR to indicate to the operating system that the devices support the KSCATEGORY\_DATADECOMPRESSOR functional category.

The KSCATEGORY\_DATADECOMPRESSOR functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

For information about the device interface class that is defined for the KS functional category that compresses a data stream, see [KSCATEGORY\\_DATACOMPRESSOR](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSCATEGORY\\_DATACOMPRESSOR](#)

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_DATATRANSFORM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_DATATRANSFORM device interface class is defined for the [kernel streaming \(KS\)](#) functional category that transforms audio data streams.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_DATATRANSFORM
Class GUID	{2EB07EA0-7E70-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_DATATRANSFORM to indicate to the operating system that the devices support the KSCATEGORY\_DATATRANSFORM functional category.

For an example of how to register this functional category in an INF file, see the *Ddksynth.inf* INF file that is included with the software synthesizer sample in the *src\audio\ddksynth* directory of the WDK.

For more information about this functional category, see [Installing Device Interfaces for an Audio Adapter](#), [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#), and [Requirements for a GFX Filter Factory](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_DRM\_DESCRAMBLE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_DRM\_DESCRAMBLE [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that unscrambles a DRM-protected wave stream.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_DRM_DESCRAMBLE
Class GUID	{FFBB6E3F-CCFE-4D84-90D9-421418B03A8E}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_DRM\_DESCRAMBLE to indicate to the operating system that the devices support the KSCATEGORY\_DRM\_DESCRAMBLE functional category.

For more information about this functional category, see [Installing Device Interfaces for an Audio Adapter](#).

## Requirements

Version	Available in Windows Vista, Windows Server 2003, Windows XP, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_ENCODER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_ENCODER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that encodes data.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_ENCODER
Class GUID	{19689BF6-C384-48fd-AD51-90E58C79F70B}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_ENCODER to indicate to the operating system that the devices support the KSCATEGORY\_ENCODER functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src/swtuner/algtuner* directory of the WDK.

For information about encoders, see [Encoder Devices](#) and [Encoder Installation and Registration](#).

## Requirements

Version	Available in Windows Vista, Windows Server 2003, Windows XP Service Pack 1 (SP1), and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_ESCALANTE\_PLATFORM\_DRIVER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_ESCALANTE\_PLATFORM\_DRIVER [device interface class](#) is obsolete. Do not use for new devices.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_ESCALANTE_PLATFORM_DRIVER
Class GUID	{74F3AEA8-9768-11D1-8E07-00A0C95EC22E}

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_FILESYSTEM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_FILESYSTEM [device interface class](#) is defined for the [kernel streaming \(KS\)](#) functional category that moves a data stream into or out of the file system.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_FILESYSTEM
Class GUID	{760FED5E-9357-11D0-A3CC-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_FILESYSTEM to indicate to the operating system that the devices support the KSCATEGORY\_FILESYSTEM functional category.

The KSCATEGORY\_FILESYSTEM functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#) functional categories.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_INTERFACETRANSFORM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_INTERFACETRANSFORM device interface class is defined for the [kernel streaming \(KS\)](#) functional category that transforms the interface of a device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_INTERFACETRANSFORM
Class GUID	{CF1DDA2D-9743-11D0-A3EE-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_INTERFACETRANSFORM to indicate to the operating system that the devices support the KSCATEGORY\_INTERFACETRANSFORM functional category.

The KSCATEGORY\_INTERFACETRANSFORM functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#) functional categories.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_MEDIUMTRANSFORM

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_MEDIUMTRANSFORM [device interface class](#) is defined for the [kernel streaming \(KS\)](#) functional category that transforms the type of medium that is being used.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_MEDIUMTRANSFORM
Class GUID	{CF1DDA2E-9743-11D0-A3EE-00A0C9223196}

## Remarks

Drivers for KS devices register an instance of KSCATEGORY\_MEDIUMTRANSFORM to indicate to the operating system that the devices support the KSCATEGORY\_MEDIUMTRANSFORM functional category.

The KSCATEGORY\_MEDIUMTRANSFORM functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#) functional categories.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_MICROPHONE\_ARRAY\_PROCESSOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_MICROPHONE\_ARRAY\_PROCESSOR [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that processes input from a microphone array.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_MICROPHONE_ARRAY_PROCESSOR
Class GUID	{830A44F2-A32D-476B-BE97-42845673B35A}

## Remarks

Drivers for KS devices register an instance of KSCATEGORY\_MICROPHONE\_ARRAY\_PROCESSOR to indicate to the operating system that the devices support the KSCATEGORY\_MICROPHONE\_ARRAY\_PROCESSOR functional category.

For more information about functional categories for audio devices, see [Installing Device Interfaces for an Audio Adapter](#) and [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

For more information about how to process a microphone array in Windows Vista, refer to the [Microphone Array Support in Windows Vista](#) and the [How to Build and Use Microphone Arrays for Windows Vista](#) white papers.

## Requirements

Version	Available in Windows Vista, Windows Server 2003, Windows XP, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_MIXER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_MIXER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that mixes data streams.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_MIXER
Class GUID	{AD809C00-7B88-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_MIXER to indicate to the operating system that the devices support the KSCATEGORY\_MIXER functional category.

For more information about this functional category and other functional categories, see [Installing Device Interfaces for an Audio Adapter](#) and [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_MULTIPLEXER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_MULTIPLEXER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for a multiplexer device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_MULTIPLEXER
Class GUID	{7A5DE1D3-01A1-452c-B481-4FA2B96271E8}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_MULTIPLEXER to indicate to the operating system that the devices support the KSCATEGORY\_MULTIPLEXER functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src/swtuner/algtuner* directory of the WDK.

For information about multiplexers, see [Topology Filters](#).

For more information about the KSCATEGORY\_MULTIPLEXER functional category, see [Encoder Installation and Registration](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_NETWORK

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_NETWORK device interface class is defined for the [kernel streaming \(KS\)](#) functional category for a network device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_NETWORK
Class GUID	{67C9CC3C-69C4-11D2-8759-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_NETWORK to indicate to the operating system that the devices support the KSCATEGORY\_NETWORK functional category.

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_NETWORK\_CAMERA

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_NETWORK\_CAMERA GUID is defined as follows:

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_NETWORK_CAMERA
Class GUID	{B8238652-B500-41EB-B4F3-4234F7F5AE99}

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_PREFERRED\_MIDIOUT\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_PREFERRED\_MIDIOUT\_DEVICE [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for the preferred MIDI output device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_PREFERRED_WAVEOUT_DEVICE
Class GUID	{D6C50674-72C1-11D2-9755-0000F8004788}

## Remarks

A user selects the preferred MIDI output device in the Multimedia property pages in the Control Panel.

This functional category is reserved for exclusive use by the system-supplied [WDM Audio Components](#).

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000 and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_PREFERRED\_WAVEIN\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_PREFERRED\_WAVEIN\_DEVICE device interface class is defined for the [kernel streaming \(KS\)](#) functional category for the preferred wave input device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_PREFERRED_WAVEIN_DEVICE
Class GUID	{D6C50671-72C1-11D2-9755-0000F8004788}

## Remarks

A user selects the preferred wave input device in the Multimedia property pages in the Control Panel.

This functional category is reserved for exclusive use by the system-supplied [WDM Audio Components](#).

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000 and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_PREFERRED\_WAVEOUT\_DEVICE

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_PREFERRED\_WAVEIN\_DEVICE device interface class is defined for the [kernel streaming \(KS\)](#) functional category for the preferred wave input device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_PREFERRED_WAVEOUT_DEVICE
Class GUID	{D6C5066E-72C1-11D2-9755-0000F8004788}

## Remarks

A user selects the preferred wave input device in the Multimedia property pages in the Control Panel.

This functional category is reserved for exclusive use by the system-supplied [WDM Audio Components](#).

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000 and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_PROXY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_PROXY device interface class represents a kernel streaming (KS) functional category that is reserved for exclusive use by the [kernel streaming proxy module](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_PROXY
Class GUID	{97EBAACA-95BD-11D0-A3EA-00A0C9223196}

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_QUALITY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_QUALITY device interface class is defined for the [kernel streaming](#) (KS) functional category for quality management.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_QUALITY
Class GUID	{97EBAACB-95BD-11D0-A3EA-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_QUALITY to indicate to the operating system that the devices support the KSCATEGORY\_QUALITY functional category.

For more information, see [Quality Management](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_REALTIME

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_REALTIME device interface class is defined for the [kernel streaming](#) (KS) functional category for an audio device that is connected to a system bus (for example, the PCI bus) and that plays back or captures wave data in real time.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_REALTIME
Class GUID	{EB115FFC-10C8-4964-831D-6DCB02E6F23F}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_REALTIME to indicate to the operating system that the devices support the KSCATEGORY\_REALTIME functional category.

Devices that register this functional category are operated by the system-supplied [WaveRT port driver](#).

For information about how to register this functional category in an INF file, see the INF file *Ac97smpl.inf* that is included with the [AC'97 sample driver](#) in the WDK.

## Requirements

Version	Available in Windows Vista and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_RENDER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_RENDER device interface class is defined for the [kernel streaming \(KS\)](#) functional category that renders wave and MIDI data streams.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_RENDER
Class GUID	{65E8773E-8F56-11D0-A3B9-00A0C9223196}

## Remarks

Drivers for KS audio adapter devices register an instance of KSCATEGORY\_RENDER to indicate that the devices support the KSCATEGORY\_RENDER functional category.

For information about how to register this functional category in an INF file, see the INF file *Ac97smpl.inf* that is included with the [AC'97 sample driver](#) in the WDK.

For information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#) and [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_SENSOR\_CAMERA

12/5/2018 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_SENSOR\_CAMERA GUID is defined as follows:

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_SENSOR_CAMERA
Class GUID	{24E552D7-6523-47F7-A647-D3465BF1F5CA}

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_SPLITTER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_SPLITTER device interface class is defined for the [kernel streaming](#) (KS) functional category that splits a data stream.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_SPLITTER
Class GUID	{0A4252A0-7E70-11D0-A5D6-28DB04C10000}

## Remarks

Drivers for KS audio adapter devices register an instance of KSCATEGORY\_SPLITTER to indicate to the operating system that the devices support the KSCATEGORY\_SPLITTER functional category.

The KSCATEGORY\_SPLITTER functional category is one of the [KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#) functional categories.

For general information about splitters, see [AVStream Splitters](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSPROPERTY\\_TOPOLOGY\\_CATEGORIES](#)

# KSCATEGORY\_SYNTHESIZER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_SYNTHESIZER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category that converts MIDI data to either wave audio samples or an analog output signal.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_SYNTHESIZER
Class GUID	{DFF220F3-F70F-11D0-B917-00A0C9223196}

## Remarks

Drivers for KS audio adapter devices register instances of KSCATEGORY\_SYNTHESIZER to indicate to the operating system that the devices support the KSCATEGORY\_SYNTHESIZER functional category.

For an example of how to register this functional category in an INF file, see the *Ddksynth.inf* INF file that is included with the software synthesizer sample in the *src\audio\ddksynth* directory of the WDK.

For general information about synthesizers, see [MIDI and DirectMusic Filters](#).

For general information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_SYSAUDIO

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_SYSAUDIO device interface class represents a [kernel streaming](#) (KS) functional category that is reserved for exclusive use by the system-supplied [WDM audio components](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_SYSAUDIO
Class GUID	{A7C7A5B1-5AF3-11D1-9CED-00A024BF0407}

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000, and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSCATEGORY\_TEXT

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_TEXT device interface class is defined for the [kernel streaming](#) (KS) functional category that supports text data.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_TEXT
Class GUID	{6994AD06-93EF-11D0-A3CC-00A0C9223196}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_TEXT to indicate that the devices support the KSCATEGORY\_TEXT functional category.

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_TOPOLOGY

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_TOPOLOGY [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for the internal topology of an audio device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_TOPOLOGY
Class GUID	{DDA54A40-1E4C-11D1-A050-405705C10000}

## Remarks

Drivers for KS audio adapter devices register instances of KSCATEGORY\_TOPOLOGY to indicate to the operating system that the devices support the KSCATEGORY\_TOPOLOGY functional category.

For information about device interface classes for audio adapters, see [Installing Device Interfaces for an Audio Adapter](#).

The [AC'97 sample driver](#) that is provided in the WDK enumerates instances of the KSCATEGORY\_TOPOLOGY device interface class.

The sysfx sample in the WDK registers instances of this device interface class. The sysfx sample is located in the *src\audio\sysfx* directory of the WDK.

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_TVAUDIO

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_TVAUDIO device interface class is defined for the [kernel streaming \(KS\)](#) functional category for a TV audio device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_TVAUDIO
Class GUID	{A799A802-A46D-11D0-A18C-00A02401DCD4}

## Remarks

Drivers for KS devices register instances of this KSCATEGORY\_TVAUDIO to indicate to the operating system that the devices support the KSCATEGORY\_TVAUDIO functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src/swtuner/algtuner* directory of the WDK.

For information about video devices, see [Video Capture Devices](#), [Filter Graph Examples](#), and [Encoder Devices](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

## See also

[KSCATEGORY\\_TVTUNER](#)

# KSCATEGORY\_TVTUNER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_TVTUNER [device interface class](#) is defined for the [kernel streaming](#) (KS) functional category for a TV tuner device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_TVTUNER
Class GUID	{A799A800-A46D-11D0-A18C-00A02401DCD4}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_TVTUNER to indicate to the operating system that the devices support the KSCATEGORY\_TVTUNER functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src/swtuner/algtuner* directory of the WDK.

For information about video devices, see [Video Capture Devices](#), [Filter Graph Examples](#), and [Encoder Devices](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

## See also

[KSCATEGORY\\_TVAUDIO](#)

# KSCATEGORY\_VBICODEC

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_VBICODEC device interface class is defined for the [kernel streaming](#) (KS) functional category for a video blanking interval (VBI) codec device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_VBICODEC
Class GUID	{07DAD660-22F1-11D1-A9F4-00C04FBBDE8F}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_VBICODEC to indicate to the operating system that the devices support the KSCATEGORY\_VBICODEC functional category.

For general information about video devices, see [Video Capture Devices](#).

For more information about video blanking, see [Streaming Data from a Video Capture Device](#) and [VBI Category](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

## See also

[KSCATEGORY\\_VIDEO](#)

# KSCATEGORY\_VIDEO

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_VIDEO device interface class is defined for the [kernel streaming \(KS\)](#) functional category for a video device.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_VIDEO
Class GUID	{6994AD05-93EF-11D0-A3CC-00A0C9223196}

## Remarks

Drivers for KS video devices register instances of KSCATEGORY\_VIDEO to indicate to the operating system that the devices support the KSCATEGORY\_VIDEO functional category.

For an example of how to register this functional category in an INF file, see the *Bdan.inf* INF file that is included with the software tuner sample in the *src/swtuner/algtuner* directory of the WDK.

For more information about this functional category, see [Providing a UVC INF File](#).

For general information about video devices, see [Video Capture Devices](#).

For information about other device interface classes for video devices, see [KSCATEGORY\\_TVAUDIO](#) and [KSCATEGORY\\_TVTUNER](#).

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

## See also

[KSCATEGORY\\_TVAUDIO](#)

[KSCATEGORY\\_TVTUNER](#)

# KSCATEGORY\_VIDEO\_CAMERA

12/5/2018 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_VIDEO\_CAMERA GUID is defined as follows:

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_VIDEO_CAMERA
Class GUID	{E5323777-F976-4f5b-9B55-B94699C46E44}

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSCATEGORY\_VIRTUAL

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_VIRTUAL device interface class represents a [kernel streaming](#) (KS) category that is reserved for exclusive use by the system-supplied [WDM audio components](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_VIRTUAL
Class GUID	{3503EAC4-1F26-11D1-8AB0-00A0C9223196}

## Requirements

Header	Ksmedia.h (include Ksmedia.h)
--------	-------------------------------

# KSCATEGORY\_VPMUX

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_VPMUX device interface class is defined for the [kernel streaming](#) (KS) functional category that supports video multiplexing.

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_VPMUX
Class GUID	{A799A803-A46D-11D0-A18C-00A02401DCD4}

## Remarks

Drivers for KS devices register instances of KSCATEGORY\_VPMUX to indicate to the operating system that the devices support the KSCATEGORY\_VPMUX functional category.

For general information about video devices, see [Video Capture Devices](#).

For information about the device interface class for video devices, see [KSCATEGORY\\_VIDEO](#).

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

## See also

[KSCATEGORY\\_VIDEO](#)

# KSCATEGORY\_WDMAUD

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSCATEGORY\_WDMAUD device interface class represents a [kernel streaming](#) (KS) functional category that is reserved for exclusive use by the system-supplied [WDM audio components](#).

ATTRIBUTE	SETTING
Identifier	KSCATEGORY_WDMAUD
Class GUID	{3E227E76-690D-11D2-8161-0000F8775BF1}

## Requirements

Version	Available in Windows Server 2003, Windows XP, Windows 2000 and later versions of Windows.
Header	Ksmedia.h (include Ksmedia.h)

# KSMFT\_CATEGORY\_AUDIO\_DECODER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_AUDIO\_DECODER [device interface class](#) is defined for the [Kernel Streaming \(KS\)](#) functional category for an audio device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_AUDIO_DECODER
Class GUID	{9ea73fb4-ef7a-4559-8d5d-719d8f0426c7}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_AUDIO\_DECODER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_AUDIO\_EFFECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_AUDIO\_EFFECT device interface class is defined for the [Kernel Streaming \(KS\)](#) functional category for an audio device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_AUDIO_EFFECT
Class GUID	{11064c48-3648-4ed0-932e-05ce8ac811b7}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_AUDIO\_EFFECT functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_AUDIO\_ENCODER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_AUDIO\_ENCODER [device interface class](#) is defined for the [Kernel Streaming \(KS\)](#) functional category for an audio device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_AUDIO_ENCODER
Class GUID	{91c64bd0-f91e-4d8c-9276-db248279d975}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_AUDIO\_ENCODER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_DEMULTIPLEXER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_DEMULTIPLEXER [device interface class](#) is defined for the [Kernel Streaming \(KS\)](#) functional category for a device that separates (*demultiplexes*) media streams.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_DEMULTIPLEXER
Class GUID	{a8700a7a-939b-44c5-99d7-76226b23b3f1}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_DEMULTIPLEXER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_MULTIPLEXER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_MULTIPLEXER device interface class is defined for the [Kernel Streaming \(KS\)](#) functional category for a device that combines (*multiplexes*) media streams.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_MULTIPLEXER
Class GUID	{059c561e-05ae-4b61-b69d-55b61ee54a7b}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_MULTIPLEXER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_OTHER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_OTHER device interface class is defined for the [Kernel Streaming \(KS\)](#) functional category for a device that does not belong in other KS functional categories.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_OTHER
Class GUID	{90175d57-b7ea-4901-aeb3-933a8747756f}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_OTHER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_VIDEO\_DECODER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_VIDEO\_DECODER [device interface class](#) is defined for the [Kernel Streaming \(KS\)](#) functional category for a video device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_VIDEO_DECODER
Class GUID	{d6c02d4b-6833-45b4-971a-05a4b04bab91}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_VIDEO\_DECODER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_VIDEO\_EFFECT

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_VIDEO\_EFFECT [device interface class](#) is defined for the [Kernel Streaming](#) (KS) functional category for a video device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_VIDEO_EFFECT
Class GUID	{12e17c21-532c-4a6e-8a1c-40825a736397}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_VIDEO\_EFFECT functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_VIDEO\_ENCODER

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_VIDEO\_ENCODER device interface class is defined for the [Kernel Streaming \(KS\)](#) functional category for a video device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_VIDEO_ENCODER
Class GUID	{f79eac7d-e545-4387-bdee-d647d7bde42a}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_VIDEO\_ENCODER functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# KSMFT\_CATEGORY\_VIDEO\_PROCESSOR

11/2/2020 • 2 minutes to read • [Edit Online](#)

The KSMFT\_CATEGORY\_VIDEO\_PROCESSOR device interface class is defined for the [Kernel Streaming \(KS\)](#) functional category for a video device.

ATTRIBUTE	SETTING
Identifier	KSMFT_CATEGORY_VIDEO_PROCESSOR
Class GUID	{302ea3fc-aa5f-47f9-9f7a-c2188bb16302}

## Remarks

AVStream drivers that have MFT codec support register instances of this device interface class to indicate to the operating system that the devices support the KSMFT\_CATEGORY\_VIDEO\_PROCESSOR functional category.

For more information about device interface classes for AVStream devices with hardware codec support, see [Getting Started with Hardware Codec Support in AVStream](#).

For more information about how to register this functional category in an INF file, see the *Hiddigi.inf* file, which is included with the *src\input\hiddigi* sample drivers in the WDK.

## Requirements

Header	Ks.h (include Ks.h)
--------	---------------------

# language XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **language** XML element localizes and customizes the items that DPInst displays on its wizard pages.

## Element Tag

```
<language>
```

## XML Attributes

<b>Code</b>	A language identifier in hexadecimal format or decimal format.
-------------	--

## Element Information

<b>Parent elements</b>	<b>dpinst</b>
<b>Child elements</b>	<b>dpinstTitle</b> (zero or one) <b>eula</b> (zero or one) <b>eulaHeaderTitle</b> (zero or one) <b>eulaNoButton</b> (zero or one) <b>eulaYesButton</b> (zero or one) <b>finishText</b> (zero or one) <b>finishTitle</b> (zero or one) <b>headerPath</b> (zero or one) <b>icon</b> (zero or one) <b>installHeaderTitle</b> (zero or one) <b>watermarkPath</b> (zero or one) <b>welcomeIntro</b> (zero or one) <b>welcomeTitle</b> (zero or one)
<b>Data contents</b>	None permitted
<b>Duplicate child elements</b>	None permitted

## Remarks

You can use a **language** element to localize and customize text, the icon, and bitmaps that DPInst displays on its wizard pages. The icon represents DPInst on the Microsoft Windows taskbar, and Windows desktop.

DPInst also uses this icon for the entries that are added to **Programs and Features** in Control Panel. These

entries represent the [driver packages](#) that DPInst installs.

**Note** In versions of Windows earlier than Windows Vista, DPInst added these entries to **Add or Remove Programs** in Control Panel.

To customize the items that appear on the wizard pages in the English-only version of DPInst, use a **language** element that specifies the English (Standard) language and include child elements of the **language** element that customize the items. To localize and customize the items that appear on the DPInst wizard pages in the multi-language version of DPInst, use a **language** element that specifies the language and include child elements of the **language** element that customize the items.

The following code example demonstrates a **language** element that specifies the English (Standard) language and includes customized **dpinstTitle** and **welcomeTitle** XML child elements. The text that specifies the custom text is shown in bold font type.

```
<dpinst>
...
<language code="0x0409">
...
<dpinstTitle>Toaster Device Installer</dpinstTitle>
<welcomeTitle>Welcome to the toaster Installer!</welcomeTitle>
...
</language>
...
</dpinst>
```

If a **dpinstTitle** element is not specified, DPInst displays the default title bar text that appears on the default welcome page.

## See also

[dpinstTitle](#)

[eula](#)

[eulaHeaderTitle](#)

[eulaNoButton](#)

[eulaYesButton](#)

[finishText](#)

[finishTitle](#)

[headerPath](#)

[icon](#)

[installHeaderTitle](#)

[watermarkPath](#)

[welcomeIntro](#)

[welcomeTitle](#)

# legacyMode XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **legacyMode** XML element is an empty element that sets the **legacyMode** flag to ON, which configures DPInst to install unsigned drivers and [driver packages](#) that have missing files.

## Element Tag

```
<legacyMode>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, DPInst installs only signed [driver packages](#) and driver packages that do not have missing files. To configure DPInst to accept unsigned driver packages or driver packages that have missing files, set the **legacyMode** flag to ON by including an **legacyMode** element as a child element of a **dpinst** XML element or by using the **/Im** command-line switch.

The following code example demonstrates a **legacyMode** element.

```
<dpinst>
  ...
  <legacyMode/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# MediumChangerClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

MediumChangerClassGuid is an obsolete identifier for the device interface class for medium changer devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_MEDIUMCHANGER](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_MEDIUMCHANGER instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_MEDIUMCHANGER](#)

# MOUNTDEV\_MOUNTED\_DEVICE\_GUID

11/2/2020 • 2 minutes to read • [Edit Online](#)

The MOUNTDEV\_MOUNTED\_DEVICE\_GUID [device interface class](#) is defined for volume devices.

ATTRIBUTE	SETTING
Identifier	MOUNTDEV_MOUNTED_DEVICE_GUID
Class GUID	{53F5630D-B6BF-11D0-94F2-00A0C91EFB8B}

## Remarks

The MOUNTDEV\_MOUNTED\_DEVICE\_GUID identifier for this device interface class is an alias for the [GUID\\_DEVINTERFACE\\_VOLUME](#) device interface class.

The storage [samples](#) in the WDK includes the [ClassPnP Storage Class Driver Library](#) that uses MOUNTDEV\_MOUNTED\_DEVICE\_GUID to register instances of the GUID\_DEVINTERFACE\_VOLUME device interface class.

## Requirements

Header	Mountmgr.h (include Mountmgr.h)
--------	---------------------------------

## See also

[GUID\\_DEVINTERFACE\\_VOLUME](#)

# package XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **package** XML element specifies an INF file for a [driver package](#).

## Element Tag

```
<package>
```

## XML Attributes

<b>path</b>	The path to an INF file for a driver package. The path is relative to the DPInst working directory.
-------------	---

## Element Information

<b>Parent elements</b>	<a href="#">group</a>
<b>Child elements</b>	None permitted
<b>Data contents</b>	None permitted
<b>Duplicate child elements</b>	None permitted

## Remarks

The following code example demonstrates a **package** element that specifies DirAbc\Abc.inf as the INF file for the [driver package](#).

```
<dpinst>
  ...
  <group>
    <package path="DirAbc\Abc.inf" />
  </group>
  ...
</dpinst>
```

## See also

[group](#)

# PartitionClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

PartitionClassGuid is an obsolete identifier for the [device interface class](#) for partition devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_PARTITION](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_PARTITION instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_PARTITION](#)

# promptIfDriverIsNotBetter XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **promptIfDriverIsNotBetter** XML element is an empty element that sets the **promptIfDriverIsNotBetter** flag to ON, which configures DPInst to display a dialog box if a new driver is not a better match to a device than a driver that is currently installed on the device. The dialog box informs a user of this situation and provides an option to replace the driver that is currently installed on the device with the new driver.

## Element Tag

```
<promptIfDriverIsNotBetter>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **promptIfDriverIsNotBetter** flag is set to OFF. You can set the **promptIfDriverIsNotBetter** flag to ON by including a **promptIfDriverIsNotBetter** XML element in a *DPInst.xml* file or by using the /p command-line switch.

The following code example demonstrates a **promptIfDriverIsNotBetter** element.

```
<dpinst>
  ...
  <promptIfDriverIsNotBetter/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# quietInstall XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **quietInstall** XML element is an empty element that sets the **quietInstall** flag to ON, which configures DPInst to suppress the display of wizard pages and most other user messages.

## Element Tag

```
<quietInstall>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **quietInstall** flag is set to OFF. You can set the **quietInstall** flag to ON by including a **quietInstall** element in a DPInst descriptor file or by using the **/q** command-line switch. The **quietInstall** flag works with the presence of a EULA page and the **suppressEulaPage** flag.

The following code example demonstrates a **quietInstall** element

```
<dpinst>
  ...
  <quietInstall/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# scanHardware XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **scanHardware** XML element is an empty element that sets the **scanHardware** flag to ON. Setting this flag to ON configures DPInst to install a [driver package](#) for a Plug and Play (PnP) function driver only if the driver package matches a device that is configured in a computer and the driver package is a better match for the device than the driver package that is currently installed on the device.

## Element Tag

```
<scanHardware>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **scanHardware** flag is set to OFF. You can set the **scanHardware** flag to ON by including an **scanHardware** XML element as a child element of a **dpinst** XML element in a DPInst descriptor file or by using the /sh command-line switch.

The following code example demonstrates a **scanHardware** element.

```
<dpinst>
  ...
  <scanHardware/>
  ...
</dpinst>
```

# search XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **search** XML element directs DPInst to search recursively for INF files in specified subdirectories under the DPInst working directory. Subdirectories are specified by one or more [subDirectory child elements](#).

## Element Tag

```
<search>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a>
Child elements	<a href="#">subDirectory</a> (zero or more)
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **search** element that contains one **subDirectory** XML element that specifies the *i386* subdirectory. DPInst will recursively search for [driver packages](#) in the *i386* subdirectory of the DPInst working directory. The text that specifies the custom subdirectory is shown in bold font style.

```
<dpinst>
  ...
  <search>
    <subDirectory>i386</subDirectory>
  </search>
  ...
</dpinst>
```

**Note** Because duplicate child elements are not permitted, each **subDirectory** child element of a **search** element must be unique.

## See also

[dpinst](#)

[subDirectory](#)

# SetupDiGetWizardPage

12/5/2018 • 2 minutes to read • [Edit Online](#)

The **SetupDiGetWizardPage** function is reserved for system use. For information about wizard pages, see the DIF\_NEWDEVICEWIZARD\_XXX requests, for example, [DIF\\_NEWDEVICEWIZARD\\_FINISHINSTALL](#).

```
HPROPSHEETPAGE  
SetupDiGetWizardPage(  
    IN HDEVINFO DeviceInfoSet,  
    IN PSP_DEVINFO_DATA DeviceInfoData..OPTIONAL,  
    IN PSP_INSTALLWIZARD_DATA InstallWizardData,  
    IN DWORD PageType,  
    IN DWORD Flags  
);
```

# SetupDiMoveDuplicateDevice

12/5/2018 • 2 minutes to read • [Edit Online](#)

This function is obsolete and no longer supported in any version of the Microsoft Windows operating system.

# SP\_ENABLECLASS\_PARAMS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This structure is obsolete.

# SP\_INSTALLWIZARD\_DATA

12/5/2018 • 2 minutes to read • [Edit Online](#)

This structure is obsolete.

SP\_INSTALLWIZARD\_DATA is a parameter to the **SetupDiGetWizardPage** function.

Instead of DIF\_INSTALLWIZARD, Windows uses the DIF\_NEWDEVICEWIZARD\_XXX requests such as DIF\_NEWDEVICEWIZARD\_FINISHINSTALL.

# SP\_MOVEDEV\_PARAMS

12/5/2018 • 2 minutes to read • [Edit Online](#)

This structure is obsolete.

This structure is associated with the [DIF\\_MOVEDEVICE](#) request, which is obsolete and cannot be used with any version of Microsoft Windows.

# StoragePortClassGuid

12/5/2018 • 2 minutes to read • [Edit Online](#)

StoragePortClassGuid is an obsolete identifier for the device interface class for storage port. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_STORAGEPORT](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_STORAGEPORT instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_STORAGEPORT](#)

# subDirectory XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **subDirectory** XML element specifies one or all the subdirectories under the DPInst working directory.

## Element Tag

```
<subDirectory>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">search</a>
Child elements	None permitted
Data contents	Specifies one or all of the subdirectories that are relative to the DPInst working directory. This element can contain: <ul style="list-style-type: none"><li>• A string to specify a specific subdirectory</li><li>• The wildcard character (*) to specify all of the subdirectories</li></ul>
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **search** element that contains one **subDirectory** element that specifies an *i386* subdirectory under the DPInst working directory. The text that specifies the custom subdirectory is shown in bold font style.

```
<dpinst>
  ...
  <search>
    <subDirectory>i386</subDirectory>
  </search>
  ...
</dpinst>
```

The following code example demonstrates a **search** element that contains a **subDirectory** element that specifies all of the subdirectories under the DPInst working directory. The wildcard character (\*) that specifies all of the subdirectories is shown in bold font style.

```
<search>
  <subDirectory>*</subDirectory>
</search>
```

## See also

[search](#)

# suppressAddRemovePrograms XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **suppressAddRemovePrograms** XML element is an empty element that sets the **suppressAddRemovePrograms** flag to ON, which configures DPInst to suppress the addition of entries to **Programs and Features** in Control Panel. These entries represent the [driver packages](#) and driver package groups that DPInst installs.

**Note** In versions of Windows earlier than Windows Vista, DPInst added the entry for the driver package to **Add or Remove Programs** in Control Panel.

## Element Tag

```
<suppressAddRemovePrograms>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">dpinst</a> or <a href="#">group</a>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **suppressAddRemovePrograms** flag is set to OFF. To set the **suppressAddRemovePrograms** flag to ON for all of the drivers that DPInst installs, including all of the drivers in [driver package](#) groups, include a **suppressAddRemovePrograms** element as a child element of a **dpinst** XML element in a DPInst descriptor file, or use the **/sa** command-line switch. To set the **suppressAddRemovePrograms** flag only for a specific driver package group, include a **suppressAddRemovePrograms** element as a child element of the corresponding **group** XML element in a DPInst descriptor file.

The following code example demonstrates a **suppressAddRemovePrograms** element that is child element of a **dpinst** element.

```
<dpinst>
  ...
  <suppressAddRemovePrograms/>
  ...
</dpinst>
```

The following code example demonstrates a **suppressAddRemovePrograms** element that is child element of a

**group** element.

```
<dpinst>
  ...
  <group>
    <package path="DirAbc\Abc.inf" />
    <package path="DirDef\Def.inf" />
    <suppressAddRemovePrograms/>
  <group/>
  ...
</dpinst>
```

## See also

[dpinst](#)

[group](#)

# suppressWizard XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **suppressWizard** XML element is an empty element that sets the **suppressWizard** flag to ON, which configures DPInst to suppress the display of wizard pages and other user messages that DPInst generates.

## Element Tag

```
<suppressWizard>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b>
Child elements	None permitted
Data contents	None permitted
Duplicate child elements	None permitted

## Remarks

By default, the **suppressWizard** flag is set to OFF. You can set the **suppressWizard** flag to ON by including a **suppressWizard** XML element as a child element of a **dpinst** XML element in a DPInst descriptor file or by using the **/sw** command-line switch. The **suppressWizard** flag works with the **suppressEulaPage** flag.

The following code example demonstrates a **suppressWizard** element.

```
<dpinst>
  ...
  <suppressWizard/>
  ...
</dpinst>
```

## See also

[dpinst](#)

# System-Defined Device Setup Classes Available to Vendors

12/1/2020 • 8 minutes to read • [Edit Online](#)

If you're writing a Windows device driver for a specific category of device, you can use the following list to select the right pre-defined values to use for the `Class` and `ClassGuid` entries in the [Version Section](#) of the driver's INF file.

## NOTE

If you're looking for info on reserved classes and GUIDs, see [System-Defined Device Setup Classes Reserved for System Use](#).

To see how these entries appear in an INF file, check out `cdrom.inf` in the [Windows driver samples](#) repo.

Values in the list can be used to install device drivers on Windows 2000 and later, unless specially noted.

## NOTE

If you're looking for info on troubleshooting a problem with a CD or DVD drive, see [The CD drive or the DVD drive does not work as expected](#).

## Device categories and class values

### Battery Devices

Class = Battery

ClassGuid = {72631e54-78a4-11d0-bcf7-00aa00b7b32a}

This class includes battery devices and UPS devices.

### Biometric Device

Class = Biometric

ClassGuid = {53D29EF7-377C-4D14-864B-EB3A85769359}

(Windows Server 2003 and later versions of Windows) This class includes all biometric-based personal identification devices.

### Bluetooth Devices

Class = Bluetooth

ClassGuid = {e0cbf06c-cd8b-4647-bb8a-263b43f0f974}

(Windows XP SP1 and later versions of Windows) This class includes all Bluetooth devices.

### Camera Device

Class = Camera

ClassGuid = {ca3e7ab9-b4c3-4ae6-8251-579ef933890f}

(Windows 10 version 1709 and later versions of Windows) This class includes universal camera drivers.

### CD-ROM Drives

Class = CDROM

ClassGuid = {4d36e965-e325-11ce-bfc1-08002be10318}

This class includes 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 hard disk drives. See also the HDC and SCSIAdapter classes.

## Display Adapters

Class = Display

ClassGuid = {4d36e968-e325-11ce-bfc1-08002be10318}

This class includes video adapters. Drivers for this class include display drivers and video miniport drivers.

## Extension INF

Class = Extension

ClassGuid = {e2f84ce7-8efa-411c-aa69-97454ca4cb57}

(Windows 10 and later versions of Windows) This class includes all devices requiring customizations. For more details, see [Using an Extension INF File](#).

## Floppy Disk Controllers

Class = FDC

ClassGuid = {4d36e969-e325-11ce-bfc1-08002be10318}

This class includes floppy disk drive controllers.

## Floppy Disk Drives

Class = FloppyDisk

ClassGuid = {4d36e980-e325-11ce-bfc1-08002be10318}

This class includes floppy disk drives.

## Hard Disk Controllers

Class = HDC

ClassGuid = {4d36e96a-e325-11ce-bfc1-08002be10318}

This class includes hard disk controllers, including ATA/ATAPI controllers but not SCSI and RAID disk controllers.

## Human Interface Devices (HID)

Class = HIDClass

ClassGuid = {745a17a0-74d3-11d0-b6fe-00a0c90f57da}

This class includes interactive input devices that are operated by the system-supplied [HID class driver](#). This includes USB devices that comply with the [USB HID Standard](#) and non-USB devices that use a HID minidriver. For more information, see [HIDClass Device Setup Class](#). (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 devices that control the operation of multifunction IEEE 1284.4 peripheral devices.

## IEEE 1284.4 Print Functions

Class = Dot4Print

ClassGuid = {49ce6ac8-6f86-11d2-b1e5-0080c72e74a2}

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 Devices That Support the 61883 Protocol

Class = 61883

ClassGuid = {7ebefbc0-3200-11d2-b4c2-00a0C9697d07}

This class includes IEEE 1394 devices that support the IEC-61883 protocol device class.

The 61883 component includes the *61883.sys* protocol driver that transmits various audio and video data streams over the 1394 bus. These currently include standard/high/low quality DV, MPEG2, DSS, and Audio. These data

streams are defined by the IEC-61883 specifications.

### **IEEE 1394 Devices That Support the AVC Protocol**

Class = AVC

ClassGuid = {c06ff265-ae09-48f0-812c-16753d7cba83}

This class includes IEEE 1394 devices that support the AVC protocol device class.

### **IEEE 1394 Devices That Support the SBP2 Protocol**

Class = SBP2

ClassGuid = {d48179be-ec20-11d1-b6b8-00c04fa372a7}

This class includes IEEE 1394 devices that support the SBP2 protocol device class.

### **IEEE 1394 Host Bus Controller**

Class = 1394

ClassGuid = {6bdd1fc1-810f-11d0-bec7-08002be2092f}

This class includes 1394 host controllers connected on a PCI bus, but not 1394 peripherals. Drivers for this class are system-supplied.

### **Imaging Device**

Class = Image

ClassGuid = {6bdd1fc6-810f-11d0-bec7-08002be2092f}

This class includes still-image capture devices, digital cameras, and scanners.

### **IrDA Devices**

Class = Infrared

ClassGuid = {6bdd1fc5-810f-11d0-bec7-08002be2092f}

This class includes infrared devices. Drivers for this class include Serial-IR and Fast-IR NDIS miniports, but see also the Network Adapter class for other NDIS network adapter miniports.

### **Keyboard**

Class = Keyboard

ClassGuid = {4d36e96b-e325-11ce-bfc1-08002be10318}

This class includes all keyboards. That is, it must also 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 SCSI media changer devices.

### **Memory Technology Driver**

Class = MTD

ClassGuid = {4d36e970-e325-11ce-bfc1-08002be10318}

This class includes memory devices, such as flash memory cards.

### **Modem**

Class = Modem

ClassGuid = {4d36e96d-e325-11ce-bfc1-08002be10318}

This class includes [modem devices](#). An INF file for a device of this class specifies the features and configuration of the device and stores this information in the registry. An INF file for a device of this class can also be used to install device drivers for a *controllerless modem* or a *software modem*. These devices split the functionality between the modem device and the device driver. For more information about modem INF files and Microsoft Windows Driver Model (WDM) modem devices, see [Overview of Modem INF Files](#) and [Adding WDM Modem Support](#).

### **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 instead 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 must also 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) multiport serial controllers or single-port serial controllers (see the Ports class).

## Network Adapter

Class = Net

ClassGuid = {4d36e972-e325-11ce-bfc1-08002be10318}

This class consists of network adapter drivers. These drivers must either call [NdisMRegisterMiniportDriver](#) or [NetAdapterCreate](#). Drivers that do not use NDIS or NetAdapter should use a different setup class.

## Network Client

Class = NetClient

ClassGuid = {4d36e973-e325-11ce-bfc1-08002be10318}

This class includes network and/or print providers.

**Note** NetClient components are deprecated in Windows 8.1, Windows Server 2012 R2, and later.

## 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, in addition to higher level drivers in transport stacks.

## PCI SSL Accelerator

Class = SecurityAccelerator

ClassGuid = {268c95a1-edfe-11d3-95c3-0010dc4050a5}

This class includes devices that accelerate secure socket layer (SSL) cryptographic processing.

### **PCMCIA Adapters**

Class = PCMCIA

ClassGuid = {4d36e977-e325-11ce-bfc1-08002be10318}

This class includes PCMCIA and CardBus host controllers, but not PCMCIA or CardBus peripherals. Drivers for this class are system-supplied.

### **Ports (COM & LPT ports)**

Class = Ports

ClassGuid = {4d36e978-e325-11ce-bfc1-08002be10318}

This class includes serial and parallel port devices. See also the MultiportSerial class.

### **Printers**

Class = Printer

ClassGuid = {4d36e979-e325-11ce-bfc1-08002be10318}

This class includes printers.

### **Printers, Bus-specific class drivers**

Class = PNPPrinters

ClassGuid = {4658ee7e-f050-11d1-b6bd-00c04fa372a7}

This class includes SCSI/1394-enumerated printers. Drivers for this class 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 HBAs (Host Bus Adapters) and disk-array controllers.

### **Security Devices** Class = Securitydevices

ClassGuid = {d94ee5d8-d189-4994-83d2-f68d7d41b0e6}

(Windows 8.1, Windows 10) This class includes [Trusted Platform Module](#) chips. A TPM is a secure crypto-processor that helps you with actions such as generating, storing, and limiting the use of cryptographic keys. Any new manufactured device must implement and enable TPM 2.0 by default. For more information, see [TPM Recommendations](#).

### **Sensors**

Class = Sensor

ClassGuid = {5175d334-c371-4806-b3ba-71fd53c9258d}

(Windows 7 and later versions of Windows) This class includes sensor and location devices, such as GPS devices.

### **Smart Card Readers**

Class = SmartCardReader

ClassGuid = {50dd5230-ba8a-11d1-bf5d-0000f805f530}

This class includes smart card readers.

### **Software Component**

Class = SoftwareComponent

ClassGuid = {5c4c3332-344d-483c-8739-259e934c9cc8}

(Windows 10 version 1703 and later versions of Windows) This class includes virtual child device to encapsulate software components. For more details, see [Adding Software Components with an INF file](#).

## **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 buses, system bridges, the system ACPI driver, and the system volume manager driver.

## **Tape Drives**

Class = TapeDrive

ClassGuid = {6d807884-7d21-11cf-801c-08002be10318}

This class includes tape drives, including all tape miniclass drivers.

## **USB Device**

Class = USBDevice

ClassGuid = {88BAE032-5A81-49f0-BC3D-A4FF138216D6}

USBDevice includes all USB devices that do not belong to another class. This class is not used for USB host controllers and hubs.

## **Windows CE USB ActiveSync Devices**

Class = WCEUSBS

ClassGuid = {25dbce51-6c8f-4a72-8a6d-b54c2b4fc835}

This class includes Windows CE ActiveSync devices.

The WCEUSBS setup class supports communication between a personal computer and a device that is compatible with the Windows CE ActiveSync driver (generally, PocketPC devices) over USB.

## **Windows Portable Devices (WPD)**

Class = WPD

ClassGuid = {eec5ad98-8080-425f-922a-dabf3de3f69a}

(Windows Vista and later versions of Windows) This class includes WPD devices.

# System-Defined Device Setup Classes Reserved for System Use

12/5/2018 • 2 minutes to read • [Edit Online](#)

The following classes and GUIDs should not be used to install devices (or drivers) on Windows 2000 or later versions of Windows:

## **Adapter**

Class = Adapter

ClassGuid = {4d36e964-e325-11ce-bfc1-08002be10318}

This class is obsolete.

## **APM**

Class = APMSupport

ClassGuid = {d45b1c18-c8fa-11d1-9f77-0000f805f530}

This class is reserved for system use.

## **Computer**

Class = Computer

ClassGuid = {4d36e966-e325-11ce-bfc1-08002be10318}

This class is reserved for system use.

## **Decoders**

Class = Decoder

ClassGuid = {6bdd1fc2-810f-11d0-bec7-08002be2092f}

This class is reserved for future use.

## **Host-side IEEE 1394 Kernel Debugger Support**

Class = 1394Debug

ClassGuid = {66f250d6-7801-4a64-b139-eea80a450b24}

This class is reserved for system use.

## **IEEE 1394 IP Network Enumerator**

Class = Enum1394

ClassGuid = {c459df55-db08-11d1-b009-00a0c9081ff6}

This class is reserved for system 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 are unsure in which class your device belongs. Either

determine the correct device setup class or create a new class.

### **Printer Upgrade**

Class = PrinterUpgrade

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.

### **Storage Volume Snapshots**

Class = VolumeSnapshot

ClassGuid = {533c5b84-ec70-11d2-9505-00c04F79deaf}

This class is reserved for system use.

### **USB Bus Devices (hubs and host controllers)**

Class = USB

ClassGuid = {36fc9e60-c465-11cf-8056-444553540000}

This class includes USB host controllers and USB hubs, but not USB peripherals. Drivers for this class are system-supplied.

# TapeClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

TapeClassGuid is an obsolete identifier for the device interface class for tape storage devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_TAPE](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_TAPE instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_TAPE](#)

# VolumeClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

VolumeClassGuid is an obsolete identifier for the [device interface class](#) for volume devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_VOLUME](#) class identifier for new instances of this class.

## Remarks

The storage [samples](#) in the WDK include an [Addfilter Storage Filter Tool](#) that uses VolumeClassGuid to enumerate instances of the [GUID\\_DEVINTERFACE\\_VOLUME](#) device interface class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use <a href="#">GUID_DEVINTERFACE_VOLUME</a> instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_VOLUME](#)

# watermarkPath XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **watermarkPath** element specifies the source file for a custom watermark bitmap that DPInst displays on the left side of the DPInst welcome page and the DPInst finish page.

## Element Tag

```
<watermarkPath>
```

## XML Attributes

None

## Element Information

Parent elements	<b>dpinst</b> or <b>language</b>
Child elements	None permitted
Data contents	String that specifies the name of the file that contains a watermark bitmap that DPInst displays on the left side of a welcome page and a finish page. The watermark file must be located in the DPInst root directory, which is the directory that contains the DPInst executable file ( <i>DPInst.exe</i> ), or in a subdirectory under the DPInst root directory. If the watermark bitmap file is in a subdirectory, specify a fully qualified file name that is relative to the DPInst root directory.
Duplicate child elements	None permitted

## Remarks

A **watermarkPath** element is customized, but not localized, if it is a child element of a **dpinst** element. A **watermarkPath** element is customized and localized if it is a child element of a **language** element.

The following code example demonstrates a **watermarkPath** element that specifies *Data\Watermark.bmp* as the source of the watermark bitmap that DPInst displays on the left side of the welcome and finish pages. The text that specifies the custom watermark bitmap file is shown in bold font style.

```
<dpinst>
  ...
  <watermarkPath>Data\Watermark.bmp</watermarkPath>
  ...
</dpinst>
```

If a **watermarkPath** element is not specified, DPInst uses a default watermark.

See also

[dpinst](#)

[language](#)

# welcomeIntro XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **welcomeIntro** XML element customizes the main text on the DPInst welcome page.

## Element Tag

```
<welcomeIntro>
```

## XML Attributes

None

## Element Information

Parent elements	<a href="#">language</a>
Child elements	None permitted
Data contents	String that customizes the main text on a welcome page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **welcomeIntro** element that customizes the main text on a welcome page. The text that specifies the custom welcome introduction is shown in bold font style.

```
<dpinst>
  ...
  <language code="0x0409">
    ...
    <welcomeIntro>This wizard will walk you through updating the drivers for your Toaster device.
  </welcomeIntro>
  ...
  </language>
  ...
</dpinst>
```

If a **welcomeIntro** element is not specified, DPInst displays the default main text that is shown on the default welcome page.

## See also

[language](#)

# welcomeTitle XML Element

11/2/2020 • 2 minutes to read • [Edit Online](#)

[DIFx is deprecated, for more info, see [DIFx Guidelines](#).]

The **welcomeTitle** XML element customizes the bold text of the welcome title that appears at the top of the DPInst welcome page.

## Element Tag

```
<welcomeTitle>
```

## XML Attributes

None

## Element Information

Parent elements	<b>language</b>
Child elements	None permitted
Data contents	String that customizes the title text at the top of a welcome page
Duplicate child elements	None permitted

## Remarks

The following code example demonstrates a **welcomeTitle** element customizes the title text on a welcome page. The text that specifies the custom welcome title is shown in bold font style.

```
<dpinst>
...
<language code="0x0409">
...
<welcomeTitle>Welcome to the toaster Installer!</welcomeTitle>
...
</language>
...
</dpinst>
```

If a **welcomeTitle** element is not specified, DPInst displays the default welcome title that is shown on the default welcome page.

## See also

[language](#)

# WriteOnceDiskClassGuid

11/2/2020 • 2 minutes to read • [Edit Online](#)

WriteOnceDiskClassGuid is an obsolete identifier for the device interface class for write-once disk devices. Starting with Microsoft Windows 2000, use the [GUID\\_DEVINTERFACE\\_WRITEONCEDISK](#) class identifier for new instances of this class.

## Requirements

Version	Obsolete. Starting with Windows 2000, use GUID_DEVINTERFACE_WRITEONCEDISK instead.
Header	Ntddstor.h (include Ntddstor.h)

## See also

[GUID\\_DEVINTERFACE\\_WRITEONCEDISK](#)