Getting Started with Windows Drivers: Windows DDK

Overview of Windows Driver Development

This document guides you through the planning and decision-making process involved in developing a Windows device driver, starting with design and continuing through distribution.

Given the wide variation among devices and the intricate nature of driver operations, no single driver design or development plan can cover every possible technology. However, any driver development project will proceed more smoothly and efficiently if you:

- Follow industry standards for hardware and drivers
- Understand important operating system and driver concepts
- Decide early in the design process which operating system platforms to support
- Test for device and driver compatibility with all Windows operating systems you plan to support
- Use up-to-date driver development, debugging, and testing tools

This guide is intended for programmers who are new to Windows driver development. It assumes that you know C and might have written a driver for other operating systems (including MS-DOS®, Windows® 95, or Windows NT® 4.0 and earlier), but have not written a driver for Windows 2000 or later, or for Windows 98/Me.

The following sections provide further introductory material:

Components of a Driver Package

Steps in Driver Development

Additional Resources for Driver Writers

Because the purpose of most drivers is to operate a physical device, this guide is aimed at those writing such drivers. If you're writing a file system driver or file system filter, see the *Installable File System Kit* (IFS Kit). You can order the IFS Kit through the Microsoft web site at www.microsoft.com/ddk/ifskit.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Components of a Driver Package

Supporting a device on a Windows operating system typically involves the following components:

- The device
- Driver files

- Installation files
- Other files

A brief description of each component follows.

The Device

If you're involved in designing and building a new device, follow industry hardware standards. Building devices that conform to industry standards can streamline your driver development process as well as reduce support costs. Not only do test suites exist for such devices, but in many cases generic drivers exist for standard types, so you might not need to write a new driver.

See the Microsoft Hardware Development web site at www.microsoft.com/hwdev for information about industry standards and specifications.

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 with the *.sys* filename extension. When a device is installed, Setup copies the *.sys* file to the *%windir%\system32\drivers* directory.

The software 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 need 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, which provide driver installation:

- A device setup information file (INF file)

  An INF file contains information that the system Setup components use to install support for the device. Setup copies this file to the %*windir*%*inf* directory when it installs the 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 digital signatures. All driver packages should be signed.

  To get a driver package digitally signed, you must submit the package to the Windows Hardware Quality Lab (WHQL) for testing and signing. WHQL returns the package with a catalog file. See Driver Signing for details.

- One or more optional co-installers

A co-installer is a Win32® DLL that assists in device installation NT-based operating systems. For example, an IHV might provide a co-installer to provide Finish Install wizard pages or to copy additional INF files. Co-installers are optional. See Supplying a Co-Installer for details.

Co-installers are not supported on Windows 98 or Me.

Other Files

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

Supplying a Device Installation Application

Providing Device Property Pages

Installing a Boot Driver

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Steps in Driver Development

This guide organizes driver development into six steps, listed below. Each step corresponds to a section in the DDK.

- Step 1: Understand Driver and Operating System Basics

  Before starting design, you should understand the fundamentals of how drivers work in Windows operating systems. Knowing the fundamentals will help you make appropriate design decisions and streamline your development process.

- Step 2: Determine Device-Specific Driver Requirements

  Each Windows device class has specific hardware and software requirements. Devices and drivers should conform to these requirements to operate correctly with other system components.

  Understanding and following these requirements during hardware and software design can save you time and money.

- Step 3: Make Driver Design Decisions

  Before you write any code, you need to make some fundamental decisions about driver design. For example, you should decide which Windows operating systems and hardware platforms your driver will run on, and whether you can modify an existing sample or legacy driver or instead should start from scratch.

- Step 4: Build, Test, and Debug the Driver

  Building a driver is not the same as building a user-mode application. This section gives tips on how to use the "free" and "checked" operating system builds and how to configure the build environment for a driver.

  Iterative testing and debugging on as many hardware configurations as possible helps to ensure a working driver.

- Step 5: Provide an Installation Package

  Exactly how a driver must be installed depends on the type of driver and device. This section helps you determine which device installation components you must provide and outlines what Windows expects during device installation.

- Step 6: Distribute the Driver

  The final step in driver development is to distribute the driver. If your driver meets the quality standards defined for the Microsoft Windows Logo program, you can distribute it through Microsoft's Windows Update program.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Additional Resources for Driver Writers

In addition to the Windows DDK, Microsoft provides several other resources for driver writers:

- The DDK web site, www.microsoft.com/ddk
- The hardware development web site, www.microsoft.com/hwdev
- The hardware test web site, www.microsoft.com/hwtest
- The Microsoft Developer Network (MSDN®) web site, at msdn.microsoft.com. The Platform SDK, which describes the Win32® API, is available from MSDN.
- The IFS Kit web site, www.microsoft.com/ddk/ifskit
- The Microsoft Hardware Newsletter, available under nondisclosure agreement to qualified vendors

Books on driver development and Windows internals are available from Microsoft Press and other publishers. Such books include the following:

- *Inside Windows 2000*, Third Edition (previously titled *Inside Windows NT*), by David Solomon, published by Microsoft Press.
- *Programming the Microsoft Windows Driver Model*, by Walter Oney, published by Microsoft Press.

Because driver structure and requirements changed significantly with the releases of Windows 98 and Windows 2000, which introduced the Windows Driver Model (WDM), make sure that any resource you consult has been updated for these releases.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Understanding Driver and Operating System Basics

This section provides fundamental information about the following for driver writers:

Overview of System Components for Driver Writers

User-Mode Drivers and Kernel-Mode Drivers

Layered Driver Architecture

Device Drivers and File System Drivers Defined

Windows Driver Model (WDM) Defined

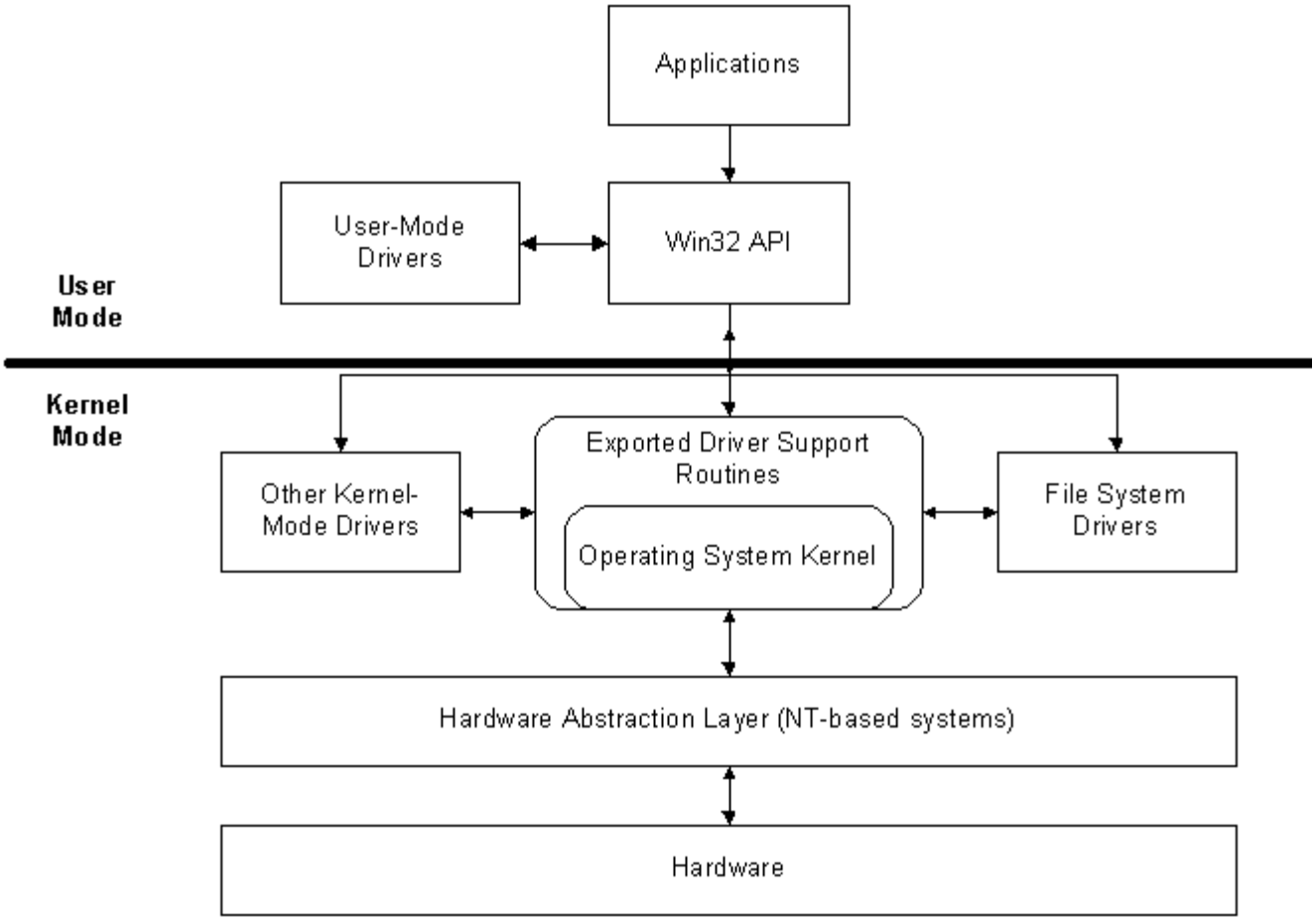
Operating System Concepts for Driver Writers

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Overview of System Components for Driver Writers

The following figure shows a broad overview of the system components that are relevant to drivers:



Windows Component Overview

As the preceding figure shows, the operating system includes kernel-mode components and user-mode components.

User-mode drivers and applications can use routines defined in the Win32® Application Programming Interface (API), which is described in the Platform SDK. The Win32 API, in turn, calls exported driver and operating system kernel routines.

Kernel-mode drivers can use support routines that are defined and exported by various components of the operating system kernel. These routines support I/O, configuration, Plug and Play, power management, memory management, and numerous other operating system features. The Overview of Windows Components describes these components in greater detail.

The NT-based operating system's kernel is designed to be portable and hardware-independent, and thus is layered on top of the hardware abstraction layer (HAL). The HAL provides hardware-dependent features. Windows 98/Me does not support this level of hardware independence.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

User-Mode Drivers and Kernel-Mode Drivers

Windows® drivers can run in either user mode or kernel mode.

- User-mode drivers run in the non-privileged processor mode in which other application code, including protected subsystem code, executes. User-mode drivers cannot gain access to system data except by calling the Win32® API which, in turn, calls system services.
- Kernel-mode drivers run as part of the operating system's executive, the underlying operating system component that supports one or more protected subsystems.

User-mode and kernel-mode drivers have different structures, different entry points, and different system interfaces. Whether a device requires a user-mode or kernel-mode driver depends on the type of device and the support already provided for it in the operating system.

Some device drivers can run wholly or partially in user mode. User-mode drivers have unlimited stack space, access to the Win32 API, and easier debugging (with user-mode debuggers).

For example, printer drivers are divided into user interface and rendering components. The user interface component runs in user mode, and calls the Win32 API to render images. The Win32 API, in turn, calls the rendering component, which can run in either kernel mode or user mode.

Most device drivers run in kernel mode. Kernel-mode drivers can perform certain protected operations and can access system structures that user-mode drivers cannot access. The increased access comes at the price, however, of more difficult debugging and a greater chance of system corruption. When code runs in the privileged kernel-mode environment, the operating system, by design, performs fewer checks on data integrity and the validity of requests.

To determine which type of driver your device requires, see System-Supplied Drivers.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Device Drivers and File System Drivers Defined

A Windows® driver can support a device or a file system:

Device driver

A device driver provides I/O services for an underlying device. For example, all of the following are considered device drivers: the IEEE 1394 bus driver, a video class driver that manages streaming input data for a variety of video devices, a video miniclass driver that communicates with the class driver to support a specific video device, and a filter driver that filters the streaming data. Some device drivers — particularly those for audio, video, and print devices — run in user

mode, but most run in kernel mode.

File system driver

A *file system driver* handles I/O independent of any underlying physical device. File system drivers include drivers for the system-supplied NTFS and file allocation table (FAT) file systems. On the NT-based operating system, file system drivers are kernel-mode drivers. *File system filter drivers* provide additional capabilities above a standard file system driver, typically by implementing such value-added services as virus screening.

Although the support is not visible to the writer of a file system driver, every file system driver ultimately depends on support from one or more underlying peripheral devices. File system drivers might also rely on support from one or more Plug and Play (PnP) hardware bus drivers.

This guide, as well as the bulk of the Windows DDK, applies to device drivers. If you are writing a file system driver or filter, see the *Installable File Systems Kit* (IFS Kit). You can order the IFS Kit through the Microsoft Web site at www.microsoft.com/ddk/ifskit.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Windows Driver Model (WDM) Defined

Any kernel-mode device driver that conforms to the Windows Driver Model (WDM) is considered a *WDM driver*. The Windows Driver Model defines a unified approach to writing kernel-mode device drivers for all Windows® operating systems.

Any device driver that will run on both Microsoft® Windows 98/Me and Windows 2000 and later operating systems must be a WDM driver. With a few special-case statements in the code, WDM drivers can be source-compatible across the Windows 98/Me and Windows 2000 and later operating systems. For more information, see Introduction to WDM.

Built on Thursday, July 19, 2001

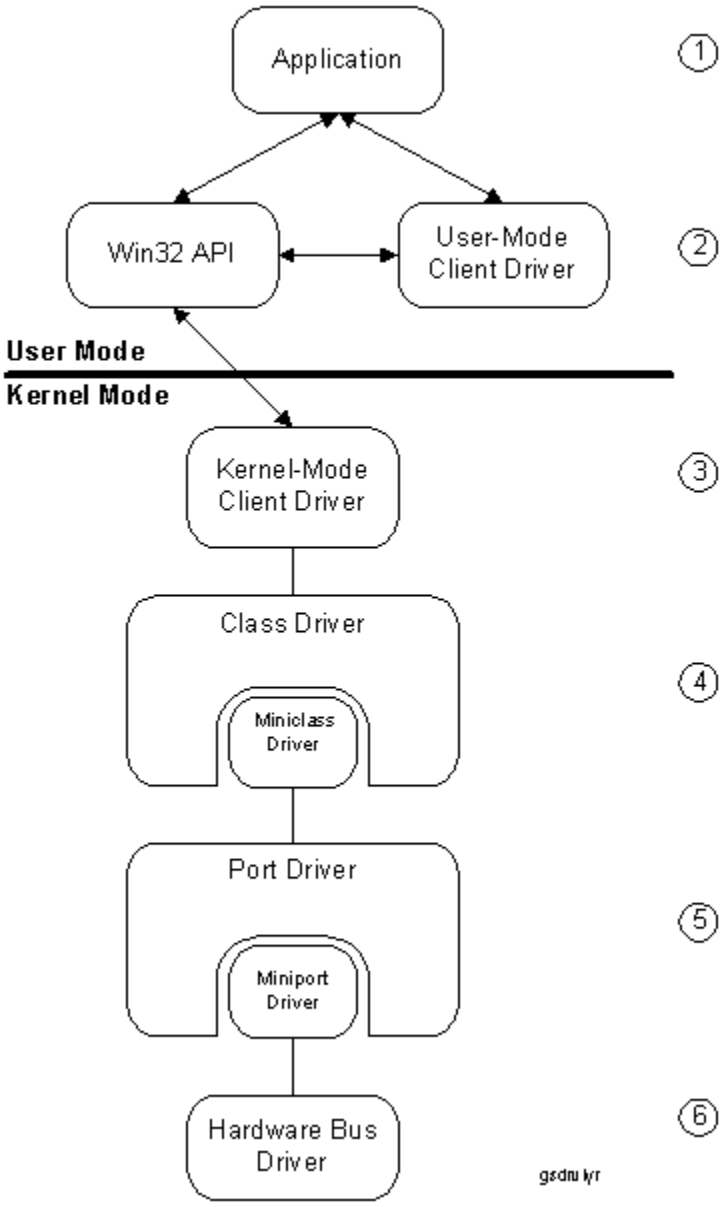
Getting Started with Windows Drivers: Windows DDK

Layered Driver Architecture

Windows® operating systems support a layered driver architecture. Every device is serviced by a chain of drivers, typically called a *driver stack*. Each driver in the stack isolates some hardware-dependent features from the drivers above it.

The following figure shows the types of drivers that could potentially be in a driver stack for a

hypothetical device. In reality, few (if any) driver stacks contain all these types of drivers.



Layered Driver Architecture

As the preceding figure shows:

1. Above the driver stack is an application. The application handles requests from users and other applications, and calls either the Win32® API or a routine that is exposed by the user-mode client driver.
2. A user-mode client driver handles requests from applications or from the Win32 API. For requests that require kernel-mode services, the user-mode client driver calls the Win32 API, which calls the appropriate kernel-mode client or support routine to carry out the request. User-mode client drivers are usually implemented as dynamic-link libraries (DLL). Printers support many operations that can be performed in user mode, and so typically have user-mode clients; disks and

other storage devices, networks, and input devices do not.

3. A kernel-mode client driver handles requests similar to those handled by the user-mode client, except that these requests are carried out in kernel mode, rather than in user mode.
4. A device class and miniclass driver pair provides the bulk of the device-specific support. The *class driver* supplies system-required but hardware-independent support for a particular class of device. Class drivers are typically supplied by Microsoft.

   A *miniclass driver* handles operations for a specific type of device of a particular class. For example, the battery class driver supports common operations for any battery, while a miniclass driver for a vendor's UPS device handles details unique to that particular device. Miniclass drivers are typically supplied by hardware vendors.

5. A corresponding *port driver* (for some devices, this is a *host controller* or *host adapter* driver) supports required I/O operations on an underlying port, hub, or other physical device through which the device attaches. Whether any such drivers are present depends on the type of device and the bus to which it eventually connects.

   All driver stacks for storage devices have a port driver. For example, the SCSI port driver provides support for I/O over the SCSI bus.

   For USB devices, a *hub* and *host controller* driver pair perform the duties of the port driver. These drivers handle I/O between the devices on the USB bus and the bus itself.

   A corresponding *miniport driver* handles device-specific operations for the port driver. For most types of devices, the port driver is supplied with the operating system, and the miniport driver is supplied by the device vendor.

6. At the bottom of the figure is the *hardware bus drive*r. Microsoft supplies bus drivers for all the major buses as part of the operating system. You should not attempt to replace these drivers.

Network drivers have their own unique terminology, defined in Windows 2000 and Later Network Architecture and the OSI Model. Nevertheless, network drivers are similarly layered, and each layer isolates device-specific or protocol-specific functionality from the layer above it.

Exactly which drivers are present, and what they are called, depends on the type of device and the bus to which it connects.

Graphics cards, for example, require a display driver, a video port driver, and a video miniport driver. The display driver is analogous to the kernel-mode client driver in the previous figure. It provides general drawing capabilities and can often work with more than one graphics card. The video port driver supports device-independent graphics operations. It works in tandem with the video miniport driver, which provides functionality that is specific to one graphics card (or a family of graphics cards). The paired video port/miniport drivers are analogous to the port/miniport drivers in the figure, and no class/miniclass drivers are present. For more information, see Display Architecture.

For simplicity, *filter drivers* are not shown in the previous figure. However, a filter driver can fit in at any layer of the driver stack above the hardware bus driver. A filter driver adds value to an existing driver by "filtering" — intercepting and manipulating device I/O. As a general rule, filter drivers do not operate the hardware directly, but work only on data and I/O requests passed to them from the next-higher or next-lower driver.

DirectShow®, the Microsoft software for video capture, includes system-supplied filter drivers that run in user mode. These filters act as clients of the kernel-mode stream class driver to expose the underlying video capture technology.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Operating System Concepts for Driver Writers

Familiarity with common operating system structures and concepts is critical to writing robust drivers that can be readily debugged, maintained, and ported.

If your device requires a user-mode driver, refer to the device-specific documentation in the Windows® DDK, and see the Platform SDK documentation for details on the Win32® API and services.

The *Kernel-Mode Driver Architecture* node in the Windows DDK provides comprehensive material on the operating system features available to kernel-mode drivers, and how to use them. If you have not previously written a driver, or have written drivers only for Windows® 95 and other single-processor operating systems, you should pay particular attention to the following topics:

Multiprocessor-Safe

Packet-Driven I/O with Reusable IRPs

Supporting Asynchronous I/O

Managing Hardware Priorities

For a list of information sources outside the Windows DDK, see Additional resources for driver writers.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Determining Device-Specific Driver Requirements

Before beginning to design or code a Windows driver, you must determine what kind of driver your device requires. In some cases, you might conclude that you do not need to write a driver after all. This section provides information on the following:

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Overview of Device Classes

Windows defines two types of device classes:

- A device setup class is a group of devices that are set up and configured in the same way.
- A device interface class describes a set of device features, regardless of the bus to which the device is connected.

Understanding the distinction between these two classes, and how your device is characterized within each class, is an essential early step in driver design. For more information, see Setup Classes versus Interface Classes.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

System-Supplied Drivers

Drivers for most standard types of devices ship with the operating system. These drivers include bus drivers for the most common buses, port and host controller drivers, class drivers, and file system drivers.

In addition, the Windows DDK provides sample drivers and documentation for a wide variety of devices, and documents interfaces for common buses so that, if necessary, you can write a driver for a device of a new type.

To determine what type of driver your device requires, and whether such a driver is already present, consult the technology-specific information contained in this DDK documentation. Use the DDK viewer's table of contents, index, and search capability to locate information about driver support for your device type. If your device is a multifunction device, you should also see Supporting Multifunction Devices.

If a class driver is available for your device type, you probably do not need to write a complete driver, but instead only a minidriver that provides device-specific support and works with the class driver.

The system-supplied class drivers are designed to hide bus-specific requirements from minidrivers. For example, the mouse class driver *mouclass.sys* interfaces with the HID class driver for USB mice, and with the I8042 port driver for PS/2 mice. As a result, supporting a new type of mouse on either of these buses is fairly straightforward. Similarly, the storage class drivers work with the storage port driver, isolating bus-related details from device-specific storage miniport drivers.

If a class driver is not available for your device type, or if the available class drivers do not support the bus or port to which your device attaches, your device probably requires a new driver. The driver should include both the "top-end" device-specific functionality and the "bottom-end" functionality to communicate with the next-lower driver (usually a bus or port driver). For suggestions on how to streamline the development process, see Driver Design and Implementation Strategies.

Do not attempt to replace the system-supplied bus drivers.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Making Driver Design Decisions

After determining what kind of driver your device requires, you can begin designing your driver. Among the design issues to consider are the following:

Choosing a Programming Language

Driver Design and Implementation Strategies

Using the DDK Samples

Upgrading a Legacy Driver

Writing Drivers for Multiple Platforms and Operating Systems

Planning for 64 Bits

Providing Driver Localization

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Choosing a Programming Language

For maximum portability, drivers should be written in a high-level language, typically C for kernel-mode drivers, and C or C++ for user-mode drivers. The following are exceptions:

- Audio/Video streaming (AVStream) client minidrivers (Microsoft® DirectX® 8.0, Microsoft® Windows® XP and later versions of the operating system) are typically written in C++.
- WDM audio miniport drivers use kernel-mode COM components and are typically written in C++.
- Windows Image Acquisition (WIA) drivers, most of which run in user mode, are typically written in C++.
- Display drivers are typically written in C. However, the system-supplied Graphics Display Interface (GDI) defines the public portions of its internal data structures as user objects. GDI also defines numerous service routines to manipulate these objects. Display drivers written in C++ can treat these routines as methods on the user object; drivers written in C can call these routines in the usual manner.

Most DDK sample and system-supplied drivers are written in C. In general, driver code should conform to the ANSI C standard, and should avoid depending on anything that the standard describes as "implementation defined." However, driver writers should take advantage of implementation-defined structured exception handling, if possible.

In particular, to write portable drivers, it is best to avoid the following:

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

To ensure compatibility across hardware platforms, drivers should not contain assembly language code.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Driver Design and Implementation Strategies

Support for a new device typically requires a device-specific driver or minidriver that interoperates with existing drivers above and below it in the driver stack.

You can add a driver to the driver stack or replace an existing driver. The location of a new (or replacement) driver in the driver stack depends partly on the hardware configuration of the individual machine, and partly on how much support the new driver can get from existing system drivers.

At minimum, a replacement driver must implement the same set of features as the driver it replaces.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Structured Exception Handling

Whether you're writing a user-mode or kernel-mode driver, you should implement structured exception handlers to improve the reliability of your code. The Visual C++® compiler provides the **try/except** mechanism, which can be used by any driver. See the Visual C++ documentation for details.

Both the Win32® API and the operating system include additional support routines for structured exception handling. For information on support routines in the Win32 API (user mode only), see the Platform SDK.

The following operating system routines (kernel mode only) support structured exception handling:

ExRaiseStatus

ExRaiseAccessViolation

ExRaiseDatatypeMisalignment

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Using the DDK Samples

The Windows® DDK includes sample source code for a variety of drivers. The samples can be useful blueprints for writing your own drivers. Samples are installed in the *\src* directory.

Each sample is distributed with a Readme file. The Readme file notes what the sample does, which operating systems it has been tested on, and how to build it. The Readme also lists the files that make up each sample.

The following table lists some of the sample directories and their contents. For more information about DDK samples, see the device type-specific documentation in this DDK.

Sample Directory	Contents
Audio	Microsoft® DirectMusic® and Audio Compression Manager (ACM) filters.
General	Sample programs to install and remove drivers; other generic driver samples.

IME	Input Method Editor (IME) DLLs and samples.
Input	Keyboard and mouse class filter drivers.
Kernel-Mode	Drivers for parallel and serial devices and specific chipsets.
Networks	Samples for various types of network drivers.
Print	DLLs, user interface tools, and sample skeleton drivers for printers and plotters.
SmartCard	Drivers for serial and PCMCIA Smartcard readers.
Storage	Storage drivers and installation applications.
VDD	Virtual device drivers for DOS compatibility.
Video	Display drivers and a mirroring driver.
WDM	WDM-compatible drivers for a variety of devices. All support Plug and Play and are source-compatible across Windows 2000 and later versions of the operating system, and Windows 98/Me. Also includes a WMI filter for any WDM driver.

If you choose to base your code on one or more of the samples, keep in mind the following:

- Some samples are not complete drivers, but instead provide guidelines for how such drivers could be written. For instance, samples might omit error-handling code in the interest of brevity and clarity.
- Make sure to use a driver sample written for the type of driver that you are writing. For example, if you are writing a WDM function driver, use a sample WDM function driver. If you are writing a storage filter driver, use a sample storage filter driver.
- Be sure to update the sample INF file for your driver. In particular, make sure that the device ID of your hardware matches the device ID specified in the INF file.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Upgrading a Legacy Driver

Although Microsoft® Windows® 2000 and later versions of the operating system continue to support some legacy drivers written for Windows NT®, new drivers should be WDM drivers. The system-supplied device drivers have been modified to support WDM, which includes Plug and Play and power management.

Drivers that support WDM, or at least Plug and Play, differ in structure from legacy drivers. For example, while legacy drivers create named device objects and symbolic links in a **DriverEntry** routine, WDM and Plug and Play function drivers create unnamed device objects in an *AddDevice* routine, and register and enable a device interface using I/O Manager routines and device installation functions. In addition, WDM and Plug and Play drivers have different requirements for resource allocation, registry usage, and device enumeration, among others.

Because of these considerable differences, in most cases it is more efficient to start with a sample WDM driver for your device type and replace device-specific routines than to revise an existing legacy driver. For examples, see the Toaster sample in *\src\general* and the device-type-specific sample WDM drivers in *\src\wdm*.

However, if you choose to try to upgrade a driver, your driver will require the following types of changes:

- Conforming to changes to the registry for devices and drivers

  These changes, made to support Plug and Play and WDM drivers, make the layout of the registry similar to that of the Windows® 95 registry tree for devices and their drivers. In fact, the boot code used in Windows 2000 and later versions of the operating system transfers all registry information from the legacy **HARDWARE\....\CurrentControlSet...** registry tree into a Windows 95-style "devnode" registry tree at installation. For more information, see Driver Information in the Registry.

- Updating or replacing the driver's installation script (an INF file)

  INF files for Windows 2000 and later versions of the operating system and for Windows 98/Me combine features from Windows NT 4.0 INF files and Windows 95 INF files to support Plug and Play and WDM drivers. For more information, see Creating an INF File.

- Supporting Plug and Play

  Windows operating systems include a set of Plug and Play/WDM hardware bus drivers for nearly every type of common I/O bus found in PCs. Drivers of devices connected on such an I/O bus should support Plug and Play.

  For some types of drivers, support for Plug and Play is provided by the operating system components that export the interfaces used by those drivers. For example, Plug and Play support is built into the system-supplied network and video port drivers. To upgrade a legacy network or display driver, ensure that it uses the system-supplied Plug and Play support.

- Supporting power management

  Because power management and Plug and Play technologies are interdependent, every legacy driver that is upgraded to support Plug and Play must also be upgraded to support power management.

- Turning a legacy driver into a WDM driver, if possible

  WDM drivers must include the master DDK header file *wdm.h* instead of *ntddk.h*, so a WDM driver can call only the exported support routines and use only the types, macros, and constants defined in *wdm.h*.

  Some types of legacy drivers, particularly drivers of devices connected on newer, dynamically configurable PC I/O buses, can be modified to become WDM drivers.

Additional device-type specific changes might also be required. For details about specific devices, see

the Design Guide for your device type.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Writing Drivers for Multiple Platforms and Operating Systems

This topic provides information useful for designing drivers to run on multiple Windows hardware platforms and operating systems. See also Planning for 64 Bits and Building Multiple Targets from One Subdirectory.

WDM Multiplatform Issues

Although Windows 2000 and later versions of the NT-based operating system, along with Windows 98/Me, all support the Windows Driver Model, the model's features have changed slightly as later operating systems have been released. Before writing a WDM driver, be sure to see WDM Versions.

For cross-system compatibility, thoroughly test drivers on all target operating systems. Operating system differences can produce slightly different driver behavior. Differences are most apparent between single processor and multiprocessor platforms.

Differences in DirectX Versions

DirectX, particularly the DirectDraw DDI, includes a number of features that operate differently on Windows 98/Me and Windows 2000 and later. For more information, see the DirectDraw DDI.

Differences Related to Physical Address Extension (PAE)

Windows 2000 and later Advanced Server and Datacenter support Physical Address Extension (PAE), which supports up to 8 gigabytes (for Advanced Server) or 32 gigabytes (for Datacenter) of physical RAM.

Special hardware compatibility (HCL) and test requirements apply to devices supported on these systems. Drivers for such devices should comply with the guidelines listed in Physical Address Extension (PAE) and in Large Memory Enabled Device Driver Hardware and Software Requirements. Note the special guidelines for NDIS and SCSI miniport drivers, also described in Physical Address Extension (PAE).

Differences in INFs

A driver can use a single INF file for installation on both Windows 2000 and later versions of the operating system and on Windows 98/Me systems, provided that any operating system-specific details are listed in the appropriate INF *DDInstall* section. For more information, see Creating INF Files for

Multiple Platforms and Operating Systems.

Windows XP and later systems also provide a way for an INF file to distinguish installations on different versions of the NT-based operating system. For more information, see the INF **Manufacturer** section.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Planning for 64 Bits

Whether you are writing a new driver or modifying an existing driver, you should consider making it compatible with 64-bit Windows. The Windows DDK includes a partial 64-bit build environment you can use to evaluate your driver source code for the upcoming 64-bit version of the Windows operating system. For details, see 64-Bit Issues.

Many of the driver samples included with the DDK are designed for 64-bit compatibility, as noted in the Readme files for the individual samples.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Providing Driver Localization

If your device/driver package will be distributed in the international marketplace, you should plan for localization of the package. The following are simple steps towards localization:

- Use UNICODE strings throughout the driver.
- Isolate text to a resource file or header file.
- Make an international INF file for device installation.
- Make sure that any user interfaces, such as device property pages, display correctly when translated into another language.

Several of the system-supplied class and port drivers, along with the WMI service, include "hooks" for locale-specific information. Check the documentation for your device type for details.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Building, Debugging, and Testing a Driver

This section provides basic information and guidance on building, debugging and testing drivers. It covers the following topics:

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Overview of Build, Debug, and Test Process

Building, testing, and debugging a driver is an iterative process that involves the following steps:

1. Writing the driver code, which should include debugging routines and macros that are flagged for conditional compilation
2. Building a checked version of the driver
3. Testing and debugging a checked version of the driver on a checked build of each target operating system
4. Testing and debugging the free version of the driver on a free build
5. Tuning the performance of the driver on the free build
6. Final testing and verification using the free build

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Using Free and Checked System Builds

To test and debug a driver, you should use both the free build and the checked build of the target operating system.

- The *free build* is the same as the build of the operating system that is sold to customers, and is thus sometimes referred to as the *retail build*. The system and drivers are built with full optimization, and debugging asserts are disabled.
- The *checked build* serves as a testing and debugging aid for both the operating system and for kernel-mode drivers. The checked build contains extra error checking, argument verification, and debugging information that is not available in the free build. You can isolate and track down driver problems much more quickly than on a free build. A checked build of the system and drivers is larger and slower, and uses more memory, than the free build.

For full details on how these builds differ and how they can be used, see The Checked Build of Windows.

In the early stages of driver development you should use the checked build to debug the driver. The additional debugging code in the checked build protects against many driver errors, such as recursive spin locks.

Performance tuning, final testing, and verification of the driver should be done on the free build. The faster speed of the free build makes it possible to detect race conditions and other synchronization problems.

Note The free and checked Windows builds are not related to the free and checked build environments. For information about the build environment windows, see Setting Up the Build Environment.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Building a Driver

The Windows DDK includes an environment for building drivers that closely matches the build environment used internally at Microsoft®. This environment includes various tools useful in developing drivers, including build utilities (*build.exe* and *nmake.exe*), a C compiler (*cl.exe*), and a linker (*link.exe*). The Build utility automatically invokes nmake, the compiler, and the linker according to the command-line options you specify.

For a description of how to use the Build utility, see Overview of the Build Utility.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Debugging a Driver

The Microsoft® Windows® DDK includes the *Debugging Tools for Windows* package. This package contains extremely powerful user-mode and kernel-mode debuggers that can debug applications, drivers, and services running on any NT-based operating system. See Debugging Tools for NT-Based Operating Systems for more information on these tools.

For debugging on Windows 98/Me, you should use the Windows System Debugger (*debugger.exe*, *wdeb386.exe*, *wdeb98.exe*). Several other tools, such as the AMLI Debugger, are also available for use on Windows 98/Me. See Windows ME Debugger and Other Tools for more information on these tools.

In addition, the Microsoft Visual Studio® debugger can be used to debug user-mode programs on all Windows operating systems. See the Visual Studio documentation for details on this debugger.

Debugging-Related Routines

There are a number of user-mode and kernel-mode routines that allow your driver to interact with a debugger. See Overview of Debugging Routines for details.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Testing a Driver

This section covers the following topics:

"Designed for Windows" Logo Testing

Tips for Testing Drivers

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

"Designed for Windows" Logo Testing

Microsoft's Windows Hardware Quality Labs (WHQL) tests drivers for compliance with requirements of the Windows Logo Program for hardware. Through this program, device manufacturers can license the Windows Logo for use on product packaging, advertising, and other marketing materials for all systems and components that pass compliance testing.

Hardware and drivers must pass WHQL testing to receive a "Designed for Windows" logo, to receive a digital signature, to be included on the Hardware Compatibility List (HCL), and to be distributed as part of the Windows Update program. See Driver Signing for additional details.

WHQL requirements vary depending on the type of device. However, if the device class is supported under the Windows Driver Model (WDM), the driver must use WDM.

For most device classes, WHQL provides a test kit that you can download. Details vary by device class; see the Microsoft Windows Hardware Quality Labs web site at www.microsoft.com/hwtest for current information, kits, and download instructions.

See also the Windows Driver and Hardware Development web site at www.microsoft.com/hwdev for additional information on device and driver testing.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Tips for Testing Drivers

The following are some suggestions for testing your driver package:

- Create the INF file early in the development process and use it throughout testing. If necessary, use GenINF to create a skeleton INF.
- Use the ChkINF tool to verify the structure and syntax of the INF file, and to assist in diagnosing INF and other installation related issues.
- Use Driver Verifier.
- Run the Hardware Compatibility Tests (HCT) and the WHQL test kit for your device type, even if you do not intend to submit your device and driver for the "Designed for Windows" Logo program. See the Microsoft Windows Hardware Quality Labs web site at www.microsoft.com/hwtest for current information, test kits, and download instructions.
- Test your driver and device on as many different hardware configurations as you possibly can. Varying the hardware can help you find conflicts between devices and other errors in device interactions.
- Test your driver and device on multiprocessor systems. Race conditions and other timing problems appear on multiprocessor systems that would not otherwise be found.
- Test your driver and device for specific system and hardware conditions, particularly edge conditions. For example, these might include "D3 hot" and "D3 cold." Make sure your driver and device can return correctly from device power state D3 "hot" (without losing power) and D3 "cold" (when power is removed from the device).

Microsoft's Hardware Test group regularly sponsors testing events ("plugfests") that are open to industry participants. System OEMs and independent hardware vendors (IHVs) who attend these plugfests can test their products with a variety of configurations. For details on upcoming plugfests, see the Hardware Test web site at www.microsoft.com/hwtest.

Special Interest Groups (SIGs) in the hardware industry may sponsor similar events for those working

with devices of specific types. See www.microsoft.com/hwdev for links to device-class-specific SIGs.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Providing an Installation Package

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

This section provides information to help you determine which components to supply, along with some guidelines for installation design. It covers the following topics:

Installation Component Overview

Supplying an INF File

Supplying a Co-Installer

Supplying a Device Installation Application

Guidelines for Device and Driver Installation

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Installation Component Overview

Installing a device or driver is substantially different from installing an application. When installing an application, a user might start the Add/Remove Programs Wizard whereas, when installing a device, he or she might start the Add Hardware Wizard. (Alternatively, the user might insert a distribution disk into a CD-ROM drive, causing an Autorun-invoked setup application to start the installation.) Installation of applications relies on the Windows Installer and corresponding **Msi** functions (described in the Platform SDK). Installation of devices and drivers relies on the system's Setup component and corresponding general Setup functions, device installation functions, and PnP Configuration Manager functions.

The Device Installation Overview 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 along with their source and destination locations
- Device-specific information, including hardware ID and compatible IDs
- The name of a catalog (.cat) file
- Information on how and when to load the services provided by each driver (Windows 2000 and later systems)

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 required. All devices and drivers require an INF file. For more information, see Supplying an INF File.

In addition to the system requirements, drivers often have their own requirements. For example, some drivers use the registry to share information with other components. Through a driver's INF file, you can add, change, or delete information from the registry. The INF file cannot, however, read directly from the registry.

If installation depends on reading from the registry, requesting input from the user, or acquiring any other dynamic information, you will need to supply an additional component, typically a *co-installer*. A co-installer is a DLL that handles device installation function codes sent by the system Setup component. For more information, see Supplying a Co-installer.

If the installation package includes vendor-supplied applications associated with the device, you'll probably also provide a device installation application. For more information, see Supplying a Device Installation Application.

The following table lists the installation components that a device/driver combination might require.

Component	**When required**
INF file	Required for all devices and drivers.
Device co-installer	Required if installation requires information from the user or from the registry. Often supplied by vendors.
Class co-installer	Required only if installation of all devices of this class requires information from the user or from the registry. System-supplied for classes that require them; vendors should rarely supply them.
Class installer	Required only for new device setup classes; system-supplied for existing classes. As a rule, vendors should not define new setup classes, and thus should not supply class installers.
Device installation application	Recommended when value-added software is provided with device and driver, or when driver

	replaces a previous driver that might already be installed

If your device is involved in booting the system, installation requirements differ. See Installing a Boot Driver.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Supplying an INF File

Every driver package must include an INF file, which the system Setup component reads 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 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 the Device Manager.

The exact contents and format of the INF file depend on the device setup class. The Summary of INF Sections describes the information required in each type of INF. In general, per-manufacturer information resides in an INF **Models** section; entries in the **Models** section refer to INF *DDInstall* sections that contain model-specific details.

The GenINF tool, provided in the *\tools* directory of the DDK, creates a skeleton ASCII INF file (Windows 2000 or later) for a device of any standard setup class. You must edit the resulting file to supply device-specific information, and to support additional operating systems or hardware platforms.

The ChkINF tool, in the same directory, checks the syntax and structure of all cross-class INF sections and directives, along with the class-specific extensions for all setup classes except for Printers.

You can use a single INF file for installation on all Windows 2000 and later operating systems and on Windows 98/Me. 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 rather than ASCII.

For more information, see Creating an INF File, the documentation for GenINF and ChkINF, the device-specific documentation in the DDK, and the INF files supplied with sample drivers for devices similar to yours.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Supplying a Co-Installer

A co-installer is a user-mode Win32 DLL. Typically, a co-installer performs installation tasks that require dynamic information that is not available when the INF for its device or device class is written. For example, a co-installer is typically used when the user must supply information for device-specific settings through a property page.

A co-installer can perform installation operations for one particular device or for all the devices in a setup class.

- A *device co-installer* handles custom installation requirements for a specific device or family of devices. For example, a hypothetical *xyzcam.dll* device co-installer could support all the still-image cameras shipped by the *Xyz* Company.
- A *class co-installer* performs identical actions for every device in its class. For example, a hypothetical *image.dll* co-installer could support all the possible devices of the imaging class, such as still-image capture devices, digital cameras (including that of the *Xyz* Company), and scanners.

Microsoft supplies class co-installers for classes that require them; third parties seldom do. Do not supply a class co-installer with your device unless you can guarantee that every device in its setup class requires the actions of your class co-installer.

On the other hand, vendors often supply device co-installers. You should write a device co-installer if you need to display property pages or custom Finish pages during device or driver installation.

For more information, see Writing a Co-Installer and Providing Device Property Pages.

If your co-installer displays a custom Finish page or other user interface elements, make sure that those elements conform to the user interface guidelines described in the MSDN® Library and the Wizard 97 section of the *Platform SDK*.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Supplying a Device Installation Application

A device installation application can handle a variety of installation tasks, from setting up the hardware to installing device-specific application software.

Consider writing a device installation application if either of the following apply:

- You supply value-added software that should be installed at the same time as the driver.

- The driver shipped with your device replaces a similar driver that ships with the operating system. For example, a generic driver for your device ships with the operating system, but you supply a driver with additional features designed specifically for your device.

Any device installation application should be able to handle the following scenarios:

1. The user installs the software package before installing the hardware.
2. The user installs the hardware before installing the software package.

For more information, see Writing a Device Installation Application.

A device installation application should conform to the user interface guidelines described in the MSDN Library and in the Wizard 97 section of the *Platform SDK*.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Guidelines for Device and Driver Installation

The fundamental goal for device/driver installation on Windows® operating systems is to make the process as simple as possible for the user. Your installation procedures should work seamlessly with the operating system's setup components. Ideally, device installation should not require any user prompts, except for the system's request to insert the distribution medium.

Some devices, however, require user settings that are input to property pages or custom Finish pages. Such pages can be displayed by a device co-installer after the driver has been loaded and started. Other devices ship with value-added software that is most readily installed at the same time as the device itself, by means of a device installation application.

In either case, any user interface presented as part of installation should conform to the look and feel of the Windows user interface in general and, more specifically, the Device Manager and the device installation wizards. (The wizard names vary from one operating system release to the next, but include Found New Hardware, Add/Remove Hardware, Hardware Update, and so on).

To provide the best possible user experience, follow these rules in designing and implementing your installation procedures:

- Do not reboot the system or require the user to do so.
- 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 the Device Manager.
- Use system-supplied interfaces (including user interface items) whenever possible.
- Conform to the Windows user interface design specifications described in the MSDN® Library, available at msdn.microsoft.com.
- Although you cannot use the Windows Installer to install a device or driver, you should use it to

install value-added software that ships along with your driver package. The Windows Installer is described in the *Platform SDK*.
- 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.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Distributing a Driver

A driver package can be distributed through the Windows® Update program if it has passed WHQL testing, received a digital signature, and qualified for the Windows® Logo program. This section includes basic information on the following to help guide you through the process:

Driver Signing

Protection for System Files

Windows Update

Program requirements are frequently updated, so be sure to check the web sites referenced in the text for the latest information.

Built on Thursday, July 19, 2001

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

Getting Started with Windows Drivers: Windows DDK

Driver Signing

Driver packages that pass WHQL testing can be digitally signed by Microsoft. Driver packages must be digitally signed if they are to be distributed through the Windows Update program or any other Microsoft-supported distribution mechanism.

You can apply for a digital signature through the WHQL web site at www.microsoft.com/hwtest. The digital signature does not change the driver binaries or the INF file submitted for testing.

To receive a digital signature, a driver package must meet the following criteria:

- A WHQL test program must be available for the driver, and the driver must have passed the test. For more information, see "Designed for Windows" Logo Testing.
- The driver must be installed by an INF file.

- The INF file must not contain errors that can be identified by the ChkINF or *INFCatReady* (*InfCatR.exe*) programs.
- The driver package must not include replacements for, or duplicates of, any Microsoft-supplied system files.

After your driver package has been signed, you should test it in every supported installation path on every target operating system to make sure that it installs without any errors or warnings.

See the following for additional information about digital signing:

The Catalog File

Enforcement of Digital Signatures

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

The Catalog File

A digital signature consists of a digitally signed catalog (*.cat*) file provided by Microsoft. The *.cat* file contains a record of every file that the driver package copies to the system.

The *.cat* file is a collection of tags, each of which corresponds to a file installed by the driver package. On Windows® 2000 and later, and on Windows Millennium Edition (Me), each tag is a cryptographic checksum value. On Windows 98, each tag is a text filename. Regardless of the intended operating system, WHQL uses cryptographic technology to digitally sign the catalog file.

The **CatalogFile** entry in the INF **Version** section of the driver's INF file specifies the name of the catalog file. At testing time, WHQL reads this entry and generates a catalog file with the specified name.

At driver installation, the system uses the **CatalogFile** entry to identify and validate the catalog. The system copies the catalog file to the *%System%\CatRoot* directory, and copies the INF to the *%System%\Inf* directory.

Changing or replacing any of the files in the driver package, including the INF file or the catalog file itself, breaks the digital signature. Even a single-byte change, for example, to correct a misspelling, will break the signature, and you will have to resubmit the package to WHQL for a new signature.

Similarly, changes to the device 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, such a driver package must be resubmitted to WHQL even if the binaries do not change.

For more information, see Enforcement of Digital Signatures.

As a general rule, there should be one catalog file for each INF in a driver package. If the driver package installs the same binaries on both Windows 98/Me and Windows 2000 and later platforms, the INF can contain a single, undecorated **CatalogFile** directive. However, if the package installs different binaries, the INF should contain decorated **CatalogFile** directives. See the INF **Version** Section for details.

If you have more than one driver package, you should create a separate catalog file for each such package, and give each catalog file a unique name. Two unrelated driver packages cannot share a single catalog file. A single driver package that serves multiple devices needs only one catalog file.

Signed catalog files are valid for 20 years.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Enforcement of Digital Signatures

Windows checks the digital signature any time the driver's INF is used to install the driver; that is, when any of the following occurs:

- A plug and play event occurs for any of the driver's devices.
- The Add Hardware Wizard runs.
- The user selects **Update Driver**.
- An application calls the **UpdateDriverForPlugAndPlayDevices** function.
- An application calls the **SetupCopyOEMInf** function. This function is documented in the Windows SDK.

At driver installation, Windows inspects the INF to determine the name of the catalog file, then finds the catalog file and verifies the digital signature. For each file installed as part of the driver package, the system verifies its checksum against the checksum listed in the catalog file.

If the driver signature is incorrect or if any file's checksum differs from the one in the catalog file, Windows takes one of the following actions, depending on the Driver Signing Options set by the administrator of the local machine:

- Warns the user
- Blocks installation
- Ignores the error and installs the driver anyway

Windows 2000 and Later Behavior

By default, Windows 2000 and later systems warn the user for most device setup classes.

Windows 2000 trusts the **DriverVer** entry (which specifies the date of the driver package) in the INF only if the driver package is signed.

Windows Me Behavior

For the following device setup classes, Windows Me blocks installation of unsigned drivers if a signed driver already exists on the system:

- Media, including WDM and VxD audio, joysticks, and certain other multimedia devices
- Display

At installation, Windows Me checks both the device setup class in the INF and the device ID of the device. If a signed driver from the correct device setup class matches the device ID, Windows Me uses the driver package with the closest match, and then checks to ensure that the package has been signed. (See How Setup Selects Drivers for more information on device IDs, compatible IDs, and driver matching.) If Windows Me cannot find a driver package that matches the device ID, it uses the INF that most closely fits.

During an upgrade, Windows Me does not replace a working driver for the Display or Media classes, unless known problems exist in that driver.

Windows Me always trusts the **DriverVer** entry in the INF.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Protection for System Files

Windows File Protection (Windows 2000 and later) and System File Protection (Windows Me) protect Windows operating system files from being replaced with unknown or incompatible versions. System File Protection is not supported on Windows 98.

File protection is implemented slightly differently on Windows 2000 and later systems, and on Windows Me, as described in the next two sections.

Built on Thursday, July 19, 2001

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

Getting Started with Windows Drivers: Windows DDK

Windows File Protection (Windows 2000 and later)

Windows File Protection (WFP) protects system files required for Windows 2000 and later operation. Protected system files include .sys, .exe, .ocx, and .dll files that ship "in the box" with the operating system.

During WHQL testing, the *InfCatReady* program checks a driver's INF to ensure that it does not attempt to replace system files. A driver package that attempts to replace system files cannot receive a digital signature. A driver package can, however, contain updated versions of files that the vendor supplied Microsoft to ship with Windows 2000 or later.

For additional information about Windows File Protection, see documentation in the Platform SDK.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

System File Protection (Windows Me only)

On Windows Me, System File Protection extends to all Microsoft system files. A list of protected files appears on the Windows Me CD in the *windows\system\restore* directory. Third-party and "in-box" drivers are not protected. System File Protection is not related to driver signing.

Only Microsoft-approved redistribution packages can update protected system files. Driver packages cannot replace protected files, even if the driver package has a digital signature.

If a driver or application tries to replace a protected file, the system automatically restores the original, protected version of the file.

Built on Thursday, July 19, 2001

Getting Started with Windows Drivers: Windows DDK

Windows Update

The Microsoft® Windows® Update web site offers downloadable drivers for Windows 2000 and later, and Windows 98/Me.

Driver packages that pass WHQL testing and receive the Windows Logo can be considered for publication on Windows Update if they meet certain additional requirements.

These requirements ensure that Windows Update can determine the correct driver package for the user's device, legally distribute it, and automatically download it.

For more information about how to have your driver published on the Windows Update web site, see http://www.microsoft.com/hwdev/ntdrivers/winup.htm.

Built on Thursday, July 19, 2001
