Using DriverStudio Development Tools

Release 3.2

COMPUWARE

Technical support is available from our Technical Support Hotline or via our FrontLine Support Web site.

Technical Support Hotline:
1-800-538-7822

FrontLine Support Web Site:
http://frontline.compuware.com

October 13, 2004

Table of Contents

Preface

Purpose of This Manual ix
What This Manual Covers x
Conventions Used In This Manual xi
Accessibility xi
Customer Assistance xii
For Non-Technical Issues xii
For Technical Issues xii

Chapter 1

Welcome

Product Description 2
Driver Development Environment 2
What is DriverWorks? 3
What is DriverNetworks? 4
Driver Code Generation Wizards 5
On-line Help 6
Example Drivers 6
DriverMonitor™ 7
EzDriverInstaller 7
WDM Sniffer 7
Advantages of the C DriverWizard Over the DDK 8
Advantages of DriverWorks/DriverNetworks Over the DDK 10
What Types of Drivers Does The C DriverWizard Help to Develop? 13
What Types of Drivers Does DriverWorks Help to Develop? 13
What Types of Drivers Does DriverNetworks Help to Develop? 14

Chapter 2

DriverStudio Development Environment

Visual Studio Integration 15
Building Drivers 17
Building Drivers Using Visual Studio 18
- Setting Up the Visual Studio IDE 18
- DDK Build Settings 18
- Building Drivers Using Visual Studio and the BUILD Utility 20
- Building Drivers Using Visual Studio Build Commands 21
- Building DriverStudio Drivers 21
- Building DriverWorks Libraries 22
- Building DriverNetworks Libraries 25

Using DriverWizard 29
- Understanding DriverWizard 30

Chapter 3

DriverWorks Object Model

The Driver Object 34
- Initialization 34
- Reinitialization 35
- Dispatching Requests 35
- Unloading 36
- Image Section Objects 36
- I/O Request Objects 37

Device Objects 39
Lower Device Objects 41
- Filter Device Objects 42

Objects Used During Initialization 43
- Driver Parameters and Registry Objects 43
- Configuration Query Objects 43
- Resource Request and Assignment Objects 44

Queue Objects and Request Serialization 45
Driver Managed Queues 46
Controller Objects 47
Interrupt Request Levels (IRQL) 48
Objects for Controlling Hardware 49
- Peripheral Address Objects 49
- Interrupt Objects and Deferred Procedure Calls 52
- Objects for DMA 53
- DMA Adapter Objects 54

Common Buffer Objects 55
DMA Transfer Objects 55
Memory Objects 56
Spin Lock Objects 57
Dispatcher Objects 57
Containers and Other Objects 61
List Objects 61
FIFO Objects 61
Files 62
Event Log Entries 62
Strings 62
Heaps 63
Chart of DriverWorks Classes 64

Chapter 4

Windows Driver Model

Overview of the WDM 67
Plug and Play and Power Management 68
Class Drivers and Minidrivers 69
Bus Drivers 70
Windows Management Instrumentation 70
WDM and DriverWorks 72
Writing a WDM Driver with DriverWorks 72
WDM Documentation 74
Class KPnpDevice Overview 75
PnP and Power Functionality Implemented in Class KPnpDevice 77
Class KPnpDevice PnP Policies 79
Class KPnpDevice Power Policies 94
Class KPnpLowerDevice 107
Overview 107
Plug and Play Resource Management 108
Making a WDM Driver Callable by a VxD 109
Overview 109
WDM Device Interfaces 111
Using Device Interfaces from the Application Level 112
WDM Power Management 113
Adding WMI Support to a Driver 115
Identify or Create the CIM Classes for Data that the Driver will Publish 115
Define C Structures that Correspond to the CIM Classes 117
Add Instances of Template Class KWmiDataBlock 118
Identify the Method Object and Declare the Array of Method Members 119
Add Code to Register and Deregister a Device's WMI Blocks 120

Add a Handler For IRP_MJ_SYSTEM_CONTROL 121
Specialize Members of Template Instances as Necessary 122
Modify the Build Procedure 122
Classes for Human Interface Device Support 123
Writing an HID Minidriver 124
Universal Serial Bus (USB) Drivers 126
Classes for USB Support 126
Class KUsbLowerDevice 127
Class KUsbInterface 128
Class KUsbPipe 129
IEEE 1394 Drivers 130
Address Space 130
Asynchronous and Isochronous Transfers 130
Bus Resets 131
IEEE 1394 Bus Requests (IRBs) 131
Support for Asynchronous and Isochronous Transfers 131
WDM Streaming Drivers 132
Classes for Stream Minidrivers 132
Relationships and Flow of Control 135
Flow of Control 136
Creating the Filter Driver 136
Creating the Adapter 137
Opening a Stream 137
Stream Control and Data 137
Summary of Initialization Logic 138
Closing Streams Adapter Shutdown 138
More Information 138
Steps for Building a Stream Minidriver 139

Chapter 5

DriverNetworks Object Model

Introduction 141
NDIS Miniport Driver Framework 143
NDIS Connection-Oriented Miniport Driver Framework 146
NDIS Miniport Call Manager Framework 149
NDIS Intermediate Driver Framework 154
NDIS Intermediate (Mux) Drivers 157
NDIS Filter Drivers 160
NDIS Protocol Driver Framework 163
NDIS Packet Driver Framework 167

NDIS Transport Driver Framework 169
Creating a Transport Address Object 173
Notify Object Framework 174
TDI Client (DriverSockets) Framework 178
Protocols and Address Families 179
Programming Model 179
DriverSockets Examples 180
DriverSockets vs. WinSock 182
Features of DriverSockets vs. Winsock 183

Appendix A

Other Resources

Microsoft DDK 187
On-line Resources 187
Index 189

Preface

- Purpose of This Manual
- What This Manual Covers
- Conventions Used In This Manual
- Accessibility
- Customer Assistance

Purpose of This Manual

Congratulations on selecting DriverWorks™ and DriverNetworks™ for your Windows device driver development projects. Whether you are a driver guru or facing your first driver development project, these tools offer significant productivity gains over the Microsoft® DDK alone.

DriverWorks and DriverNetworks offer a development environment designed to make it easier to understand and develop software drivers for Windows devices. DriverWorks and DriverNetworks include class libraries, driver development wizards, on-line references and how-to Help files, example drivers, associated utility programs, and full library source code.

With powerful and sophisticated code generation wizards and tens of thousands of lines of tested code in both libraries and sample drivers, DriverWorks and DriverNetworks are the most complete and the easiest solutions for Windows NT family and WDM driver development.

This manual is intended for programmers who want to use DriverWorks and DriverNetworks to build drivers for Windows ME and Windows NT/2000/XP platforms.

Users of previous versions of DriverWorks and DriverNetworks should read the Release Notes to see how this version differs from previous versions.

This manual assumes that you are familiar with the Windows interface and with software driver development concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

The *Using DriverStudio Development Tools* manual is organized as follows:

- Chapter 1, “Welcome”

  This chapter briefly describes the components and features of the DriverStudio Development Tools, including DriverWorks and DriverNetworks.

- Chapter 2, “DriverStudio Development Environment”

  This chapter describes how DriverStudio enables Microsoft Visual Studio to be a driver development environment. This chapter also describes the DriverWizard, and Network Driver Wizard.

- Chapter 3, “DriverWorks Object Model”

  This chapter provides an overview of the DriverWorks object model. Read this chapter to gain an understanding of the classes that comprise DriverWorks and how they relate to each other.

- Chapter 4, “Windows Driver Model”

  This chapter describes the Windows Driver Model (WDM) programming environment, and how DriverWorks supports the WDM.

- Chapter 5, “DriverNetworks Object Model”

  This chapter provides an overview of the DriverNetworks object model. Read this chapter to gain an understanding of the classes that comprise DriverNetworks and how they relate to each other.

- Appendix A, “Other Resources”

  This appendix describes other resources available to the driver writer.

- Index

Conventions Used In This Manual

This book uses the following conventions to present information:

Enter	Indicates that you should type text, then press RETURN or click OK.
italics	Indicates variable information. For example: *library-name*.
`monospaced text`	Used within instructions and code examples to indicate characters you type on your keyboard. Also, computer commands and file names appear in `monospace typeface`. For example: The ***Using SoftICE*** manual (`Using SoftICE.pdf`) describes...
SMALL CAPS	Indicates a user-interface element, such as a button or menu.
UPPERCASE	Indicates key words and acronyms.
Bold typeface	Screen commands and menu names appear in **bold typeface**. For example: Choose **Item Browser** from the **Tools** menu.
`italic monospace`	Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in *`italic monospace type`*. For example: Enter `http://`*`servername`*`/cgi-win/itemview.dll` in the **Destination** field...

Accessibility

Prompted by federal legislation introduced in 1998 and Section 508 of the U.S. Rehabilitation Act enacted in 2001, Compuware launched an accessibility initiative to make its products accessible to all users, including people with disabilities. This initiative addresses the special needs of users with sight, hearing, cognitive, or mobility impairments.

Section 508 requires that all electronic and information technology developed, procured, maintained, or used by the U.S. Federal government be accessible to individuals with disabilities. To that end, the World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) has created a workable standard for online content.

Compuware supports this initiative by committing to make its applications and online help documentation comply with these standards. For more information, refer to:

- W3C Web Accessibility Initiative (WAI) at www.W3.org/WAI
- Section 508 Standards at www.section508.gov
- Microsoft Accessibility Technology for Everyone at www.microsoft.com/enable/

Customer Assistance

For Non-Technical Issues

Customer Service is available to answer any questions you might have regarding upgrades, serial numbers and other order fulfillment needs. Customer Service is available from 8:30 am to 5:30 pm EST, Monday through Friday. Call:

- In the U.S. and Canada: 1-888-283-9896
- International: +1 603 578-8103

For Technical Issues

Technical Support can assist you with all your technical problems, from installation to troubleshooting. Before contacting Technical Support, please read the relevant sections of the product documentation as well as the Readme files for this product. You can contact Technical Support by:

- **E-mail:** Include your serial number and send as many details as possible to:

  `mailto:nashua.support@compuware.com`

- **World Wide Web:** Submit issues and access additional support services at:

  `http://frontline.compuware.com/nashua/`

- **Fax:** Include your serial number and send as many details as possible to:

  `1-603-578-8401`

- **Telephone:** Telephone support is available as a paid Priority Support Service from 8:30 am to 5:30 pm EST, Monday through Friday. Have your product version and serial number ready.
  - In the U.S. and Canada, call: 1-888-686-3427
  - International customers, call: +1-603-578-8100

Note: Technical Support handles installation and setup issues free of charge.

When contacting Technical Support, please have the following information available:

- Product/service pack name and version.
- Product serial number.
- Your system configuration: operating system, network configuration, amount of RAM, environment variables, and paths.
- The details of the problem: settings, error messages, stack dumps, and the contents of any diagnostic windows.
- The details of how to reproduce the problem (if the problem is repeatable).
- The name and version of your compiler and linker and the options you used in compiling and linking.

Chapter 1
Welcome

- Product Description
- Driver Development Environment
- What is DriverWorks?
- What is DriverNetworks?
- Driver Code Generation Wizards
- On-line Help
- Example Drivers
- DriverMonitor™
- EzDriverInstaller
- WDM Sniffer
- Advantages of the C DriverWizard Over the DDK
- Advantages of DriverWorks/DriverNetworks Over the DDK
- What Types of Drivers Does The C DriverWizard Help to Develop?
- What Types of Drivers Does DriverWorks Help to Develop?
- What Types of Drivers Does DriverNetworks Help to Develop?

Product Description

The DriverStudio Development Tools consist of development environments, utilities, and frameworks for developing Windows device drivers. DriverStudio enables Microsoft Visual Studio as a driver development environment for Windows device drivers. DriverStudio includes several tools designed to simplify driver development including DriverMonitor, EzDriverInstaller, Wdm Sniffer, and the sources to DSP Converter. DriverStudio also includes DriverWorks and DriverNetworks C++ frameworks to simplify developing device drivers. DriverWorks and DriverNetworks include C++ class libraries (including complete source code), code generation wizards, on-line references, how-to help files, and example drivers for real hardware.

DriverStudio contains the following Driver Development components:

- Visual Studio integration that enables driver development using Visual Studio
- DriverWorks C++ driver development framework for WDM and NT kernel mode drivers
- DriverNetworks C++ driver development framework for network drivers
- Wizards to jump start driver development
- DriverMonitor debugging aid
- EzDriverInstaller driver installation utility
- WDM Sniffer kernel message tracing tool
- On-line Help which supplements DDK help
- Numerous example drivers for real hardware

Driver Development Environment

The Microsoft Driver Development Kit (DDK) does not ship with an Integrated Development Environment (IDE) for developing device drivers. DriverStudio provides Microsoft Visual Studio integration that enables Visual Studio to become a driver development IDE. With DriverStudio integration, Visual Studio can be used to develop, build, debug, test, and tune Windows device drivers. DriverStudio integrates Microsoft Visual Studio with the DDK making driver development and debugging easier.

DriverStudio provides the **Sources to DSP Converter** tool that converts a DDK SOURCES file to a Visual Studio project file. A SOURCES file is the file used by the DDK's BUILD.EXE utility to specify the source files to be built and various custom compiler and linker options. The project converter tool supports converting the SOURCES file to either a Visual C++ 6.0 workspace and project (.dsw and .dsp files), or to a Visual Studio .NET solution and project (.sln and .vcproj files). This allows a DDK project to be loaded into Visual Studio for development.

DriverStudio provides DDK Build Settings and the DDK Build button that allows building a driver from Visual Studio using a DDK SOURCES file and the Microsoft DDK Build utility.

DriverStudio provides integrated code generation wizards into Visual Studio. The wizards can save weeks of development time by providing working skeleton driver code customized to your specific hardware which builds, installs, and runs.

Integrated support for the MSVC Visual Studio provides a complete driver development environment, including checked and free builds selected with a mouse click, the familiar Studio editor and jump-to-error facilities, and class browsing.

What is DriverWorks?

DriverWorks is designed for Windows device driver developers. In its core, DriverWorks is a C++ class library that captures the Windows object model into a set of compact classes and provides a framework for writing WDM and NT kernel mode drivers — quickly, efficiently, and in an object-oriented manner. DriverWorks also includes the Driver Wizard application that generates full driver source code, project files for Microsoft Visual C++, and installation files, so that you can develop and test your driver immediately.

The Windows NT and WDM driver interfaces lend themselves well to an object-oriented approach. DriverWorks takes full advantage of the underlying object-based nature of the operating system and offers the developer a set of classes that present a clearer picture of the Windows NT and WDM architectures. The class library encapsulates tens of thousands of lines of code that are common to many drivers. Common tasks are handled automatically, and many complex tasks are greatly simplified by the classes.

The DriverWorks/DriverNetworks classes serve as a vehicle to deliver to the developer a wealth of detailed knowledge of the system interface. The code has been debugged, field-tested on real drivers, and meets Microsoft certification testing standards such as Windows Hardware Quality Lab (WHQL).

Furthermore, the C++ compiler offers powerful advantages not available in ANSI C, including improved type safety, inline optimizations, and far better code organization.

The developer can use general purpose DriverWorks classes (such as interlocked containers, registry access, STL, and more) without the overhead of the DriverWorks driver/device dispatch framework. This is useful for porting existing driver code, using C++ as a better C, designing a custom framework, or having a procedural driver design but still using the general purpose DriverWorks classes.

DriverWorks contains the following components:

- C++ Class Library (including complete source code)
  - For WDM drivers
  - For NT kernel mode drivers
  - For USB, PCI, 1394, HID, filter, and streaming drivers
- Driver Wizard
- On-line help
- Example drivers for real hardware

DriverWorks supports WDM and NT kernel mode driver development for Windows 9x/ME/NT/2000/XP targets and Windows Server 2003.

What is DriverNetworks?

DriverNetworks is designed for Windows network driver developers. In its core, DriverNetworks is a C++ class library that captures the NDIS and TDI object model into a set of compact classes and provides a framework for writing NDIS and TDI kernel mode drivers — quickly, efficiently, and in an object-oriented manner. DriverNetworks also includes the Network Driver Wizard application that generates full driver source code, project files for Microsoft Visual C++, and installation files, so that you can develop and test your driver immediately.

DriverNetworks contains the following components:

- C++ Class Library (including complete source code)
  - For NDIS drivers
  - For TDI Transports
  - For TDI clients (DriverSockets)
- Network Driver Wizard
- On-line help
- Examples

DriverNetworks supports NDIS and TDI driver development for Windows 9x/ME/NT/2000/XP targets and Windows Server 2003.

Driver Code Generation Wizards

The DriverWorks Driver Wizard, the DriverNetworks Network Driver Wizard, and the C DriverWizard are step-by-step code generation wizards integrated into the Microsoft Visual C++ development environment. You can also access these wizards from the Start menu.

The wizards generate complete source code, Visual C++ project files, and installation files so that you can develop and test your driver immediately. Without any code modifications, the resulting driver will install, load, and initialize. The wizards give a powerful jump-start to any driver development project. The guesswork of how the driver interacts with the operating system can be removed from driver development. Instead, the focus can be on programming the hardware.

The DriverWorks DriverWizard, for example, can generate several device driver project types:

- Windows Driver Model (WDM) function drivers, WDM filter drivers, WDM bus drivers, and NT4 style kernel mode drivers to jump-start development of a new DriverWorks device driver.
- Drivers to control USB, 1394, and PCI hardware devices.
- Simple C++ driver and empty driver projects to give a jump-start for a developer who is unwilling to use DriverWorks framework, but willing to use C++ as a better C, or design a custom framework, or port existing driver code. The developer can still include and use general purpose DriverWorks classes (such as interlocked containers, registry access, STL, and more) without the overhead of the DriverWorks driver/device dispatch framework.

The C DriverWizard provides an alternative for programmers who do not want to use C++ in the Windows kernel. The C DriverWizard also generates several device driver project types including:

- Windows Driver Model (WDM) function drivers, WDM filter drivers, WDM bus drivers, NDIS miniport drivers, NDIS intermediate drivers, NDIS protocol drivers, and NT4 Style kernel mode drivers to jumpstart development of a new device driver.
- Drivers to control USB, 1394, and PCI hardware devices.
- Straightforward non-library based C drivers similar in look and feel to sample drivers available in the Windows DDK.

The DriverStudio DriverWizards provide a great deal of control over the generated code. You can allocate hardware resources, create registry entries and variables to hold registry values, and define device I/O control codes to interface with the application. The DriverWizards automates the creation of drivers with multiple device classes, and generate code to create one or more instances of each device.

On-line Help

The on-line help system provides a comprehensive reference to class libraries, with general overviews and detailed descriptions of every class. The on-line help system also provides “How-To” topics that explain device driver programming. The on-line help files are installed into the `DriverStudio\HELP` directory. The help files are designed to supplement the Microsoft DDK.

Example Drivers

Both DriverWorks and DriverNetworks ship with many complete examples. Numerous examples are similar to Microsoft DDK samples that illustrate driver specific topics. It is easy to compare the code and see the power of using the C++ frameworks. Typically, a driver written with the C++ frameworks contains far less source code, sometimes 50% less code.

DriverWorks and DriverNetworks provide many real hardware samples that are not contained in the Microsoft DDK. This illustrates how to develop real hardware drivers using the C++ frameworks and also how to develop such drivers in general.

Compuware adds examples to the release on a regular basis and additional examples may be available from the Compuware Web site. For a complete and up-to-date list of examples, please see the on-line help and refer to the EXAMPLES directory.

DriverMonitor™

DriverStudio includes DriverMonitor, a utility that allows you to monitor the state of your device driver without requiring a debugger, the debug version of the operating system, or a second computer.

DriverMonitor serves two primary purposes:

- It enables you to display formatted output from your driver (running on ether a local or remote machine), including a powerful filtering capability based on regular expressions.
- It allows you to dynamically install, configure, start, and stop an NT4 kernel mode driver for testing.

You may redistribute the DriverMonitor executable (MONITOR.EXE) and the related message driver (DBGMSG.SYS or NDBGMSG.VXD) and DLL (DBGMSGCFG.DLL) to your customers. This allows you to embed diagnostic information in your driver (perhaps enabled with a registry switch or IOCTL code) to help debug problems that occur in the field.

EzDriverInstaller

EzDriverInstaller is a tool that will install WDM drivers without having to go through the Windows Device Manager or the “Add New Hardware” wizard. EzDriverInstaller has the capability of installing, uninstalling, enabling, disabling, restarting, and updating drivers. Using EzDriverInstaller is much faster and easier than using the Windows Device Manager.

WDM Sniffer

WdmSniffer is a simple kernel message-tracing tool. It installs a pass-through upper filter driver for a selected device or a class of devices, and records the messages sent to that device driver. You can filter the messages by message type, destination device, and message data. You can then save the resulting log in a file, which you can load at a later time.

WdmSniffer has a built-in knowledge of several types of device messages:

- IRP
- USB
- 1394 (Firewire)
- SCSI
- NDIS
- TDI

For these types of messages it logs the request header and the request data, along with timestamp and request direction information.

WdmSniffer can be useful in following situations:

- To better understand the communication protocol between two drivers
- To monitor device activity
- To record a set of commands leading to a device or system failure

Advantages of the C DriverWizard Over the DDK

The DriverStudio C DriverWizard saves considerable time over using just the Windows DDK to develop drivers.

Table 1-1. C DriverWizard vs. the Windows DDK

Feature	C DriverWizard Support	DDK Support
Wizards	Yes	No
Visual Studio Support	Yes	No
Development Time	Fast	Slow
Complete Driver Code	Yes	No

Powerful Jump Start

The C DriverWizard provides a powerful jump-start for your driver development effort. The starting curve is one of the most time consuming parts of a kernel mode driver development cycle. These tools save you time by generating a complete set of necessary source, project, and install files, and by providing the user with a complete working driver. Getting started with just the Microsoft Windows DDK is difficult. It requires tricky cut and paste code modifications to DDK samples that are notoriously buggy and incomplete. The DDK samples are not customized to specific hardware. It is not clear where in the DDK sample code to make modifications. It can take weeks to modify a DDK sample to your specific driver and hardware requirements.

Removes Guesswork

The C DriverWizard implements a significant amount of the glue and boilerplate code typical, for instance, for plug and play, power management, WMI, and driver initialization and shutdown. This removes the guesswork from the initial stages of your project and allows you to focus on the driver design decisions specific to your hardware. The C DriverWizard can even generate a GUI Win32-based test app, class and co-installer DLLs, and NDIS NotifyObject DLLs if desired.

Saves Time

Using the C DriverWizard results in faster development time. Hundreds or thousands of lines of tested and customized C code are generated for the user. This results in faster development time, fewer bugs, and less maintenance. No more cut and paste errors, and no more mistakes in code that the user has written a hundred times before. The code is plain C, so it is easily customizable and extensible.

Advantages of DriverWorks/DriverNetworks Over the DDK

The DriverStudio C++ frameworks save considerable time over using just the Windows DDK to develop drivers.

Table 1-2. Frameworks vs. the Windows DDK

Feature	Frameworks Support	DDK Support
C++	Yes	No
Wizards	Yes	No
Visual Studio Support	Yes	No
Real Hardware Samples	Yes	No
Development Time	Fast	Slow
Reusable Code	Yes	No
WHQL Certifiable Code	Yes	Maybe
Built-In Tracing & Diagnostics	Yes	No
Uniform Driver Structure/Design	Yes	No
Multi-Platform Support	Yes	Some

Powerful Jump Start

DriverWorks and DriverNetworks provide a powerful jump-start for your driver development effort. The starting curve is one of the most time-consuming parts of a kernel mode driver development cycle. These tools save you time by generating a complete set of necessary source, project, and install files, and by structuring your code based on the reusable class framework. Getting started with just the Microsoft DDK is difficult. It requires tricky “cut and paste” code modifications to DDK samples that are notoriously buggy and incomplete. The DDK samples are not customized to specific hardware. It is not clear where in the DDK sample code to make modifications. It can take weeks to modify a DDK sample to your specific driver and hardware requirements.

Removes Guesswork

DriverWorks and DriverNetworks implement and hide a significant amount of the glue and boilerplate code typical, for instance, for plug and play, power management, driver initialization and shutdown. This removes the guesswork from the initial stages of your project and allows you to focus on the driver design decisions specific to your hardware.

Object Oriented

DriverWorks and DriverNetworks provide the developer with all of the design power and object orientation of C++. The class libraries offer many thousands of lines of tested code that reduce many complex tasks to simple library calls. In addition to the general driver framework, both DriverWorks and DriverNetworks are comprised of a number of useful classes that simplify programming with common NT objects, such as IRPs, hardware resources, file and registry access, synchronization objects, and more. They also encapsulate a number of design patterns typical for driver implementation, such as DMA and IRP queuing. The end result is driver code that facilitates quicker understanding, debugging, and maintenance. Using just the DDK requires that many thousands of lines of driver code be written in the C language.

Hardware Samples

DriverWorks and DriverNetworks have numerous samples for real hardware. These samples illustrate driver design techniques for specific hardware. DriverWorks is designed with “depth” where supported types of drivers have extensive support via samples and reusable code. The DDK has many samples that illustrate the operating system and many types of drivers but lacks real hardware samples. The DDK samples cover a broad range of operating system issues, but lack depth into real hardware samples.

Portable Code

DriverWorks and DriverNetworks support many Windows operating systems. Differences among the many versions of Windows are hidden within the C++ framework reducing the amount of platform dependent code written by the developer. DriverWorks WDM and NT kernel mode driver C++ classes are compatible with DriverNetworks NDIS and TDI C++ classes. This yields significant power and ease to add networking access to a WDM driver or add WDM support to an NDIS driver. Also, DriverWorks, DriverNetworks, and VtoolsD feature Device Access Architecture, which is a set of classes that are source code compatible with VtoolsD for Windows `.vxd` drivers. Device Access Architecture is designed to provide both optimal performance and simple, common objects and interfaces offering source code portability among various Windows platforms. The Windows DDK has limited support for legacy operating systems. Designing drivers with just the DDK to support multiple platforms requires tricky modifications to source code.

Saves Time

Using DriverWorks and DriverNetworks results in faster development time. Thousands of lines of code in the C++ libraries can be used. Many users of the framework have tested this code for years. The code is tested regularly using the Windows Hardware Quality Labs (WHQL) tests. This results in less code for the driver developer to implement. The source code reduction can be as much as 50% less code to develop. This results in faster development time, fewer bugs, fewer WHQL test failures, and less maintenance. There is no reusable code in the Windows DDK. The driver developer has to write many thousands of lines of complicated code in the C language. Also, the code has to be debugged and meet WHQL testing standards.

Easy Debugging

DriverWorks features a built-in tracing facility for the DriverStudio BoundsChecker events. The DriverWorks C++ library calls generate BoundsChecker events. When used in conjunction with the DriverStudio SoftICE and DriverWorkbench components, it significantly simplifies the debugging of your driver. BoundsChecker is a powerful tool to study the operation of your driver and the DriverWorks framework.

WHQL Certifiable

DriverWorks and DriverNetworks promote WHQL support. The product is tested against available WHQL tests and DriverVerifier. The benefit of using thousands of lines of tested and debugged code gets you over the WHQL hurdle. This reduces the risk of having an expensive WHQL failure.

What Types of Drivers Does The C DriverWizard Help to Develop?

C DriverWizard provides extensive support for the following types of Windows device drivers:

- Windows Driver Model (WDM) drivers.
- NT kernel mode (kernel service) drivers
- WDM function drivers
- WDM filter drivers
- WDM bus drivers
- Virtual serial drivers
- USB drivers
- 1394 drivers
- PCI drivers
- ISA drivers
- Human Interface Device (HID) drivers
- TDI Client drivers
- NDIS Miniport Drivers
- NDIS Intermediate Filter Drivers
- NDIS Intermediate MUX Drivers
- NDIS Protocol Drivers

What Types of Drivers Does DriverWorks Help to Develop?

DriverWorks provides extensive support for the following types of Windows device drivers:

- Windows Driver Model (WDM) drivers.
- NT kernel mode drivers
- WDM function drivers
- WDM filter drivers
- WDM bus drivers
- WDM multifunction drivers
- USB drivers
- 1394 drivers
- PCI drivers
- ISA drivers
- PCMCIA drivers
- NT filter drivers
- Human Interface Device (HID) drivers
- Kernel streaming drivers
- AvStream drivers
- IP Filter Drivers

- TDI Client drivers (using the DriverNetworks DriverSockets classes)
- NDIS Protocol Drivers (using DriverNetworks)

What Types of Drivers Does DriverNetworks Help to Develop?

DriverNetworks provides extensive support for the following types of Windows network device drivers:

- Network Driver Interface Specification (NDIS) drivers
- NDIS Miniport drivers
- NDIS WDM Miniport drivers (using DriverWorks)
- Connection-oriented NDIS Miniport drivers
- NDIS Intermediate Filter drivers
- NDIS Intermediate Mux drivers
- NDIS Protocol drivers
- PnP Transports
- TDI Clients
- Notify Object DLLs

Chapter 2
DriverStudio Development Environment

- Visual Studio Integration
- Building Drivers
- Building Drivers Using Visual Studio
- Using DriverWizard

Windows Driver Development is accomplished by using the Microsoft Windows Driver Development Kit (DDK). The DDK provides build environments, header files, libraries, utilites, and extensive documentation required for Windows driver development. However, the DDK lacks ease of use features such as an Integrated Development Environment (IDE), Wizards to jump-start driver development, and real hardware samples.

DriverStudio functions with the DDK Build Environments. DriverStudio also provides Visual Studio integration for those users who would like to optionally use the Visual Studio IDE for driver development.

Visual Studio Integration

DriverStudio enables the popular Microsoft Visual Studio 6 and Visual Studio .NET IDE to be used as an IDE for Windows driver development using the DDK. With DriverStudio integration, Visual Studio can be used to develop, build, debug, test, and tune Windows device drivers. DriverStudio integrates Microsoft Visual Studio with the DDK making driver development and debugging easier.

To quickly get started using Visual Studio with an existing driver or DDK sample driver, DriverStudio provides the **Source to DSP Converter** tool that converts a DDK SOURCES file to a Visual Studio project file. A SOURCES file is the file used by the DDK's BUILD.EXE utility to specify the source files to be built and various custom compiler and linker options. The project converter tool supports converting the SOURCES file to either a Visual C++ 6.0 workspace and project (.dsw and .dsp files), or converting the SOURCES file to a Visual Studio .NET solution and project (.sln and .vcproj files). This allows a DDK project to be loaded into Visual Studio for development.

To quickly get started using Visual Studio for a new driver project, DriverStudio provides integrated code generation wizards into Visual Studio. The wizards generate all of the source files and Visual Studio project files to use Visual Studio to develop your driver. Without any code modifications, the driver source code can be compiled and linked into a working driver. Using the install file generated by the wizard, the driver can be installed and run properly in the system. All without writing any source code. The wizards can save weeks of development time by providing working skeleton driver code customized to your specific hardware which builds, installs, and runs.

DriverStudio provides DDK Build Settings and the DDK Build button which allows building a driver from Visual Studio using a DDK SOURCES file and the Microsoft DDK Build utility.

Integrated support for the MSVC Visual Studio provides a complete driver development environment, including checked and free builds selected with a mouse click, the familiar Visual Studio editor and jump-to-error facilities, and class browsing.

Using DriverWorks and DriverNetworks, you can start your driver development project two ways:

- Modify one of the numerous DriverWorks or DriverNetworks samples.
- Use the wizards to generate a new driver.

If one of the samples is similar to the driver you plan to develop, the most efficient solution might be to modify one of the samples provided. Otherwise, use the wizards.

Building Drivers

In order to build a device driver, the Microsoft Windows DDK is required. The DDK contains all of the necessary headers, libraries, compilers, and linkers required to build a device driver. The DDK-recommended way to build device drivers is using the BUILD utility shipped with the DDK. After installing the DDK, numerous "Build Environment" shortcuts are available from the Start menu.

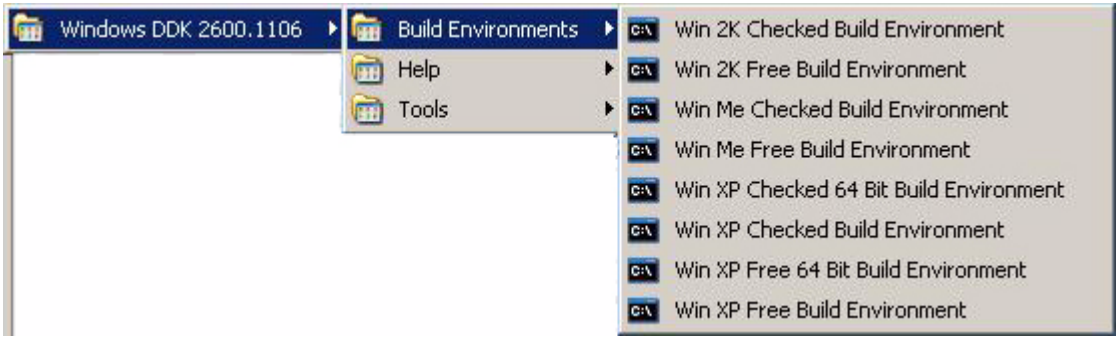


Figure 2-1. Build Environment Shortcuts

These shortcuts invoke a batch file to setup the device driver build environment within a Command Prompt shell. The build environment is selected based on the target platform (x86 32-bit builds, IA64 64-bit builds, etc.) and the target operating system (Windows XP, Windows 2000, etc.). The build environment also allows checked or free builds.

The BUILD utility requires a `SOURCES` file and `MAKEFILE` — each are provided for all DriverStudio samples and wizard projects. Build your driver by selecting a build environment shortcut such as "Win XP Checked Build Environment". This launches a command prompt shell. At the command prompt, browse to the folder containing the driver project with the `SOURCES` file. Simply type `BUILD` at the command prompt. The driver will be built automatically. The BUILD utility contains numerous options.

For a complete list of BUILD options, type:

```
build -?
```

To help debug problems encountered using BUILD, use the –e switch to generate log files. Review the `BUILD.WRN` and `BUILD.ERR` files to ensure no warnings or errors are reported. `BUILD.LOG` contains the full output from the build process. Use the –cef switches to force a complete rebuild with log files:

```
build -cef
```

Note: See the DDK for further details about the BUILD utility and all of the details about SOURCES files.

Building Drivers Using Visual Studio

The following sections describe how to build drivers using Visual Studio.

Setting Up the Visual Studio IDE

It is important to set up the Visual Studio development environment correctly to ensure that drivers are able to build with the DDK.

The DriverStudio integration for both Visual Studio 6 and .NET allows you to run driver builds from the Visual Studio IDE. Given a DDK directory, it can set up the correct DDK build environment for your driver. This includes setting up include and library paths as well as selecting the correct compiler to use. Note that the XP (or later) DDK requires that drivers be built with the DDK compiler rather than the default Visual C++ compiler. If the XP (or later) DDK is selected, the DriverStudio add-in will substitute the DDK compiler for the default Visual Studio compiler when using the standard Build commands from Visual Studio. The advantage of this is that Visual Studio does not have to be re-launched to use a different DDK build environment. You can continue working in one session of Visual Studio and build against different DDK Build environments.

DDK Build Settings

To setup the Visual Studio IDE, use the DriverStudio DDK Build Settings dialog from within the IDE. This feature enables changing the active DDK without leaving the IDE. Changing the active DDK redefines DDK environment variables such as BASEDIR. You can also change Target Operating System and DDK compiler and linker options.

The DDK options available are described in the following sections.

DDK Root Directory (BASEDIR)

Use the DDK directory combo box to select the DDK you wish to build against. The dialog automatically detects the following installed DDKs and puts them in the drop-down menu for the DDK:

- Windows Server 2003 DDK
- Windows XP DDK (Build 2600) and SP1 (Build 2600.1106)
- Windows 2000 DDK
- Windows 98 DDK

DDKs need to be installed in the current operating system in order to be detected by the setup DDK environment. If the DDK you wish to select does not appear in the list, or if a DDK was installed on another operating system on a multiple boot machine, you may type in or browse to the DDK directory.

Target OS

Note: The Target OS option is only available when the XP (or later) DDK is selected.

The XP (or later) DDK provides separate build environments for drivers meant to run on different operating systems. Each environment has different header and library files. For example, the XP DDK recommends using the XP build environment so that drivers can take full advantage of the features available in Windows XP. On the other hand, the Windows 2000 environment is meant for drivers that are required to run on Win2K machines as well as WinXP machines. Select a build environment that is best suited to your driver.

Windows DDK Compiler Support

Note: DDK compiler support is only applicable when the XP (or later) DDK is selected.

The following DDK options are available:

- Disable for all projects.
  Select **Disable for all projects** to use the default Visual Studio compiler for all projects, even drivers. Note that the XP (or later) DDK header files do not support building with older compilers like Visual Studio 6.
- Enable only for DriverStudio, DDK or 64-bit C/C++ projects.
  The default selection **Enable only for DriverStudio, DDK or 64-bit C/C++ projects** indicates that the DriverStudio add-in will enable the DDK compiler only for driver projects. This means that, when a driver is built, the correct Windows DDK compiler will be used to build the driver. The DriverStudio add-in uses the following criteria to determine if a project is a driver project:
  - Detection of `CRT_INC_PATH`, `DDK_INC_PATH` or `TARGET_INC_PATH` in the C/C++ compiler include directory
  - Detection of _IA64_ or WIN64 in the C/C++ compiler definition to qualify as a 64-bit project
  - Detection of either entry point **DriverEntry**, or a linked library path to the DriverWorks or DriverNetworks directory

If a project does not meet one of the previous tests, it is not considered a driver project and will continue to use the default Visual Studio compiler. If the Windows DDK compiler is used, the compiler log will indicate this as "Compiling with <BASEDIR> DDK compiler..."

- Enable for all C/C++ projects.
  Select **Enable for all C/C++ projects** to use the Windows DDK compiler for all projects. For example, this selection could be used to build a test application with the DDK compiler.

Building Drivers Using Visual Studio and the BUILD Utility

Microsoft recommends that drivers be built with the BUILD.EXE utility provided in the DDK. DriverStudio supplies Visual Studio integration, which enables invoking the DDK BUILD utility from within Visual Studio. The advantage of doing this is that the Visual Studio jump to error feature is enabled. If any compiler errors occur, you can double-click on the error in the output window, and the source line will be displayed in the editor.

Invoking the BUILD utility is done with the DriverStudio **Build with BUILD.EXE** option located in the Visual Studio IDE. Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu, or from the DriverStudio DDK Build toolbar. Using Visual Studio 6, this is accessible from the DriverStudio menu or toolbar.

Complete the following steps to build your driver with the **Build with BUILD.EXE** option.

1. Set the DDK directory.
   You can set this by changing the value of **BASEDIR** in the **Driver Build Settings** dialog available from the DriverStudio menu.
2. Set the active configuration in the Build menu.
   The DDK build uses the current configuration to decide whether to run a checked or free build. It also runs BUILD.EXE in the directory where the active project file resides.
3. Start the build by selecting **Build with BUILD.EXE** from the DriverStudio menu.

DriverStudio runs BUILD.EXE in the directory where the active project file (.dsp or .vcproj) is located. Using the **Build with BUILD.EXE** button is equivalent to doing a **Rebuild All** with the DDK build system. The build output will be written to the build pane in the Visual Studio IDE. Just like running a regular build, double-clicking on an error

message will bring up the file with the cursor on the line where the error occurred.

The **Build with BUILD.EXE** option runs `BUILD.EXE` on the `SOURCES` file in your project file directory. The `SOURCES` file and the accompanying `MAKEFILE` must be present in the project directory for the button to be enabled. Make sure the `SOURCES` file is up to date with your project file before trying a DDK build.

Note: If your project file has multiple checked or free configurations, all configurations will be treated identically by the DDK build option. The **Build with BUILD.EXE** button always acts on the `SOURCES` file in the directory with the project file.

Building Drivers Using Visual Studio Build Commands

DriverStudio samples and wizard generated drivers can use Visual Studio project files instead of sources files for building drivers. Also, the **Sources to .dsp Converter** utility can create Visual Studio project files to build drivers. Microsoft recommends using the DDK BUILD system and a `SOURCES` file to build drivers. We recommend doing this for production versions of your driver. During active development, it is usually easier to work with Visual Studio and its user interface to adjust project settings and perform builds.

For Windows DDKs that require using the DDK compiler and linker, the DriverStudio integration will allow you to switch the default compiler and linker to the DDK compiler and linker. You can configure this option using the **DDK Build Settings** from within Visual Studio.

Complete the following steps to build drivers from within Visual Studio.

1. Configure the device driver build environment.
   Refer to “DDK Build Settings” on page 18
2. Open the Visual Studio project file.
3. Select the appropriate configuration from **BUILD > Configurations** to switch between CHECKED and FREE builds.
4. Select either **Build** or **Rebuild All** to build the newly active configuration.

Building DriverStudio Drivers

The DriverWorks and DriverNetwork libraries do not ship pre-built. This is to ensure that the libraries are built with the same DDK as your driver. Prior to building any DriverStudio sample or wizard driver, you must build the DriverStudio libraries on which the driver is dependent.

Refer to “DriverWorks Library Configurations” on page 24 for relevant library configurations for specific drivers, DDKs, and platforms.

There are three methods to build the libraries:

- Using the Microsoft Visual C++ build commands from within Microsoft Visual Studio
  This is the most convenient method, but it does not use the BUILD utility supplied by Microsoft. We recommend that you use this method while you are actively developing your driver, but that you use the BUILD program to build the final production release of your driver.
- Using the DriverStudio **Build with BUILD.EXE** command from within Microsoft Visual Studio
  This is a convenient way to use the DDK BUILD utility in conjunction with the `SOURCES` file and `MAKEFILE` from within Visual Studio.
- Using command line build scripts with the `SOURCES` files and the **DDK BUILD** utility

Microsoft Visual C++ Build Commands within Visual Studio

Complete the following steps to build the libraries using the Microsoft Visual C++ build commands from within Visual Studio.

1. Select a DDK using the **DDK Build Settings** dialog within the Microsoft Visual Studio IDE.
   Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar. Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.
2. Open the DriverWorks library Visual Studio project file.
   This file is located in the "source" subfolder of the DriverWorks root installation folder. For Visual Studio .NET, the project file is `vdwlibs.sln`. For Visual Studio 6, the project file is `vdwlibs.dsw`. For a typical installation, this would be:
   ```
   C:\Program Files\Compuware\DriverStudio\DriverWorks\Source\vdwlibs.dsw
   ```
3. Select **Batch Build** from the Visual Studio **Build** menu.

4 Click **Select All i386** for 32-bit builds.
This will select all configurations for the i386 platform.

For 64-bit builds, click **Select All ia64**.
This will select all configurations for the ia64 platform. The Windows XP (or higher) DDK must be used. If the Windows XP DDK is not the active DDK, these configurations will be ignored.

5 Select **Rebuild All**.

DriverStudio 'Build with BUILD.EXE' within Visual Studio

This method will invoke the DDK BUILD utility, used in conjunction with the `SOURCES` file and `MAKEFILE` contained in the project root folder. This is the recommended way to build the libraries for Windows XP DDK.

1 Select a DDK using the **DDK Build Settings** dialog within the Microsoft Visual Studio IDE.
Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar.
Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.

2 Open the DriverWorks library Visual Studio project file.
This file is located in the "source" subfolder of the DriverWorks root installation folder. For Visual Studio .NET, the project file is `vdwlibs.sln`. For Visual Studio 6, the project file is `vdwlibs.dsw`. For a typical installation, this would be:

```
C:\Program Files\Compuware\DriverStudio\DriverWorks\Source\vdwlibs.dsw
```

3 Select the Project Configuration to Build.
On the Visual Studio **Build** menu, select **Set Active Configuration**.

4 Select a 64-bit configuration (**IA64**) for 64-bit build environments, or a 32-bit configuration for 32-bit build environments.

5 Select **Build with BUILD.EXE**.
Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar.
Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.

6 Repeat Steps 3 and 4 for all 32-bit or 64-bit configurations.
Refer to "DriverWorks Library Configurations Table" on page 24 for relevant library configurations for specific drivers, DDKs, and platforms.

Command Line Build Scripts

DriverWorks ships with batch files for building the libraries from the command line. The batch file `bldlib.bat` is used to automatically set up the correct DDK build environment, set up the DriverWorks build environment, and build the library. The correct usage is:

```
bldlib [checked or free] [target] [64]
```

where target can be either `WDM`, `NT`, or `NDIS`. Including the `64` flag will build the 64-bit library (Windows XP or higher DDK only).

Complete the following steps to build the libraries using command line build scripts.

1. Launch a command shell.

Note: It is not necessary to use the DDK Build Environment shortcuts, however you must set your BASEDIR first.

2. Change directories to the DriverWorks root installation folder. For a typical installation, this would be:

   ```
   C:\Program Files\Compuware\DriverStudio\DriverWorks
   ```

3. Execute the batch file `bldlib.bat` with appropriate parameters. For example, to build the checked version of the DriverWorks WDM library, type:

   ```
   bldlib.bat checked WDM
   ```

4. Repeat Step 3 for other DriverWorks libraries as required. Refer to “DriverWorks Library Configurations” for relevant library configurations for specific drivers, DDKs, and platforms.

DriverWorks Library Configurations Table

The following table lists project configurations to be used in respective DDK environments for specific target drivers and platforms. The DriverWorks workspace `vdwlibs.dsw` contains project files `vdwlibs.dsp` and `ndiswdm.dsp`.

Table 2-1. DriverWorks Library Configurations

Configuration	DDK to Use	Library File	Target Drivers
VdwLibs - Win32 WDM Free	Windows 2000 DDK or Windows Me or XP DDK	`vdw_wdm.lib`	WDM-style drivers targeting Windows 2000, XP, 98, or Me
VdwLibs - Win32 WDM Checked			

Table 2-1. DriverWorks Library Configurations (Continued)

Configuration	DDK to Use	Library File	Target Drivers
VdwLibs - Win32 NT4 Free	NT4, Windows 2000, or XP DDK	`vdw.lib`	NT4-style drivers targeting NT4, Windows 2000, XP, 98, or Me
VdwLibs - Win32 NT4 Checked			
VdwLibs - IA64 WDM Free	Windows XP DDK	`vdw_wdm.lib`	WDM-style drivers targeting Windows XP for the IA64 platform (64-bit only)
VdwLibs - IA64 WDM Free			
NdisWdm - Win32 NDIS WDM Free	Windows 2000 or XP DDK	`kndiswdm.lib`	NDIS WDM drivers targeting Windows 2000, XP, 98, or Me
NdisWdm - Win32 NDIS WDM Checked			
NdisWdm - Win32 IA64 Free	Windows XP DDK	`kndiswdm.lib`	NDIS WDM drivers targeting Windows XP for the IA64 platform (64-bit only)
NdisWdm - Win32 IA64 Checked			

Building DriverNetworks Libraries

There are three methods to build the libraries:

- Using the Microsoft Visual C++ build commands from within Microsoft Visual Studio
  This is the most convenient method, but it does not use the BUILD utility supplied by Microsoft. We recommend that you use this method while you are actively developing your driver, but that you use the BUILD program to build the final production release of your driver.
- Using the DriverStudio **Build with BUILD.EXE** command from within Microsoft Visual Studio
  This is a convenient way to use the DDK BUILD utility in conjunction with the `SOURCES` file and `MAKEFILE` from within Visual Studio.
- Using command line build scripts with `SOURCES` files and the DDK BUILD utility

Refer to “DriverNetworks Library Configuration Table” on page 28 for relevant library configurations for specific drivers, DDKs, and platforms.

Microsoft Visual C++ Build Commands within Visual Studio

Complete the following steps to build the libraries using the Microsoft Visual C++ build commands from within Visual Studio.

1. Select a DDK using the **DDK Build Settings** dialog from within the Microsoft Visual Studio IDE.
   Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar. Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.
2. Open the DriverNetworks library Visual Studio project file.
   This file is located in the "source" subfolder of the DriverNetworks root installation folder. For Visual Studio .NET, the project file is `dnw.sln`. For Visual Studio 6, the project file is `DNW.dsw`. For a typical installation, this would be:
   `C:\Program Files\Compuware\DriverStudio\DriverNetworks\Source\DNW.dsw.`
3. Select **Batch Build** from the Visual Studio **Build** menu.
4. Check all boxes except the 64-bit configurations (IA64) for 32-bit builds.
5. Select all of the 32-bit configurations except **KNdisLib - Win32 NDIS 4 Miniport Free and Checked**. If you are using the NT4 DDK, select only **KNdisLib - Win32 NDIS 4 Miniport Free and Checked** configurations for building.
6. Click **Rebuild All**.

DriverStudio 'Build with BUILD.EXE' within Visual Studio

This method will invoke the DDK BUILD utility, used in conjunction with the `SOURCES` file and `MAKEFILE` contained in the project root folder.

1. Select a DDK using the **DDK Build Settings** dialog from the Microsoft Visual Studio IDE.
   Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar. Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.
2. Open the DriverNetworks library Visual Studio project file.
   This file is located in the "source" subfolder of the DriverNetworks root installation folder. For Visual Studio .NET, the project file is `dnw.sln`. For Visual Studio 6, the project file is `DNW.dsw`. For a typical installation, this would be:
   `C:\Program Files\Compuware\DriverStudio\DriverNetworks\Source\DNW.dsw.`

3 Select the Project Configuration to build.
On the Visual Studio **Build** menu, select **Set Active Configuration**.

4 Select a 64-bit configuration (**IA64**) for 64-bit build environments, or a 32-bit configuration for 32-bit build environments.

5 Select **Build with BUILD.EXE**.
Using Visual Studio .NET, this is accessible from the **Tools > DriverStudio** menu or from the DriverStudio DDK Build toolbar. Using Visual Studio 6, this is accessible from the **DriverStudio** menu or toolbar.

6 Repeat Steps 3 and 4 for all 32-bit or 64-bit configurations.
Refer to "DriverNetworks Library Configuration Table" on page 28 for relevant library configurations for specific drivers, DDKs, and platforms.

Command Line Build Scripts

DriverNetworks ships with batch files for building the libraries from the command line. The batch file `buildndis.bat` is used to automatically set up the correct DDK build environment, set up the DriverNetworks build environment, and build the ndis libraries. The correct usage is:

```
buildndis [checked or free] [target] [IA64]
```

where target can be either `miniport4`, `miniport5`, or `protocol5`. Including the `IA64` flag will build the 64-bit version of the library (Windows XP DDK only).

The batch file `buildtdi.bat` is used to automatically set up the correct DDK build environment, set up the DriverNetworks build environment, and build the tdi client libraries. The correct usage is:

```
buildtdi [checked or free] [target] [IA64]
```

where target can be either `NT`, `9xVxd`, or `9xSYS`. Including the `IA64` flag will build the 64-bit version of the library (Windows XP DDK only).

Complete the following steps to build the libraries using the command line build scripts:

1 Launch a command shell using the DDK Build Environment shortcuts.

2 Change directories to the DriverNetworks root installation folder. For a typical installation, this would be:

```
C:\Program Files\Compuware\DriverStudio\DriverNetworks
```

3 Execute the batch file `buildndis.bat` with appropriate parameters. For example:

```
buildndis checked miniport5
```

This builds the checked version of the DriverNetworks NDIS 5 Miniport library.

4 Repeat Step 3 for other DriverNetworks libraries, as required. Refer to "DriverNetworks Library Configurations" for relevant library configurations for specific drivers, DDKs, and platforms.

DriverNetworks Library Configuration Table

The `DNW.dsw` workspace file is comprised of two projects for 32-bit targets:

- `KNdisLib.dsp` (for NDIS classes)
- `Tdiclient.dsp` (for DriverSockets classes)

and 2 projects for 64-bit (IA64) targets

- `KNdis64.dsp` (for NDIS classes)
- `Tdi64.dsp` (for DriverSockets classes)

Each of the projects, in turn, include multiple configurations for different NDIS versions, checked vs. free, and target platform builds. The following DriverNetworks Library Configurations table lists project configurations to be used in respective DDK environments.

Table 2-2. DriverNetworks Library Configurations

Configuration	DDK to Use	Library File	Target Drivers
KNdisLib – Win32 NDIS 4 Miniport Free	NT 4 DDK	`kndis4mp.lib`	NDIS 4 Miniports
KNdisLib – Win32 NDIS 4 Miniport Checked			
KNdisLib – Win32 NDIS 5 Miniport Free	Windows 2000 or XP DDK	`kndis5mp.lib`	NDIS 5 Miniports & IM Drivers
KNdisLib – Win32 NDIS 5 Miniport Checked			
KNdisLib – Win32 NDIS 5 Protocol Free		`kndis5pt.lib`	NT 4, Win2K TDI Clients; Protocol Drivers
KNdisLib – Win32 NDIS 5 Protocol Checked			

Table 2-2. DriverNetworks Library Configurations (Continued)

Configuration	DDK to Use	Library File	Target Drivers
KNdis64 – Win32 NDIS 5 Miniport Free	Windows XP DDK	`kndis5mp.lib`	NDIS 5 Miniports & IM Drivers for XP IA64
KNdis64 – Win32 NDIS 5 Miniport Checked			
KNdis64 – Win32 NDIS 5 Protocol Free		`kndis5pt.lib`	TDI Clients; Protocols for XP IA64
KNdis64 – Win32 NDIS 5 Protocol Checked			
Tdiclient – Win32 Debug NT 5	Windows 2000 or XP DDK	`tdint5.lib`	NT 4, Win2K TDI Clients
Tdiclient – Win32 Release NT 5			
Tdi64 – Win32 Debug NT 5	Windows XP DDK	`tdint5.lib`	Windows XP IA64 TDI Clients
Tdi64 – Win32 Release NT 5			
Tdiclient – Win32 Debug W9X SYS	Windows 2000 VtoolsD	`tdiw9sys.lib`	Win9X TDI Clients for NDIS Miniports & WDM Drivers
Tdiclient – Win32 Release W9X SYS			
Tdiclient – Win32 Debug W9X VxD	Windows 2000 VtoolsD	`tdiw9vxd.lib`	Win9X TDI Clients for VxDs (VtoolsD)
Tdiclient – Win32 Release W9X VxD			

Using DriverWizard

For many types of device drivers, including drivers for most USB, IEEE 1394, and PCI boards, DriverWizard will generate source code tailored to your hardware. You can launch DriverWizard from within Microsoft Visual Studio via the **DriverStudio** menu or toolbar, or from the **Start** menu.

DriverWizard leads you through the definition of a driver with a series of dialogs, or “steps.” You can click **Finish** during any step to bypass the rest of the steps and create a driver based on the selections you have already made. However, we recommend you view each dialog step to determine whether there are additional choices that should be made for your driver.

DriverWizard is used to create the initial framework of your driver. While DriverWizard handles quite a bit of routine initialization and setup, you will always need to add additional code to handle the specifics of your device. DriverWizard inserts comments with a “TODO” tag indicating areas that you need to review.

DriverWizard cannot be used to modify the driver after the framework is defined. Of course, you can always generate a new driver using DriverWizard.

If you need to add code to an existing DriverWizard generated driver, it might be useful to use DriverWizard to generate a small test driver that contains just the additional handler or initialize that you need. Use a text editor to cut and paste a code fragment into your driver. Please note that the integrated debugging environment of MSVC is not able to debug device drivers. You must use a system-level kernel-mode debugger such as the Microsoft WINDBG or Compuware SoftICE to debug your driver. DriverWorks includes a convenient program, DriverMonitor™, you can use to load and start the driver and to view messages sent from your driver without requiring you to load a kernel-mode debugger.

Understanding DriverWizard

The best way to understand the capabilities and limitations of the DriverWizard is to build several test drivers and review the generated source code. While running DriverWizard, you can press **F1** at any time to get additional information about the dialog box being displayed. In order to understand the content of the various screens, a basic understanding of the NT/WDM device driver architecture is required.

Begin a New Device Driver Project

The following screen capture shows the first step in creating a new device driver project.

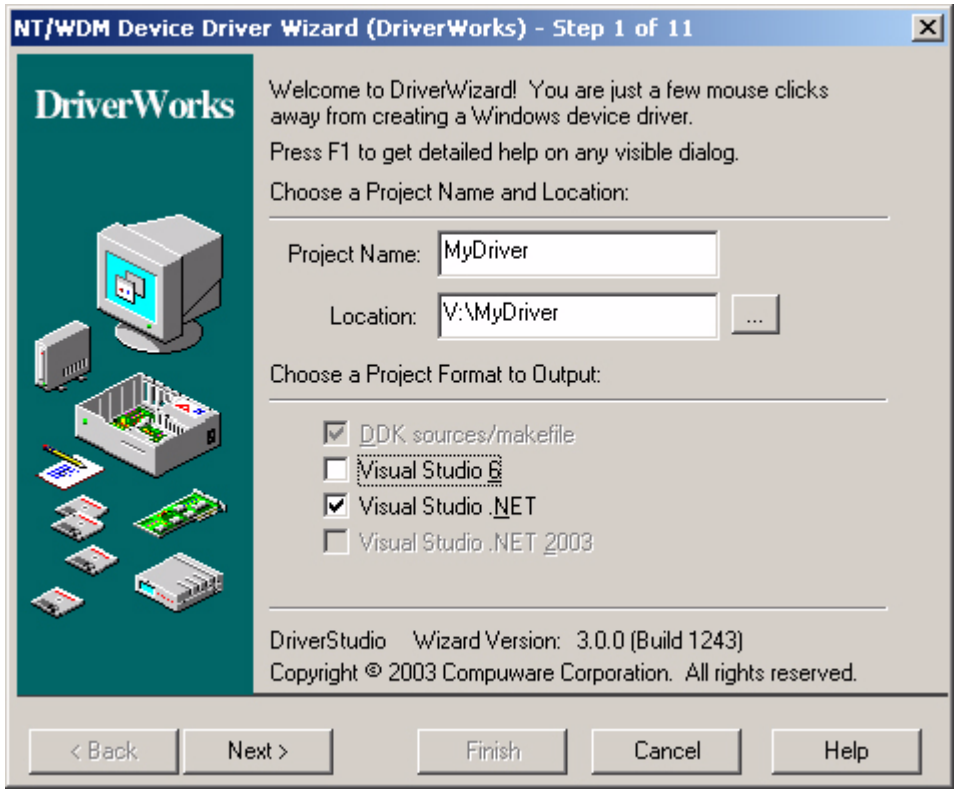


Figure 2-2. Creating a New Driver with DriverWorks DriverWizard

1. From within Visual Studio, on the DriverStudio menu, select **DriverWizard**. You can also access the wizard from the **Start** menu.
2. Then choose a name and location for your project.
   The name that you select will be used as the default base name for the files and classes that DriverWizard generates for you, although you can override the default in later dialogs.
3. Select the appropriate options at each dialog step to define the parameters of your device driver.
4. Click **Finish** and **OK** to confirm the creation of your new driver.
   The device driver generated by DriverWizard should compile and link without errors.
5. On the **BUILD** menu, select **BUILD driver.SYS** to build the driver.
   You must have the correct version of the DDK installed and you must

have already built the DriverWorks libraries. Otherwise, compile and/or link errors will be generated.

6 To complete your driver, search for the string “TODO:” and add any necessary code specific to your hardware.

Note: Refer to the `README.TXT` file created by the Wizard for a list of files and registry entries generated for your driver.

Chapter 3
DriverWorks Object Model

- The Driver Object
- Device Objects
- Lower Device Objects
- Objects Used During Initialization
- Queue Objects and Request Serialization
- Driver Managed Queues
- Controller Objects
- Interrupt Request Levels (IRQL)
- Objects for Controlling Hardware
- Containers and Other Objects
- Chart of DriverWorks Classes

This chapter provides an overview of the DriverWorks object model. Read this chapter to gain an understanding of the classes that comprise DriverWorks and how they relate to each other. Each section of the chapter discusses the architecture abstractly and includes references to the related DriverWorks class.

The DriverWorks object model reflects the architecture of Windows NT and the Windows Driver Model. It amplifies the object orientation of the operating system by providing classes built on system services. The goal of DriverWorks is to provide objects that closely model the underlying system while at the same time provide powerful methods to simplify and clarify the task of developing kernel mode drivers.

The Driver Object

***The Driver Object is implemented by class* KDriver.**

The I/O Manager uses driver objects to create a structure for managing devices. Each driver object is responsible for managing one or more device objects it creates. An I/O request made by a user subsystem causes a request to be sent to a particular device object. The I/O Manager, a kernel component, determines which driver object is responsible for the targeted device and passes the request object to a method exported by that driver. Driver objects have additional methods for initialization and shutdown.

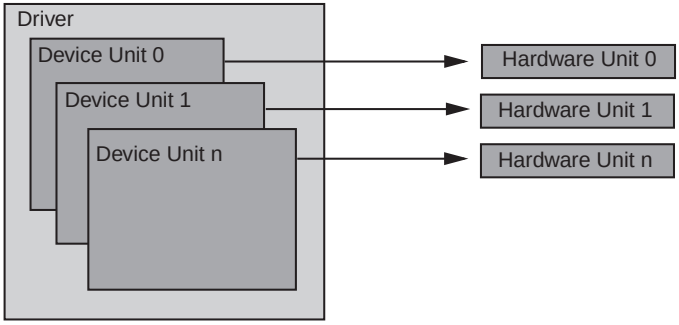


Figure 3-1. **Driver with Multiple Devices**

Initialization

The driver writer derives a class from KDriver to form the basic framework of a driver. The system creates a driver object for each driver that is loaded. The driver subclass must supply an initialization entry point that the system calls. This entry point is always named DriverEntry.

The initialization phase might have several steps. In a typical non-WDM case, the driver first queries for the presence of devices it must control. In order to support those devices, the driver might require system resources such as interrupts, DMA channels, or address ranges. Depending on the driver's design, these resources might be claimed by the driver object, or each device object created by the driver might be responsible for claiming the resources it needs.

After probing the system for configuration and resource information, a driver creates device objects that correspond to actual devices under its control. Device objects are typically created during initialization, although this is not required.

In a WDM driver, the scope of the DriverEntry routine is very limited. It can query the registry and initialize some variables, but because the burden of resource management is on the operating system under WDM, the driver is not obligated to request resources.

A WDM driver object exports an entry point the system calls when a supported device is detected. This routine is named AddDevice and is responsible for creating device objects to model the physical devices the driver controls.

Reinitialization

Besides DriverEntry, a non-WDM driver object can export an additional initialization method to be called at a later stage of the system's initialization process. Drivers do this in order to reduce sensitivity to load order. A driver might depend on the presence of other drivers and might require those drivers initialize first. To some degree, a driver can control this by means of installation parameters stored in the registry, but this is not always sufficient.

The driver object exports a method that DriverEntry calls to set up a callback to a reinitialization method. After the system makes a first pass of initialization for all drivers, it calls any reinitialization methods drivers have registered. The reinitialization method can request additional callbacks.

Dispatching Requests

Given an I/O request object intended for a particular device object, the I/O Manager delivers the request to the driver object that created the targeted device object. The default action of the driver object is to route the request to the targeted device as quickly as possible. A driver subclass can override this default behavior. For example, the driver object might need to compile statistics of how many requests have been processed. To do so, it overrides a dispatch filter method that can view all I/O requests coming into the driver. The driver object's dispatch filter can examine or process the request before choosing to pass it on the targeted device object. This mechanism is not to be confused with the concept of a *filter driver*, which can filter requests targeted to devices under the control of other drivers.

Unloading

A well-designed driver indicates to the system that it can be unloaded when it is no longer needed. To do so, it exports an unload method. Certain kinds of drivers that are critical to the operation of the computer, such as the keyboard or display, do not export unload methods because they are always needed.

A driver object is responsible for destroying any objects it has created when the system calls its unload entry point, including all device objects. Invoking the device destructors should, in turn, cause device objects to destroy any objects they have created. If a driver claimed system resources with a resource request during initialization, its unload method should release them.

Image Section Objects

***Image Section Objects are implemented by class* KImageSection**.

Image section objects correspond to sections of the driver's executable image, each of which has characteristics that are defined when the driver is linked. Sections that have names begining with "INIT" are discarded by the system after initialization. Sections that have names begining with "PAGE" are capable of being paged out to disk when not in use. Other sections remain resident in memory and are said to be locked, or nonpaged.

Because physical memory is a critical shared resource, a driver should minimize the total size of its nonpaged sections. By making a section pageable, the driver enables the memory manager to allocate the physical memory where it is most needed.

At run time, a driver can control which sections can be paged out and which cannot. After making all sections pageable, a driver creates an image section object for each section of code that is to be locked. An image section object created in this fashion exports methods enabling it to be dynamically locked and unlocked.

I/O Request Objects

***I/O Request Objects are implemented by class* KIrp.**

I/O request objects, known more commonly as IRPs (I/O Request Packets), drive the operation of most kernel mode drivers. When applications send data to or request data from a device, the I/O manager creates an IRP and delivers it to the driver responsible for the device. The driver dispatches the request to one of its device objects. The device might begin processing the IRP, or it might place it on a queue for processing later. The action taken depends on the nature of the request and the state of the device. Later, when the request is satisfied, a device sends the IRP back to the I/O Manager.

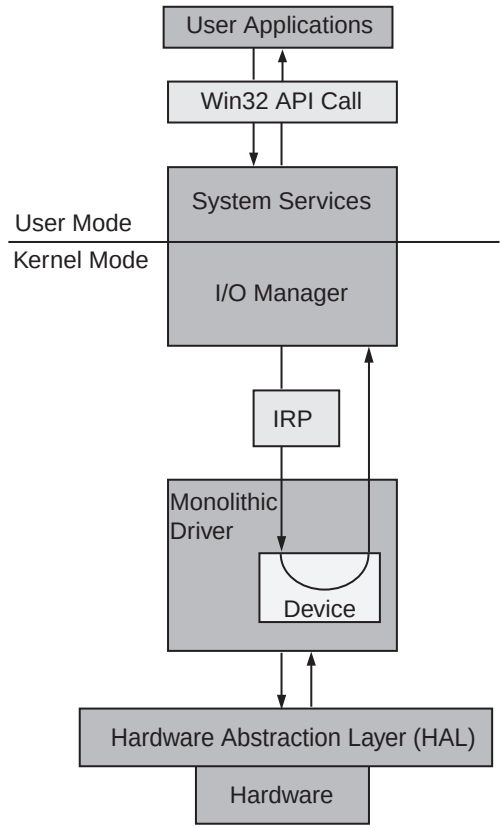


Figure 3-2. **Monolithic Driver**

Each IRP describes an I/O operation to be performed by a particular device. Accessor methods retrieve attributes of the request, such as the major and minor function codes, the status, and the parameters to various operations, such as offsets and counts for read and write requests.

In general, an IRP might require action on its behalf by multiple device objects in order to satisfy it. For each device object that might share in its processing, the IRP reserves a data area where parameters and other information can be stored. Each of these data areas is known as an "IRP stack location." When a device object processes a request, it can access only the IRP stack location reserved for it and one for that of a device to which it passes the IRP. A device object that receives an IRP can set up the next IRP stack location and pass it to another device object for further processing.

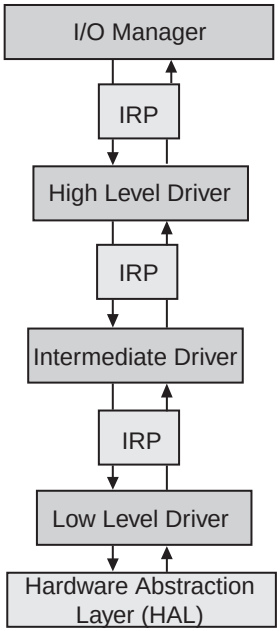


Figure 3-3. **IRP Flow Through Layers of Drivers**

While the I/O Manager creates most IRPs in response to application calls, drivers commonly create them as well. The most common IRPs are requests for Create, Close, Read, Write, or DeviceControl processing. A device object might need to break up a large request into smaller pieces that other device objects closer to the hardware can handle. To do so, the device object creates multiple IRPs and sends them to the lower devices. When they have all completed, the device object deletes the IRPs it created and completes the original request.

IRPs can also be canceled when an I/O operation needs to be aborted, possibly because the originating application has been terminated. The IRP object contains a pointer to a cancel routine that will be called when the IRP is canceled.

Device Objects

***Device Objects are implemented by classes* KDevice *and* KPnpDevice**.

Because they provide a representation of the I/O capabilities available on the computer, device objects are central to the architecture of the I/O subsystem. Device objects comprise the set of logical I/O targets with which applications can exchange information. The system supplies APIs (such as CreateFile, ReadFile, and WriteFile) that applications use to send requests to device objects indirectly via the I/O Manager. Although the I/O Manager delivers the request first to the driver responsible for the device, it is the device object that does the bulk of the processing. Device objects are collectively responsible for providing a set of abstract services (e.g. read, write, and control functions) that provide access to the available hardware.

Note: Throughout this document, references to “device” mean “device object,” except when the term “device” might be construed to mean “physical device.”

Most devices are named, although anonymous devices are allowed. A device can communicate with a second device if the name of the second device is known.

In order for an application to communicate with a device, the device must be referenced by at least one *symbolic link*. To understand what a symbolic link is, understand that the system maintains a directory of objects of varying types, such as drivers, devices, events, and semaphores. A symbolic link is a special type of object that provides an indirect reference to another object in the system. A special subdirectory of the object hierarchy is reserved for symbolic links to device objects. Symbolic links in this subdirectory are visible to applications via Win32 APIs. For a given device, there can be multiple symbolic links that point to it, or there can be none at all. Each link is either protected or unprotected. If a link is unprotected, user mode code can change the device object the link references.

Every kernel mode driver implements one or more subclasses of the system device object. For each subclass, the driver writer determines the behavior of the class by overriding virtual methods, including those methods that process the various types of IRPs.

In order for a device object to fulfill its IRP processing responsibility, it must have the ability to control the hardware it represents to the system. In order to maintain compatibility across multiple hardware architectures, all low-level access to hardware is performed through a set of hardware objects that are in turn defined in terms of the system Hardware Abstraction Layer (HAL). These hardware objects, such as I/O port objects, interrupt objects, and DMA objects, are described later.

The device object is primarily an IRP target. The I/O Manager delivers requests to the device's driver object and the driver object automatically dispatches them to the device(s). The system does not attempt to abstract any particular capabilities of physical devices. That is the job of subclasses of the system device object that are implemented in kernel mode drivers.

Another way to think of it is to consider a device as having an upper edge and a lower edge. Class KDevice and subclasses thereof define the upper edge of the device. In general, this is an IRP based interface, although that is not always the case. A subclass of KDevice can implement processing for a set of IRPs specific to a class of devices or it can implement an upper edge that has an entirely different interface.

The lower edge of a device is either a direct interface to the hardware or an interface to another device. Direct control of hardware is done with the aid of the HAL. Interfacing to lower devices employs a "lower device" object. A device object can have one or more embedded lower devices.

The important architectural consideration here is that the upper edge and lower edge are intentionally decoupled. Class KDevice provides the basis for defining the upper edge of a device, but has no bearing on the lower edge. The lower edge is determined by the objects that the driver writer embeds in the subclass of KDevice. The decoupling of the upper and lower edges enables the object model to flexibly support the layered nature of the system.

Lower Device Objects

***Lower Device Objects are implemented by classes* KLowerDevice, KPnpLowerDevice, KUsbLowerDevice, *and* K1394LowerDevice.**

Some device objects process IRPs without ever accessing hardware. Instead, they rely on the services of device objects under the control of other drivers to provide the interface to the hardware. From the system's point of view, all device objects are treated symmetrically. However, from the perspective of a given driver, device objects it creates are logically distinct from devices under the control of other drivers. In other words, the set of operations a driver performs with its own device objects is quite different from the set of operations performed with device objects belonging to other drivers. The DriverWorks object model therefore distinguishes owned device objects as a class distinct from unowned device objects, and the latter are referred to as "lower devices."

Some device objects use lower devices to carry out the processing of an I/O request. This characteristic of the system enables layering of drivers. It is common to speak of a *stack* of drivers, meaning a set of hierarchically related device objects, each with a well-defined interface. Each device object has its own role in the processing of an IRP. Upper level drivers implement an interface to applications and lower level drivers implement an interface to the hardware. For example, the keyboard class driver presents a generic keyboard interface to user subsystems. It relies on a keyboard port driver to provide the interface to the physical keyboard. The class driver is interoperable with a number of different port drivers, each of which supports a different kind of physical keyboard.

There are several benefits to this approach. A given device object is responsible for a limited set of functions. This enables applications to be written in a device independent manner and facilitates development of drivers for new hardware. The benefits of modularity are as important in driver development as in most other areas of software development.

In a WDM driver, the system bus drivers create device objects to model physical devices for purposes of configuration and power management. Drivers for these devices create their own device objects that correspond to physical device objects the bus drivers create. A special class in DriverWorks models the special relationship between the two device objects representing the same physical device.

For instance, the system has a USB bus driver and host controller for USB devices. WDM drivers for USB devices communicate to the system drivers to perform I/O to the USB device. The DriverWorks class KUsbLowerDevice models the interaction between the WDM driver and the USB system drivers. Also, the system has a 1394 bus driver and host controller for 1394 devices. WDM drivers for 1394 devices communicate to the system drivers to perform I/O to the 1394 device. The DriverWorks class K1394LowerDevice models the interaction between the WDM driver and the 1394 system drivers.

Filter Device Objects

***Filter Device Objects are implemented by classes* KWdmFilterDevice *and* KFilterDevice.**

Filter device objects are very similar to device objects. A filter device object is primarily an IRP target and the upper edge of the device. However, a filter device object is used to intercept IRPs targeted at a different device object. The system will route all I/O requests targeted to the filtered device to the filter device object. This gives the filter device object an opportunity to process the IRP before or after the filtered device object processes it. Usually, most IRPs are simply passed from the filter device object to the filtered device object. The filter device object often provides special processing for a small number of IRPs. In this way, the filter device object can provide value-added processing to the capabilities of the filtered device object.

In a WDM driver, DriverWorks class KWdmFilterDevice provides a default implementation which acts as a pass-through WDM filter. This means that all IRPs are sent to the filtered device object for processing. To add special processing for an IRP, the KWdmFilterDevice base class method is overridden. The base class takes care of all of the boilerplate details.

Objects Used During Initialization

The following sections describe objects used during initialization.

Driver Parameters and Registry Objects

***Interface to the registry is implemented by the class* KRegistryKey**.

The installation procedure for a driver and the devices it supports can store information about the devices in the system *registry*. The registry is a system-wide database. It is organized in a tree-like manner, much like a directory of files. Each node of the registry tree is called a *key*. The leaves of the tree are called *values*. Each value has a name and associated data. The data can be of several types, such as a simple integer, a string, or arbitrary binary data.

The system provides a registry key for each driver, the path to which is passed to DriverEntry. A driver sometimes reads parameters from subkeys of its registry key. DriverWorks provides a registry key object that enables the driver to conveniently access parameters in the registry.

A registry key is constructed by specifying its path. A registry key object exports methods to read and write values and to enumerate its values and its subkeys.

Configuration Query Objects

***Query objects are implemented by class* KConfigurationQuery.**

Before a non-WDM driver can construct device objects, the initialization method must determine how many devices that fall under the driver's control are present and what the specific characteristics of those devices are. The device query can be carried out in an ad hoc manner by the initialization code or can employ a query object. Query objects search for devices with particular characteristics and call a supplied entry point in the driver for each one that is found.

There are two kinds of query objects. The first type searches for devices the system can detect automatically, such as serial ports or disk controllers. The second type searches in the system registry under the key for the driver. The installation procedure sets up the registry keys in a predetermined hierarchy that includes a separate key for each device present. This enables the driver to dynamically determine what devices are present on a given system.

Both types of queries require the driver to supply a callback to be invoked for each qualifying device. For system detected devices, the parameters to the callback include the set of system resources the device requires. For devices configured in the registry, the parameters to the callback specify which unit of a particular device class was found. Typically, the callback will construct a new device object for each unit discovered. The callback or the device class constructor uses the registry key to access the registry and determine the resources that the unit requires.

In WDM drivers, the system delivers resource information to a device when the device is started.

Resource Request and Assignment Objects

The responsibility for claiming resources can be assigned to a non-WDM driver object or to its subordinate device objects. In either case, the process involves a resource request object and sometimes one or more resource assignment objects. WDM drivers do not request resources; instead, the system determines all resource assignments based on device requirements determined by the bus driver or stored in the registry.

***Resource request objects are implemented by class* KResourceRequest.**

A resource request is an object that describes the set of system resources a driver or device requires in order to operate successfully. It might be very simple and contain a single required item, such as a particular interrupt line (IRQ), or it might be a complex tree of alternative resource sets, each set having multiple resources of various kinds, and each resource allowing a range of values.

***Assignment objects are implemented by class* KResourceAssignment.**

In cases where the request does not include ranges or alternates, the request either succeeds or fails. If the system cannot grant the required resource, the driver aborts the initialization process. For the more complex requests that present choices to the system, a driver needs a resource assignment object to extract the resources that are granted by the system. Each resource assignment object corresponds to a particular item of the request. If a driver creates a request object with three distinct resources, it requires three assignment objects to determine which resources the system assigned. The resource request and assignment objects hide the complex data structures required for the raw interface to the underlying system calls.

Queue Objects and Request Serialization

The nature of a device determines whether it can process multiple requests simultaneously. For many kinds of hardware, the physical resources can only do one thing at a time. For example, it would not make sense for a simple parallel port to attempt to simultaneously transmit data for more than one I/O request, because the device is implemented as a single physical channel. As a result, many devices require a mechanism to assist in the serialization of requests, i.e., a means of ensuring that only one request can be processed at a time.

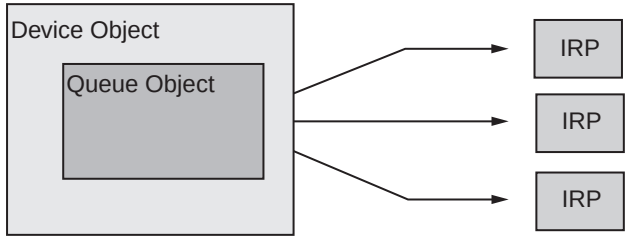


Figure 3-4. **Queue Object Embedded in Device Object**

***Queue objects are implemented by class* KDeviceQueue.**

The simple solution to this problem is a queue. The system device object embeds a queue object for IRPs pending service. A subclass of the device object can call methods that manipulate the embedded queue. The operations are more sophisticated than simply placing an IRP on the queue or removing one from the queue. Instead, there is a method whose semantics are: "If the device is busy processing a request, then place this request on the queue. Otherwise, pass the request to a function that can start processing it immediately." An additional method behaves as follows: "This request is complete, so send it back to the I/O Manager. If there are requests in the queue, remove the head and pass it to a function that can start processing it." These methods greatly simplify the task of serialization for the device subclass.

Driver Managed Queues

***Driver managed queue objects are implemented by class* KDriverManagedQueue.**

The queue object embedded in the system device object is sufficient to serialize a single set of IRPs, all of which require common resources of the device. However, in some cases a device object is capable of processing dissimilar requests independently. The classic example is a full duplex serial driver that can transmit and receive independently. It would be inefficient to serialize both transmit and receive requests in the same queue, but the system device object provides only a single queue.

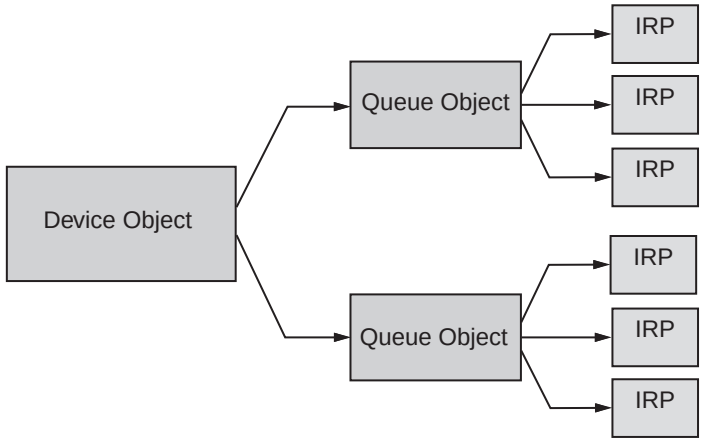


Figure 3-5. **Driver-Managed Queues**

To deal with situations such as this, the system allows device objects to create and manage additional queue objects. These objects have methods with the same functionality as described above for the queue embedded in the device object. The driver writer subclasses the queue object, overriding the virtual method the queue object invokes to start processing a request. When a device object receives an IRP, it calls the queuing method of the created queue object, rather than the queuing method of the system device object.

Controller Objects

***Controller objects are implemented by class* KController.**

Another kind of synchronization problem arises when a set of device objects belonging to the same driver all share a common resource. For example, a pair of disk drives on the same disk controller is abstracted as separate device objects, but they cannot operate independently from one another because there are shared data channels on the controller.

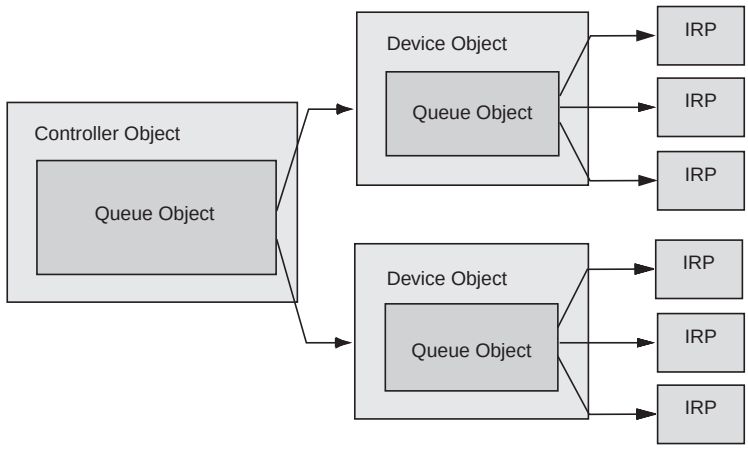


Figure 3-6. **Callbacks to Devices Queued in Controller**

The system provides controller objects to simplify queuing of requests in cases such as this. When a device object dequeues an IRP that requires the shared resource, it calls a method of the controller object to request access. If the resource represented by the controller is available, the controller object immediately invokes a function specified by the device (possibly a method of the device). If the resource is not available, then the controller object queues the address of the callback function for invocation when the resource is freed. Note the queue object embedded in a controller is not a queue of IRPs; it is a queue of function addresses passed to the controller by devices requesting access to the shared resource. When the controller calls one of these functions, it means access to the shared resource has been granted to the called device object. The device processes its current IRP, and then calls a controller method to release ownership of the shared resource.

Interrupt Request Levels (IRQL)

The design of the operating system recognizes that conditions can arise which require the service of a processor within a short interval. The obvious example is an interrupt from a device that needs to transfer data. If the system does not call the interrupt service routine promptly, data might be lost.

To handle the competing demands of the various processes and devices that might be running on a computer, the system implements the concept of *interrupt request levels*, commonly referred to as IRQL. All processing occurs at one of 32 IRQLs, numbered from 0 to 31, with 31 being the highest priority. IRQLs are not to be confused with the scheduler's process priority levels, which are gradations within the lowest IRQL, that is, IRQL 0 or Passive Level.

Highest = 31	Bus Error
30	Power Failure
29	Interprocessor Communication
28	System Timers
3-27	I/O Device Interrupts (DIRQL)
2	Dispatch Level
1	APC Level
Lowest = 0	Passive Level

The highest priorities are reserved for catastrophic events such as bus errors and power failures. Below that is the IRQL for interprocessor communication, followed by the IRQL for system timers. Next are the IRQLs for I/O devices, known also as device IRQLs or simply DIRQLs. The three lowest IRQLs in descending order are named DISPATCH_LEVEL, APC_LEVEL, and PASSIVE_LEVEL. These are associated with software events and will be discussed further below.

At any given time, each processor is running at a particular IRQL, depending on the kind of processing it is doing. IRQLs of the processors in a multiprocessor system are independent from one another. The kernel will interrupt a processor's execution only if an event that requires an IRQL higher than that at which the processor is currently running occurs. This means all code in a kernel mode driver is interruptible, but only by events of higher priority.

It is important for kernel mode drivers to minimize the time spent at raised IRQLs, especially at device IRQLs. Running at DIRQLs for longer than necessary can interfere with the operation of other devices by delaying the execution of their interrupt service routines. The system provides mechanisms that aid in writing drivers that are well behaved in this regard.

Many methods of system objects are not callable above DISPATCH_LEVEL, and some are only callable at PASSIVE_LEVEL. Others specify a minimum IRQL, rather than a maximum. Furthermore, code that runs at DISPATCH_LEVEL or above must not access paged code or data.

Objects for Controlling Hardware

The object model provides objects that assist in the control of hardware devices. A kernel mode driver creates these objects to represent the physical attributes of the devices it supports. These objects include ports and memory ranges, interrupts, and objects that support DMA.

Peripheral Address Objects

In order to understand the abstraction of peripheral bus addresses, it is necessary to understand how the system views physical addresses. In the simplest machine model, there is a single address space common to both the processor and programs. That is, the address that a program accesses in software is the same as the address output on the processor's address lines. In a more sophisticated model, there are two address spaces, namely physical and virtual. In this model, the address a program uses is different than the address represented on the processor's address lines, as a result of a translation performed to make discontiguous physical memory appear as contiguous to software.

Windows NT extends this concept further. In modern PC designs, there are multiple buses in the system. Typically, the CPU, external cache, and main memory reside on a high-speed bus whose architecture is specific to the processor. The CPU bus is bridged to a standardized local bus on which reside high-speed devices such as the display. The local bus is bridged to secondary local buses and to external buses that host storage devices, network adapters, and communications ports. It is the multiple bus architecture of modern PCs that gives rise to a more complex address space model.

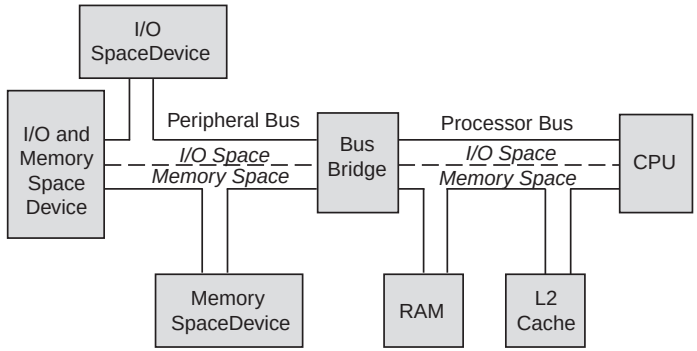


Figure 3-7. **Multiple Bus Architecture**

In order to access devices residing on buses other than the CPU bus, the system hardware must create a mapping between the CPU bus address and the corresponding address on the bus where the target device resides. Therefore, the address the processor puts on its address lines and the address the target device is configured to decode might be different.

Some peripheral buses supply separate address spaces for I/O and memory operations. Similarly, some processors (such as Intel x86) recognize this distinction. The mapping the system makes from a peripheral bus to the processor bus must consider this. An address in the memory space of a peripheral bus always maps to a memory address on the CPU bus. However, an I/O address on the peripheral bus does not always map to an I/O address on the CPU bus. This is obviously the case if the CPU does not recognize an I/O space.

The system does not require the device driver writer to know the details of how each bus in the system is mapped to the processor's address space, but it does require taking into account that such a mapping can occur. For example, suppose it is learned from the registry that device *D* is on bus *B* of bus type *I*, and responds to I/O address *A* on that bus. In order to access that device, the driver must first request the system to map that device address information to a physical address on the CPU bus. The result could be in either the memory space or the I/O space on the CPU bus, depending on how the peripheral bus is hosted. If it is in memory space, a driver that performs programmed I/O to that device must make an additional request to the system in order to set up page tables to access that physical address.

***Peripheral address objects are implemented by the following classes:* KPeripheralAddress, KIoRange, KMemoryRange, KIoRegister, *and* KMemoryRegister.**

In the DriverWorks object model, peripheral address objects hide the operations that configure the system to support a particular range of ports or memory registers of a physical device. To create a bus address object for a particular range of addresses, the following are required.

- Bus type and instance number of that type
- Peripheral address space to be accessed, either memory or I/O
- Start address as decoded by a device on that bus (this can be a 64-bit value)
- Size of the address range

Subclasses of the address object export methods for reading from and writing to the locations specified during construction. These methods use system services appropriate for the address space of the peripheral bus mapped by the base class. These platform dependent services ensure caches are properly flushed.

This document uses “register” to refer to a location on the device. A register can be in the I/O space or the memory space of the peripheral bus. Sometimes, “register” is qualified as being a memory register or an I/O register. In both cases, the address space of the peripheral bus on which the device resides determines what kind of a register it is. A register of a device in the I/O space of its peripheral bus is still an I/O register, even if it is mapped to the memory space of the CPU bus. In the Microsoft DDK, the word “port” is sometimes used to mean “register.” We use the more neutral “register” to avoid confusion and ambiguity.

Interrupt Objects and Deferred Procedure Calls

***Interrupt objects are implemented by class* KInterrupt. *Deferred procedure call objects are implemented by class* KDeferredCall.**

The system abstracts interrupts from I/O devices in a manner that can be implemented in a platform-independent fashion. The attributes of an interrupt object include the type and instance of the bus on which the interrupt is asserted, the priority level on that bus, the bus vector number, and the mode, either latched or level sensitive. On some buses, such as ISA, bus priority level and bus vector are mapped to the same value. It is possible for more than one interrupt object to have the same attributes, as long as the drivers that create those objects specify that the interrupt is sharable. The system associates a specific IRQL with each interrupt object, depending on the bus and bus level of the interrupt. A driver that creates multiple interrupt objects can arrange for all of them to be associated with the same IRQL.

The interrupt object exports methods to connect and disconnect an interrupt service routine, or ISR. Once an interrupt object is connected, the system invokes the ISR whenever the interrupt occurs. The object model allows the ISR to be a standalone function or a method of any class defined or derived by the driver developer.

The ISR cannot run until a processor is available at an IRQL with a priority less than the IRQL associated with the corresponding interrupt object. Similarly, the ISR cannot be interrupted unless an interrupt of higher priority occurs.

Because it runs at device IRQL, an ISR must execute as quickly as possible. Running at DIRQL for too long can interfere with the operation of other devices by blocking out interrupts. If an interrupting device requires significant processing time to service the interrupt, then as much as possible of that processing should be executed below device IRQL. A second reason for deferring to a lower IRQL is that the processing of the interrupt might require system services that are not callable at DIRQL.

The system provides Deferred Procedure Call objects (DPCs) to enable drivers to continue the processing of an interrupt at a lower IRQL, namely DISPATCH_LEVEL. The construction of a DPC requires the address of a callback function to be invoked by the system. If an ISR requires the use of a DPC, the driver must construct the DPC prior to connecting the interrupt object.

The operation of a typical ISR follows a simple format. It first queries its hardware to verify that the device is actually requesting service. It then executes code to acknowledge the interrupt and put the interrupting device in a known state. Next, it queues a previously initialized DPC object, and returns. The kernel services the DPC queue whenever the IRQL of some processor is below DISPATCH_LEVEL. This means that on a system with a single CPU, the DPC cannot execute until sometime after the ISR returns. On a multiprocessor system, it can execute in parallel.

If the callback function of a DPC needs to access variables modified by the ISR, then the callback function must take measures to ensure its access to those variables is atomic. If the callback function failed to do this, it would risk using data that was temporally inconsistent. As a simple example, suppose the DPC callback function reads the value of some variable, increments it, and stores it back to memory. If the ISR modifies the variable after it has been read but before it is written, then the variable has been corrupted. This is an issue not only for DPC callback functions, but for any code in the driver that executes at a lower IRQL than that associated with the interrupt object.

To deal with the issue of atomic data access, the interrupt object exports a method for synchronization. This method takes the address of a function as a parameter, and synchronously calls that function in a way that prevents the associated ISR from executing (on any processor) while the function is executing. The function called by the synchronization runs at the IRQL associated with the interrupt object.

Objects for DMA

Direct Memory Access (DMA) is a transfer of data between system memory and a device, distinguished by the fact that the bus cycles that cause the transfer are not generated by a CPU. Instead, the bus cycles are generated either by the device itself or by a DMA controller on the system board.

Devices that themselves generate the bus cycles for a transfer are referred to as *bus masters*, while devices that rely on the system DMA controller are referred to as *slaved devices*. Transfer to and from slaved devices is also referred to as *system DMA* transfers.

The memory region that acts as the source or target of a DMA transfer is normally physically contiguous. This is necessary because the system DMA controller and some bus masters can only be programmed with a single physical address. Some bus masters can be programmed with multiple physical addresses, a capability known as supporting *scatter/gather*. A bus master that supports scatter/gather does not require the memory region for a DMA transfer to be physically contiguous.

DMA Adapter Objects

***DMA adapter objects are implemented by class* KDmaAdapter.**

A DMA adapter object represents the physical resources a device requires for DMA transfers. The object model uses the same object for both bus master and slaved DMA devices, and this distinction is an attribute of the object specified to its constructor. For both cases, the object is created for a device on a particular bus type and bus number. For bus master devices, support for scatter/gather is indicated by a variable in the adapter object. For slaved devices, the system DMA channel on which the device transfers data is an adapter attribute.

DMA adapter objects resemble controller objects in that they export a method for serialization of access. When a device object needs to transfer data, it calls this method, supplying the address of a function to invoke when the object is unowned. If the adapter is unowned at the time of the call, it invokes the supplied function immediately. If the adapter is busy, the function address is placed on a callback queue embedded in the adapter object. When another device object releases the adapter object, the adapter checks its queue to see if there are other devices that have requested access. If so, it de-queues the next function address and calls it, thereby granting ownership of the adapter to the called device.

If the adapter object was created for a device that does not support scatter/gather, the adapter object provides access to resources that enable mapping of a limited range of bus space to contiguous *virtual* memory. Recall from the earlier discussion of multiple bus architectures that this mapping depends on how many buses are implemented on a particular computer and the relationship between them. Some computers have mapping registers that map a contiguous range of a peripheral bus to an arbitrary set of memory pages on the physical CPU bus. For computers lacking such registers, using a range of contiguous physical memory as an intermediate buffer for the transfer affects the mapping. In either case, the mapping is transparent to kernel mode drivers. A mapping from a device on a peripheral bus to contiguous virtual memory consumes scarce resources regardless of the physical implementation of the mapping.

Common Buffer Objects

Common buffer objects are implemented by class **KCommonDmaBuffer**.

A common buffer is simply a range of physically contiguous virtual memory shared between a device and its driver. The storage is allocated from system memory and is guaranteed to be accessible by the device and the driver at all times. The attributes of a common buffer include its size, virtual address, and logical address. Driver software uses the virtual address to access the buffer. The logical address is the physical address a device on a peripheral bus must assert in order to access the buffer.

Construction of a common buffer requires a DMA adapter object. In order to provide a buffer accessible by a device, the system requires the characteristics of the device and the bus on which it resides. Like any mapping of a peripheral bus to contiguous virtual memory, a common buffer might consume scarce system resources.

A driver creates a common buffer object to deal with devices that can initiate DMA without direction from the driver. This applies to many bus master devices and to slaved devices that exploit the auto-initialize capability of the system DMA controller. In a typical scheme, a bus master writes data into the common buffer, and asserts an interrupt to signal the driver of its presence. The driver and the device can both access the region without re-mapping the entire DMA transfer for each exchange of information.

DMA Transfer Objects

***DMA transfer objects are implemented by class* KDmaTransfer.**

DMA transfer objects manage the transfer of data between memory and a device. A DMA adapter object for the device must be available. The adapter object informs the transfer object of the type of transfer to be made, i.e. whether it is slaved or bus master, scatter/gather or not, which DMA channel is to be used, etc.

The purpose of the transfer object is to oversee the transfer, calling out to external functions as needed to manipulate the physical device. The transfer object acquires control of the associated adapter object, breaks up the transfer into segments that conform to the limits of the hardware and available buffers, and flushes caches and buffers as necessary. The client of a transfer object supplies code to pass physical memory addresses to bus masters, to tell slaved devices to start requesting DMA cycles, and to detect when a segment of a transfer has completed. The executive transfer object does most of the work.

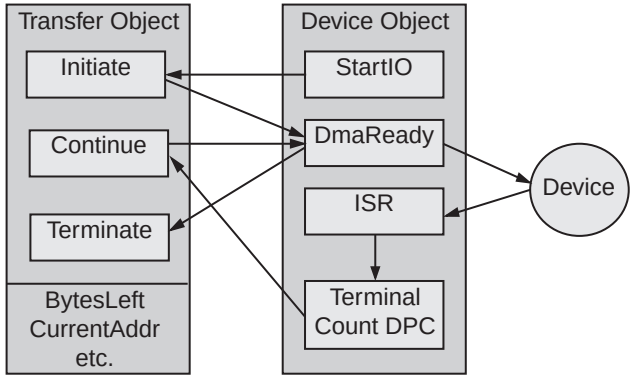


Figure 3-8. **Control Flow for DMA Transfer**

The transfer object exports a method to initiate a transfer. This method takes as parameters the direction of the transfer, a memory object that describes the buffer to or from which the transfer is to take place, and the address of a function to be called each time the transfer object has set up the next segment of the transfer. When the transfer object calls this function, the client causes the device to start transferring the current segment. When the client later detects completion of the segment, usually by an interrupt, it calls a method of the transfer object that continues with the next segment. This is repeated until the client callback function detects the transfer is complete and calls the transfer's termination method.

Memory Objects

***Memory objects are implemented by class* KMemory.**

A memory object is a description of a region of virtual memory. Its attributes include the following items.

- Virtual address of the page where the region starts
- Offset on that page of the start of the region
- Size of the region, the process in which the virtual address is valid
- System virtual address of the region (if one has been mapped)
- Physical page numbers for each page of the region when it is locked into memory

Memory objects are also referred to as Memory Descriptor Lists, or MDLs.

Memory objects are referred to by IRPs for devices that do direct I/O. When such a device performs a read or write, the I/O Manager locks down the user buffer and creates a memory object to describe it. It passes the address of the memory object to the device in an IRP.

Memory objects are also required by DMA transfer objects to initiate a transfer.

Spin Lock Objects

***Spin lock objects are implemented by class* KSpinLock.**

Kernel mode drivers use *spin locks* to maintain integrity of data in multiprocessor systems. A driver associates a spin lock with data it must access atomically. A spin lock exports two methods: lock and unlock. By requiring all code that accesses the data to acquire (lock) the associated spin lock before doing so, the driver ensures that different processors do not interleave accesses to the data in a way that could corrupt it.

The spin lock mechanism is twofold: The lock method first raises IRQL to DISPATCH_LEVEL. Then, on a multiprocessor system, it performs a read-modify-write cycle on a variable that indicates the state of the lock. If it was already locked, the code loops, or spins, until a thread running on another processor unlocks the lock. The lock method returns when it succeeds in changing the state to locked. The caller remains at DISPATCH_LEVEL until the unlock method is called.

On a single processor system, acquiring a spin lock is equivalent to raising IRQL to DISPATCH_LEVEL. Therefore, spin locks are important for arranging atomic operations even on single processor systems, since only code running at device IRQL or higher can interrupt code that holds a spin lock.

Spin locks cannot be used to synchronize access to data that are accessed by interrupt service routines. This requires using the synchronization method of the interrupt object, as described above. Furthermore, interrupt service routines cannot lock arbitrary spin locks, because doing so would lower IRQL to DISPATCH_LEVEL.

Dispatcher Objects

***Dispatcher objects are implemented by the class* KDispatcherObject.**

The system supports a class of objects for synchronization and scheduling, called *dispatcher objects*. Despite the nomenclature, these have nothing to do with DISPATCH_LEVEL nor dispatching of IRPs. Rather, they are objects that interact with scheduler.

Applications can create dispatcher objects to which kernel mode drivers can gain access. Therefore, it is possible to share dispatcher objects between drivers and applications, which is useful for signaling and synchronizing user mode code with kernel mode code.

All dispatcher objects have the property of being in one of two states: signaled or not signaled. Furthermore, all dispatcher objects export a method which causes its callers to block execution while the object is in the not signaled state. This method is named Wait.

When a thread calls method Wait of a dispatcher object that is not signaled, the thread suspends. When the object becomes signaled, the thread might return from method Wait and continue execution. Each of the various kinds of dispatcher objects behaves somewhat differently. The dispatcher objects are: *events*, *semaphores*, *mutexes*, *timers*, and *system threads*.

Events

***Event objects are implemented by class* KEvent.**

The simplest dispatcher object is the *event*. An event can be set, cleared, queried, and waited on.

Semaphores

***Semaphore objects are implemented by class* KSemaphore.**

The attribute that distinguishes semaphores from the other dispatcher objects is a count. The state information for a semaphore includes a count which increments when the semaphore is signaled and decrements when a wait is satisfied. When the count is greater than zero, the semaphore is in the signaled state.

A semaphore's count enables it to perform useful synchronization operations. Suppose a driver maintains a pool of *n* buffers and it has a dedicated set of threads that require exclusive access to one of the buffers in order to run. The driver can then initialize a semaphore with a count of *n* and each thread can wait on the semaphore to acquire a buffer. When no buffer is available, the threads block.

Here is another example. Suppose a driver queues I/O requests (IRPs) to a dedicated thread. The driver creates a semaphore whose count corresponds to the number of IRPs in the queue. Initially, the count is zero and the thread is blocked. As IRPs arrive, the driver queues them and signals the semaphore, thereby unblocking the thread. The thread un-queues the request, processes it, and waits on the semaphore again. By ensuring that the semaphore count is equal to the number of IRPs in the queue, the driver ensures the thread runs only when there is work to be done.

Mutexes

***Mutex objects are implemented by class* KMutex.**

Drivers use mutex objects to protect sections of code or data objects from simultaneous access by multiple threads. Once a thread gains ownership of a mutex, any other thread that attempts to gain ownership of that mutex is blocked until the owner releases it.

A driver that uses multiple threads often needs a mechanism to prevent multiple threads from accessing sensitive resources simultaneously. Operations that leave data objects in a temporary transient state must be performed atomically in order to prevent corruption. For example, insertion of an object into a queue might require manipulation of several related pointers. Since the states of the pointers are related, accessing them in a transient state would corrupt the queue. By forcing threads to acquire a mutex before accessing the queue, a driver ensures the integrity of the queue.

Mutexes have the attribute that they can be recursively acquired. That is, a thread that owns a mutex can enter it again. However, the thread must release the mutex exactly once for each time the thread acquires it.

Fast Mutexes

***Fast Mutex objects are implemented by class* KMutexFast.**

Fast mutex objects are similar to regular mutexes in that they are used to protect sections of code or data objects from simultaneous access by multiple threads. They *are* faster to acquire and release than regular mutexes. However, they cannot be acquired recursively, and, while a fast mutex is held, the IRQL is raised to APC level, preventing the system from receiving APCs.

Timers

***Timer objects are implemented by class* KTimer.**

Timer objects enable kernel mode drivers and system threads to arrange for notification when a specified time interval elapses. Like all dispatcher objects, a timer is either in the signaled or non-signaled state. When first set, it is non-signaled; when the timer expires, it becomes signaled. In addition to methods inherited by all dispatcher objects, the class has a method to set the timer, to cancel it, and to query its state.

Timed Callbacks

Timed callback objects are implemented by class **KTimedCallback**.

Timed callback objects are derived from timer objects. The constructor takes as a parameter the address of a function to call when the timer expires. The object model allows this function to be a member of any class defined or derived by the developer of the driver. A driver can cancel a timed callback while it is pending execution. The object uses a DPC object internally to invoke the callback function.

System Threads

***System thread objects are implemented by class* KSystemThread.**

Kernel mode drivers can create system thread objects. A thread object is an execution context under the control of the system scheduler. Threads can be used to implement continuous processing at the system level, rather than depending on external events or calls from the user subsystem to instigate driver actions.

Threads are appropriate for drivers that have long-term tasks to carry out, such as polling a slow device that cannot interrupt. By utilizing a thread, the system avoids borrowing time quanta from an arbitrary thread. Instead, the scheduler can balance execution of the system thread according to the load on the system.

When the thread is active, it is not signaled. When it terminates, it becomes signaled. This allows a waiter to take action when the thread terminates.

A system thread exploits the fact that its IRQL is PASSIVE_LEVEL. This removes restrictions on which system services it can call, and enables it to access paged code and data. It also enables use of synchronization objects such as events, semaphores, mutexes, and timers.

Containers and Other Objects

There are additional useful classes in the object model which aid in driver development. Some of these are more like container classes, such as the list and FIFO classes. Others are more general, such as file objects and strings. Most are based on system services, while others were developed for DriverWorks to simplify the development of typical drivers.

List Objects

***List objects are implemented by classes* KList, KInterlockedList, *and* KInterruptSafeList.**

The list objects present a simple implementation of doubly linked lists. Use of templates enables type-safe manipulation of objects in the list. The methods of this object include those for insertion, deletion, traversing the list, and testing for emptiness.

There are three types of list templates, each having different synchronization methods. For the simplest list object, no synchronization is provided and it is the responsibility of the developer to ensure corruption does not occur. The second form uses a spin lock to protect all access to internal pointers. The third form requires a connected interrupt object and uses a specialized set of protected methods that run under the control of the interrupt object's synchronization method. ISRs can manipulate this last type of list object.

FIFO Objects

***FIFO objects are implemented by classes* KFifo, KLockableFifo, *and* KInterruptSafeFifo.**

FIFO is an acronym for "first in, first out", which describes the behavior of the object with respect to data written to and read from it. Conceptually, it is a pipe: data enters at one end and emerges at the other. FIFOs are very useful for buffering data between devices or between an application and a device. The methods exported by a FIFO include those for reading, writing, querying available data, and flushing.

Like list objects, FIFO objects are implemented using templates, enabling type-safe access. Declaration of a FIFO requires specification of the data type to be buffered, and the methods for reading and writing operate only on data of that type.

Again like list objects, there are three types of FIFO templates, each having different synchronization methods: no synchronization, spin lock protected, and synchronized via an interrupt object.

Files

***File objects are implemented by class* KFile.**

The file object encapsulates kernel mode file services. The methods for file objects are typical of most abstractions of files: create, open, close, read, write, seek, query properties, and set properties.

Files are accessible only when IRQL is PASSIVE_LEVEL, which limits the utility of file objects in kernel mode drivers. Much of the execution of a driver occurs at higher IRQLs, and the driver cannot access files while in that state. For this reason, drivers sometimes create system threads to handle file operations.

Event Log Entries

***Event Log Entry objects are implemented by class* KErrorLogEntry.**

The operating system provides a facility for logging events of interest to the system administrator. Drivers use this facility to document error conditions encountered during execution. This is preferable to displaying confusing error messages or bringing down the system. The system administrator can view the event log with a system utility.

This event log entry object encapsulates the services of the system error logger. Error log entries consist of a fixed length structure containing the event parameters, optional Unicode strings, and optional binary data. Drivers create an error log entry object and then post it to the system. The class has methods to set the binary data and to add strings to the entry. It allows direct access to the event parameter structure, enabling the driver to set the various fields efficiently.

Strings

***String objects are implemented by class* KUstring.**

String objects are ordered groups of Unicode characters. The attributes of a string include its current length and the maximum length to which it can grow. Objects which are kept in low-level system object directories, such as registry keys, files, device names, symbolic links, and dispatcher object, use string objects for naming. Strings are also used for creating event log entries.

The methods of a string object enable assignment, concatenation, comparison, access to first and last characters, and various type conversions.

A specialized string is provided for *unitized names*, i.e., strings that consist of a base string to which a number is appended. These are useful for device names and symbolic link names, and, occasionally, for accessing the registry.

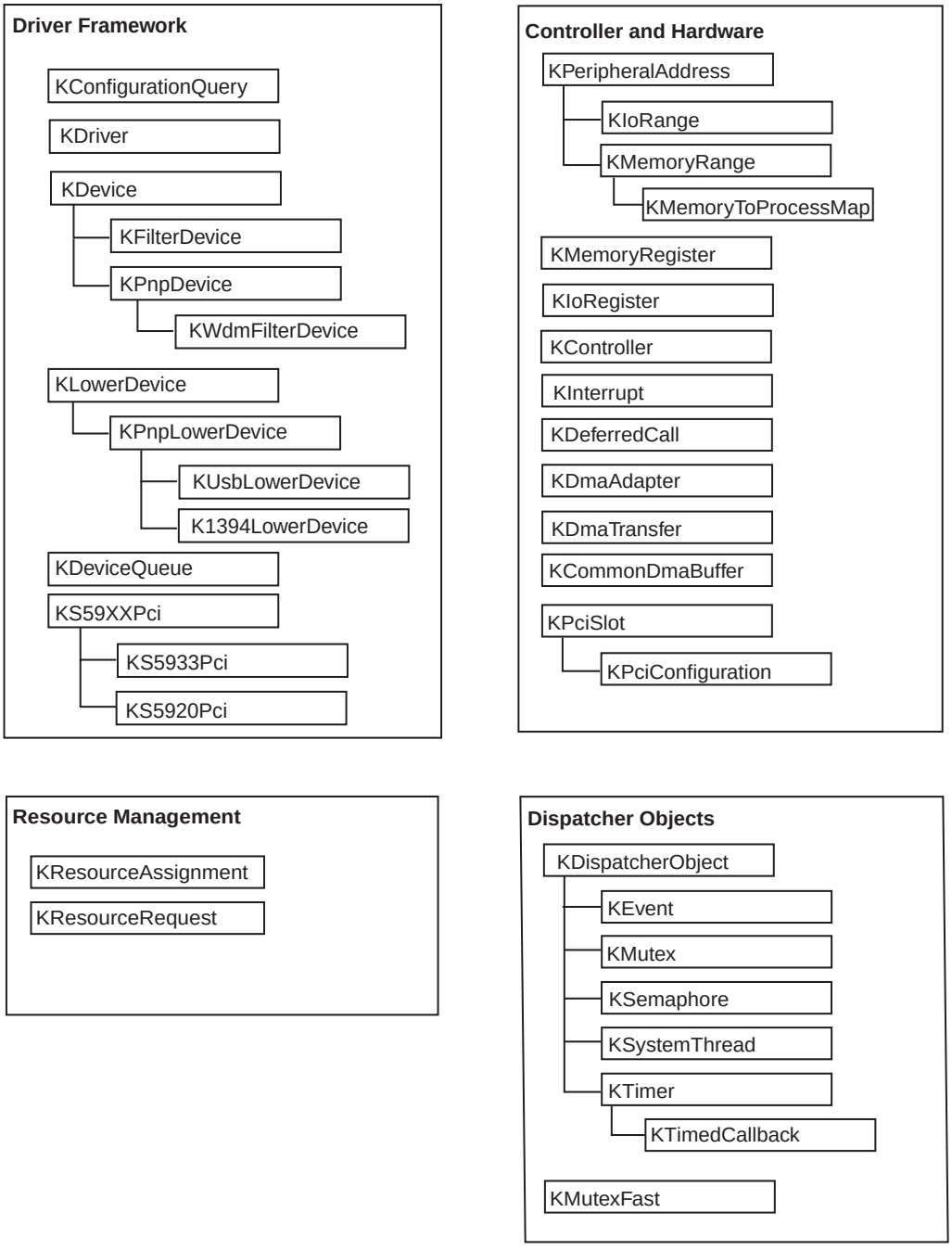
Heaps

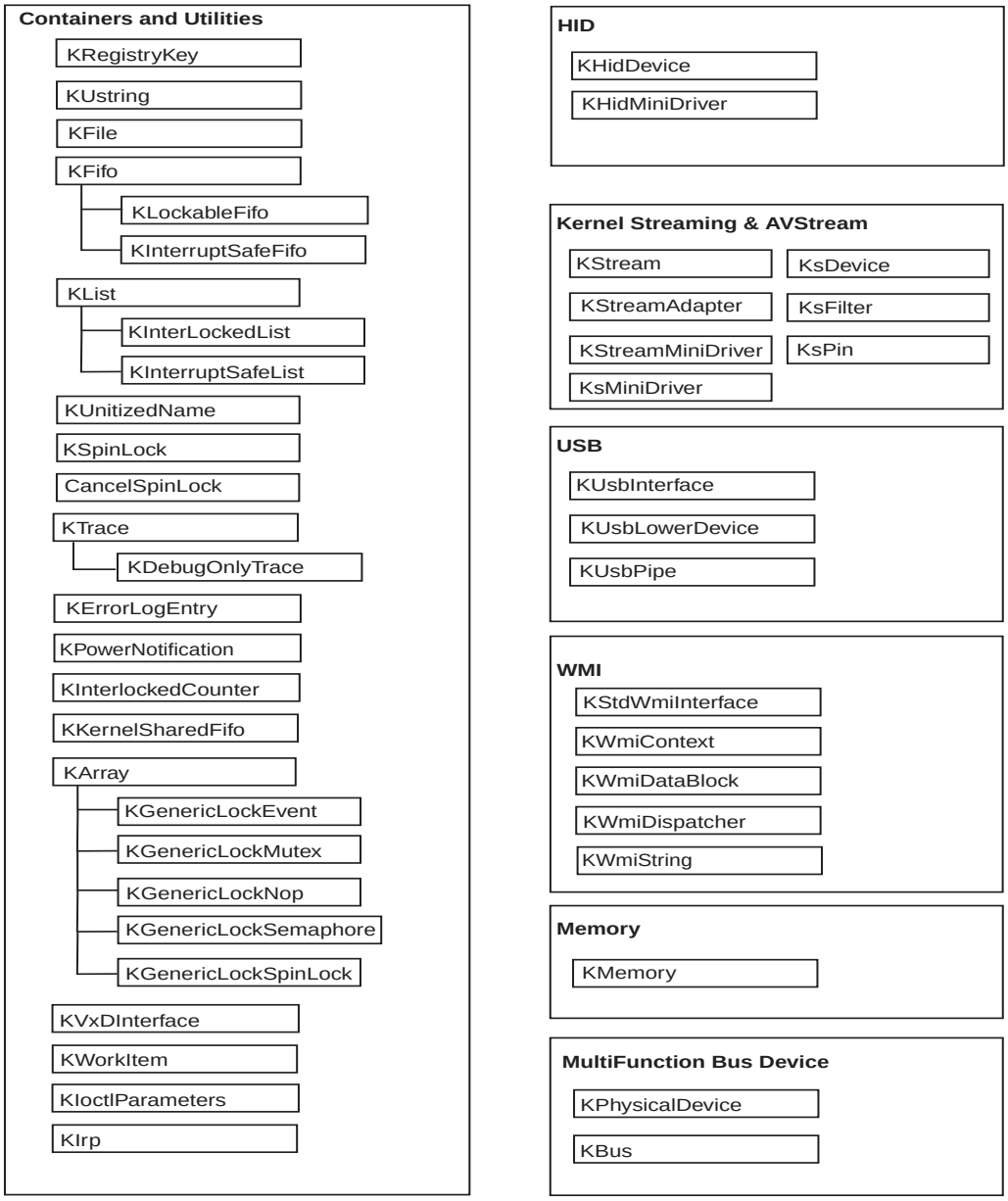
***Heap objects are implemented by the heap template classes* KHeap, KHeapClient, KPagedHeap, *and* KPagedHeapClient .**

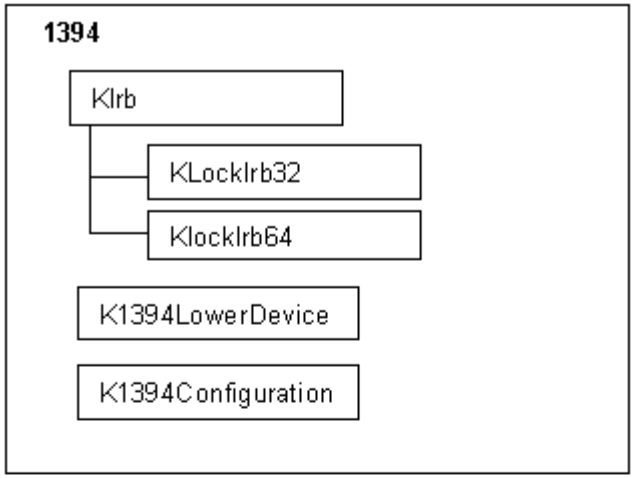
The heap template classes provide alternate heaps for drivers that make frequent allocations and deallocations of small blocks of a fixed size. Using an alternate heap might improve performance, and, more importantly, help prevent fragmentation of the system memory pools.

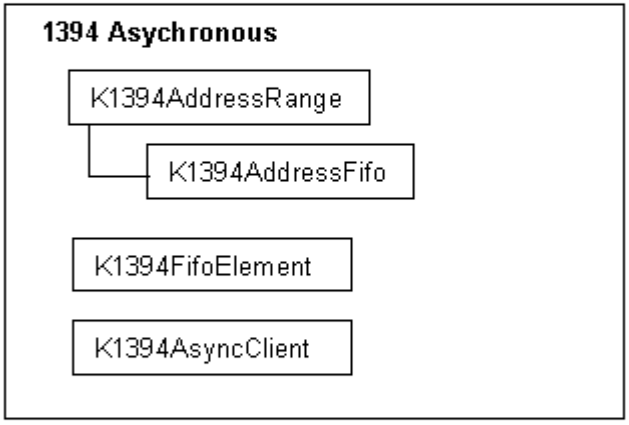
The heap classes are built on an operating system feature known as lookaside lists. A lookaside list is a set of fixed size blocks chained together in a list. Initially the list is empty. When a driver tries to allocate a block from the list, the list object tries to find a free block in the list. If not found, one is allocated from the system pool and put into the list. When the client frees the block, it stays in the list as a free block and is available for allocation.

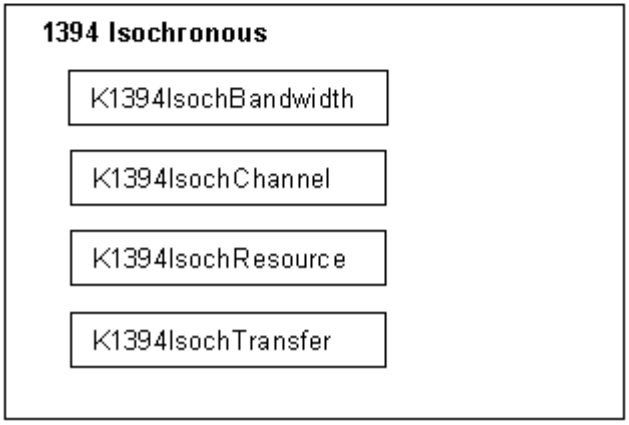
Chart of DriverWorks Classes




Containers and Utilities
KRegistryKey
KUstring
KFile
KFifo
KLockableFifo
KInterruptSafeFifo
KList
KInterLockedList
KInterruptSafeList
KUnitizedName
KSpinLock
CancelSpinLock
KTrace
KDebugOnlyTrace
KErrorLogEntry
KPowerNotification
KInterlockedCounter
KKernelSharedFifo
KArray
KGenericLockEvent
KGenericLockMutex
KGenericLockNop
KGenericLockSemaphore
KGenericLockSpinLock
KVxDInterface
KWorkItem
KIoctlParameters
KIrp
HID
KHidDevice
KHidMiniDriver
Kernel Streaming & AVStream
KStream
KsDevice
KStreamAdapter
KsFilter
KStreamMiniDriver
KsPin
KsMiniDriver
USB
KUsbInterface
KUsbLowerDevice
KUsbPipe
WMI
KStdWmiInterface
KWmiContext
KWmiDataBlock
KWmiDispatcher
KWmiString
Memory
KMemory
MultiFunction Bus Device
KPhysicalDevice
KBus


1394
Klrb
KLockIrb32
KlockIrb64
K1394LowerDevice
K1394Configuration


1394 Asychronous
K1394AddressRange
K1394AddressFifo
K1394FifoElement
K1394AsyncClient


1394 Isochronous
K1394IsochBandwidth
K1394IsochChannel
K1394IsochResource
K1394IsochTransfer

Chapter 4
Windows Driver Model

- Overview of the WDM
- Class KPnpDevice Overview
- Class KPnpLowerDevice
- Making a WDM Driver Callable by a VxD
- WDM Device Interfaces
- WDM Power Management
- Adding WMI Support to a Driver
- Classes for Human Interface Device Support
- Universal Serial Bus (USB) Drivers
- IEEE 1394 Drivers
- WDM Streaming Drivers

DriverWorks enables you to develop drivers for Windows Driver Model (WDM) platforms if you have the required DDK from Microsoft. WDM is supported in the Windows 98, Windows Millennium, Windows 2000, and Windows XP DDKs.

Overview of the WDM

The WDM is a specification for writing device drivers. A driver that conforms to the WDM specification is responsible for handling a set of I/O requests (IRPs) delivered to it by the system. In order to satisfy these requests, the driver can call system services specified by WDM and issue I/O requests to be handled by other drivers, including particular system drivers that are designed into the WDM architecture.

WDM drivers have a great deal in common with NT drivers. The basic IRP handling mechanism is the same. Furthermore, a large fraction of the system services available to NT drivers is available to WDM drivers and works identically. Driver writers who have written NT drivers are immediately familiar with the WDM programming environment.

There are some significant differences between WDM drivers and NT drivers. First of all, WDM drivers must handle a set of dynamic configuration requests (the Plug and Play IRP), as well as a set of power management requests. Second, whereas NT drivers rely on HAL services to interface to the hardware, much of the responsibility for hardware interfaces is taken by system bus drivers in WDM. Third, WDM promotes a class/minidriver schema under which functionality common to a set of devices is implemented in a generalized class driver that interfaces to subordinate minidrivers, each of which provides device specific input and/or output to the class driver.

Plug and Play and Power Management

The WDM environment is designed to support dynamic configuration of devices. This means that devices can be added or removed from the system while it is running and the hardware resources that a device uses can be reassigned by the operating system at any time. Furthermore, the WDM environment is designed to accommodate dynamic changes in system power. The system notifies drivers of configuration and power events by sending IRPs with major functions IRP_MJ_PNP and IRP_MJ_POWER. Each such IRP includes a minor function code that specifies the request. A device driver is responsible for responding correctly to these requests.

In order to model the system configuration, the operating system maintains its own set of device objects to represent physical devices. Each of these device objects is known as a Physical Device Object, or PDO. The operating system creates a PDO when a device is detected and destroys it when the device is removed. In the meantime, it can associate various properties with a PDO, such as its bus information, its device ID, and its power state. All properties and operations performed on PDOs are generic; the operating system uses PDOs for configuration and power management only.

The system notifies a device driver of the presence of a physical device for which the driver is responsible by passing a pointer to the PDO to the driver. The device driver then creates a second device object that it will use for its representation of the device. The driver's device object is called the Functional Device Object, or FDO, because the driver uses it to model the device's function within the system.

A device driver attaches the FDO to the PDO, so that any IRPs addressed to the PDO are seen by the driver. There can be several FDOs attached to a single PDO in a linear chain. As the system detects PnP and power events, it sends IRPs to the affected PDOs. By virtue of having attached the FDO to the PDO, a device driver gains the opportunity to respond to these events at the functional level before passing the request through to the PDO.

The logic for handling PnP and power requests at the functional level is complex. The driver must take into account that there are other I/O requests currently queued or in process. Furthermore, the semantics of some PnP requests change the operational state of device, such that it must defer or fail processing of incoming requests. Fortunately, much of the complexity of WDM is handled in DriverWorks class **KPnpDevice**.

Class Drivers and Minidrivers

A WDM driver is written to interact with other drivers. It might receive IRPs from drivers that rely on it to provide services and it might send IRPs to other drivers. The net effect is a layered architecture in which requests are progressively processed by a series of drivers, each of which performs some operation appropriate to its level in the hierarchy.

Fundamental to the layered architecture are the concepts of *class drivers* and *minidrivers*. A class driver handles requests for a set of devices that fall into a particular class. Its upper edge exposes an IRP-based interface that is generic to all devices in the class. At its lower edge, a class driver exports a defined interface to one or more minidrivers. The role of the minidriver is to provide device specific service to the class driver. By factoring common functionality out of the device specific driver, the class/minidriver architecture reduces the effort required to support new kinds of hardware and enables a more flexible and robust system.

As an example, consider the stream class driver. Its upper edge exposes a generalized interface for manipulating, filtering, and synchronizing multimedia streams, such as audio and video. Its lower edge interfaces to stream minidrivers, which are responsible for generating the stream data. A minidriver does not concern itself with the higher level functions of the class driver. Instead, it is only responsible for delivering stream data to the class driver. When a new multimedia device becomes available, a new stream minidriver enables it to inherit all the capabilities of the stream class.

There can be several pairs of class drivers and minidrivers participating in the processing of a given request. While a minidriver often implements functionality specific to a particular device, it can more generally provide linkage to another class driver. Continuing the example above, a stream minidriver might not get its data directly from some hardware bus, but rather from a network. In this case, the lower edge of the stream minidriver connects to the upper edge of the network class driver. This kind of class/minidriver layering can be configured to arbitrary levels.

Bus Drivers

The WDM environment presumes the presence of system drivers responsible for low-level control and configuration of devices on hardware buses. Access to a system bus driver is via the PDO. A functional driver sends IRPs to the PDO in order to communicate with the device's bus driver.

For example, consider the Universal Serial Bus (USB). The system provides a bus driver that device drivers use to control hardware on the USB. The bus driver provides a higher level interface for putting requests on the USB, so that a functional driver (i.e., a USB client driver) does not need to interact directly with the USB controller. To perform a write operation, for instance, the driver of a USB device builds an INTERNAL_DEVICE_CONTROL request and sends it to the PDO. The PDO's handler for the IRP translates the request into a series of transactions on the USB.

The bus driver architecture extends to internal buses, as well. For example, in WDM, access to the configuration space of the PCI bus, formerly done via HAL services, is accomplished by sending IRPs to the PDO of the PCI device.

Windows Management Instrumentation

Windows Management Instrumentation (WMI) is a protocol that enables providers of measurement and instrumentation data to publish that data for consumption by clients on the local machine or elsewhere on a network.

A driver that supports WMI can expose to applications performance measurements, configuration data, or other kinds of information. WMI also allows client applications to change settings in a driver and to cause a driver to execute specified control functions. (A kernel mode driver is one example of a WMI provider.) Finally, a driver can use WMI to notify an application of some event or alarm condition that the driver detects.

All WDM drivers must support WMI requests by at least forwarding the IRP_MJ_SYSTEM_CONTROL to the PDO.

WMI providers and clients communicate via blocks. A block's type (also known as its class) is identified by a Globally Unique Identifier (GUID). A GUID is a 128-bit structure defined by Windows.

The type and format of the information that a block contains are described in a text file, using a description language called Managed Object Format (MOF). Classes are related in a schema, or hierarchy, which allows one class to be based on the definition of another. This data architecture is referred to as Common Information Model (CIM) and is part of an industry-wide initiative for administration of networked computers.

A block may contain data items and methods. Each data item is of a specific CIM type — such as boolean, string, datetime, or uint32. The data model permits arrays of these primitive types.

Each data item must have a **WmiDataId** qualifier, in order to enable WMI to extract data from the block. Each data item may optionally include other qualifiers, such as description, read, write, or key. The full list of qualifiers is found in the Windows 2000 DDK.

A block's associated methods are functions that can be called by applications. Each method item specifies that method's required input and output parameters.

Some blocks are distinguished as describing events. An application can use WMI calls to wait for notification that a particular event has occurred. A driver signals the event with a special call, which causes the WMI subsystem to deliver the contents of the block describing the event to the waiting client application.

The DriverWorks library provides an object-oriented means of quickly adding WMI support to a driver:

- DriverWorks automatically organizes WMI blocks, simplifying registration and deregistration.
- With minimal direction from the driver writer, DriverWorks dispatches WMI requests to the appropriate handlers.
- DriverWorks classes provide default handlers that — in many cases — satisfy WMI requests (requiring no additional programming).
- DriverWorks automatically dispatches method execution requests to the targeted object member function.
- DriverWorks provides a simple method for signaling WMI event blocks.

WDM and DriverWorks

DriverWorks fully supports WDM driver development. Through derivation from DriverWorks classes, a driver easily inherits a great deal of functionality for correct handling of requests in the dynamic WDM environment. Furthermore, DriverWorks includes classes that simplify the interfaces to various system class drivers (such as HID, Stream) and bus drivers (such as PCI, 1394, and USB). Future versions of DriverWorks will include classes that encapsulate the interface to other WDM classes and bus drivers.

Writing a WDM Driver with DriverWorks

Here are the guidelines for building a driver for a WDM platform using DriverWorks:

- The NTVERSION environment variable must be set to 'WDM':

  ```
  set NTVERSION='WDM'
  ```

  Note the single quotes; they are required. You can also set this in the `SOURCES` file or the project settings.

- Most WDM drivers have an entry point named **AddDevice**, which the system calls when it detects the presence of a device for which the driver is responsible. If your `function.h` `#define` is DRIVER_FUNCTION_ADD_DEVICE, then the class you derive from **KDriver** must override the pure virtual member function **AddDevice**.

  For most WDM drivers, member function **AddDevice** accepts as a parameter a pointer to a system device object. This device object represents the physical device in the system and is referred to as the Physical Device Object, or PDO. For those familiar with the Windows 95 Plug and Play architecture, there is a strong analogy between the PDO and a 'devnode'.

  Member function **AddDevice** is tasked with creating an additional device object, named the Functional Device Object, or FDO. Function **AddDevice** typically creates an instance of a subclass of **KPnpDevice** to serve as the FDO. This class, the analogue of **KDevice** in non-PnP drivers, has a set of overridable virtual member functions which correspond to the subfunctions of the I/O requests related to configuration and power management.

- The interface to the upper edge of a WDM bus driver is implemented by class **KPnpLowerDevice**, which is the analogue of **KLowerDevice** in non-PnP drivers. Initialization of the object connects the FDO to the PDO. This class also provides access to the properties of the PDO.
- The DriverWorks WDM classes (**KPnpDevice**, **KPnpLowerDevice**, and others) simplify the task of writing Plug and Play drivers. Because Plug and Play drivers can be stopped, removed, or restarted as hardware is added or removed to a running system, the device driver must respond to an expanded set of messages (new Plug and Play IRPs) and track additional state information. DriverWorks simplifies this with the DriverWorks Plug and Play Policy Manager. The driver developer gets fine-tuned control over Plug and Play behavior by adjusting policy settings that control how the base classes handle various Plug and Play events.
- DriverWorks provides several classes to support Universal Serial Bus clients. These classes facilitate the abstraction of USB devices and offer easy delivery of requests to the physical device. The classes are based on both the USB specification and the system USB class driver. For USB clients, the lower device object is represented by a instance of **KUsbLowerDevice**, which is a subclass of **KPnpLowerDevice**. Sample drivers that demonstrate both Bulk and Isochronous data transfer are included.
- DriverWorks provides a set of classes to support development of IEEE 1394 drivers. These classes support both asynchronous and isochronous I/O on the 1394 bus, and make it easy for the driver to send requests (IRBs) to the 1394 bus driver. Several 1394 examples demonstrating both asynchronous and isochronous I/O are included.
- DriverWorks provides a set of classes to support development of Stream Minidrivers, along with a sample Video Capture driver.
- Some of the DriverWorks examples are written for NT 4.0, while others are WDM examples. The NT 4.0 examples are in the `DriverWorks\examples\nt` folder, while the WDM examples are in the `DriverWorks\examples\wdm` folder. The `DriverWorks\examples\dual` folder contains samples for either platform. Though similar, note that NT and WDM drivers are not interchangeable. WDM drivers will not run on Windows NT 4.0, and NT 4.0 drivers will not generally work on Windows 98. The `DriverWorks\examples\non-framework` directory contains example DDK drivers using a few DriverWorks classes. These examples can either be NT or WDM, and illustrate how to use the DriverWorks classes in a non-DriverWorks driver.

- If your driver needs to conditionalize code for WDM, you can use the following syntax *after* `#include <vdw.h>`:

```
#if _WDM_
// do stuff specific to WDM
#endif
#if ! _WDM_
// do stuff that is not allowed/desired under WDM
#endif
```

- To test if the driver is running under Windows 98 or Windows ME (as opposed to any NT platform), you can simply test global BOOLEAN **_bWindows98_**.

  This is declared in `kdriver.h`, which is included in `vdw.h`.

```
if (_bWindows98_)
{
// do stuff specific to Windows 98
}
```

WDM Documentation

The WDM documentation in this version of DriverWorks includes the following topics:

- Class **KPnpDevice** and its implementation of Plug and Play policy options and Power Management options. These sections describe how to build a driver that correctly handles Plug and Play messages and Power Management messages.
- Class **KPnpLowerDevice**, the class that models the interface to the PDO.
- Plug and Play Resource Management, which highlights the differences between NT 4 and WDM with respect to allocation of hardware resources (interrupts, memory ranges, I/O ranges, and DMA channels).
- Class **KVxDInterface**, a special purpose class that enables a driver to export a VxD interface. This is active only on Windows 98 and Windows ME.
- Device Interfaces, which are the new WDM way of exposing device functionality to applications and other drivers.

- Classes for **Human Interface Device** support. HID minidrivers take advantage of the HID class driver, which provides a standardized way for applications to interact with Human Interface devices, such as mice, keyboards, and joysticks.
- **Universal Serial Bus**, a new external bus that supports a variety of device types. The DriverWorks classes provide abstractions for modeling the device and interfacing to the system bus driver.
- Classes for **Stream Minidrivers**, which take advantage of the kernel stream class driver for support of audio and video streams.
- Classes for adding WMI support to a WDM driver.
- Classes for IEEE 1394 drivers.

Class KPnpDevice Overview

Class **KPnpDevice** is a subclass of **KDevice** designed to support Plug and Play drivers in the WDM environment. It is used only as a base class.

The member functions consist primarily of handlers for the major functions of IRP_MJ_PNP and IRP_MJ_POWER. The class overrides **KDevice::Pnp** and **KDevice::Power** with a dispatcher that calls the member function corresponding to the major function code of the IRP.

All the major function handlers are virtual; a subclass of **KPnpDevice** overrides the member functions it needs to handle explicitly. The subclass *must* also override member functions **DefaultPnp** and **DefaultPower**, pure virtual members which are called by the base class implementations of the minor function handlers.

There are several member functions that are commonly overridden. The first is **OnStartDevice** in which WDM drivers receive a list of resources the system has assigned to the device. These resources are claimed and initialized in **OnStartDevice**. For drivers that use system resources, three other member functions — **OnStopDevice**, **OnRemoveDevice**, and **OnSurpriseRemoval** — should also be overridden, to allow the resources to be properly released.

If your driver is a WDM driver, it must handle IRP_MJ_PNP and IRP_MJ_POWER. Therefore, you must have **#define DRIVER_FUNCTION_PNP** and **#define DRIVER_FUNCTION_POWER** in your `function.h` file. As noted above, **KPnpDevice** implements **Pnp** and **Power**, so the subclass normally does not need to do so. Note that if your subclass does override **Pnp** or **Power**, it is responsible for dispatching IRPs to the major function handlers. It can do this by invoking the base-class member function.

Class **KPnpDevice** implements intelligent preprocessing and post-processing of a subset of the PnP minor functions. The driver controls this processing via a set of policy options, which are set up during class construction. This approach enables the driver writer to customize the degree of support to be provided by the base class.

There are several components that a driver writer can enable by setting the various policy options. When the appropriate options are selected, DriverWorks code will:

- Maintain a count of outstanding I/O requests
- Maintain a count of open handles to the device
- Monitor the device's PnP state (e.g. started, stopped, removed, etc.)
- Monitor the device's Power state (e.g. on, off, sleep, etc.)
- Forward requests to the Physical Device Object (PDO)
- Handle queries to stop and remove the device
- Maintain a queue of requests for possible deferred processing
- Detach the PDO on remove
- Delete the underlying system device object
- Delete the device class instance
- Request Power IRP's

Class **KPnpDevice** has member functions that will initialize a default PnP and default power management policy. The driver writer can call these functions, and then change individual policy options to tailor the driver to a particular implementation. To a large degree, the various components of the PnP and power functionality operate independently of one another.

The objective of the PnP and Power support is to facilitate implementation of PnP and Power logic. A device can easily inherit a vast amount of important functionality, with the ability to selectively enable or disable individual features. The correspondence of member functions to IRP minor functions provides a simple mechanism for adding custom logic to handle particular PnP and Power requests.

Although there are more than 30 policy options available to the developer, it is far easier to selectively enable these options for a particular driver than to implement all the functionality controlled by these options. Furthermore, the availability of a 'standard' policy enables the developer to select a workable default set and then fine-tune according to specific requirements.

The net benefit to the driver writer is that the base class minimizes the amount of code a developer must write in order to implement a new WDM driver or to port an existing NT 4.0 driver to WDM.

PnP and Power Functionality Implemented in Class KPnpDevice

Maintain a Count of Outstanding I/O Requests

Counting outstanding requests is important because a driver can take different actions depending on if it has received requests that are still in process. For example, a driver can choose to wait for outstanding I/O to complete before allowing a device to be stopped.

Maintain a Count of Open Handles to the Device

Counting open handles is in some ways similar to counting outstanding requests, because when there are open handles on a device it might respond differently to some events. For example, a device might fail a query for removal if there are open handles to it.

Monitor the PnP State of the Device

At any given time, the state of a device with respect to the PnP system determines how it responds to PnP events. The state variables maintained automatically by class **KPnpDevice** are STARTED, REMOVED, STOP_PENDING, and REMOVE_PENDING. There are several policy options which depend on the device state. For example, a driver typically wants to fail all I/O requests for a device that has been removed.

Monitor the Power State of the Device

At any given time, the state of a device with respect to the Power system (e.g., on, off, sleep, etc.) determines how it responds to Power events and normal device I/O processing. The state variables maintained automatically by class **KPnpDevice** are DEVICE_POWER_STATE (corresponding to the on, off and sleep states defined by WDM: PowerDeviceD0, PowerDeviceD1, PowerDeviceD2, and PowerDeviceD3), transition states such as PowerOffPending, SleepPending, WaitWakePending, and IdleDetectionEnabled. There are several policy options which depend on the device power state. For example, a driver typically wants to power on the device if it is in a low power state when an I/O request for the device is received.

Forward Requests to the Physical Device Object

Depending on policy settings, class **KPnpDevice** will automatically handle forwarding of a subset of the PnP IRP minor functions to the Physical Device Object (PDO). To obtain this functionality, the driver must call a function that informs the device of the top-of-stack device to which it has been attached. This is usually done in the constructor. For some minor functions, the PDO sees the IRP prior to the device, and for other minor functions, after. The base class enforces the correct behavior.

Handle Queries to Stop and Remove the Device

There is a set of policy options specific to handling of minor functions IRP_MN_QUERY_REMOVE_DEVICE and IRP_MN_QUERY_STOP_DEVICE. The driver may configure a device to fail either query if there is outstanding I/O, to fail if the device is open, to wait for outstanding I/O to complete, to cancel the current IRP, and/or to cancel IRPs in the device queue. The last two options apply only if the device is using system queuing, and not driver managed queues. There is an optional timeout period when waiting for outstanding I/O to complete.

Maintain a Queue of Requests for Possible Deferred Processing

In certain device states, such as when a QUERY_STOP request is pending, a driver might want to defer processing of all non-PnP requests. A device can inherit this ability from **KPnpDevice** and enable it via a set of policy options. The options control the circumstances under which IRPs will be deferred, how the deferred requests are to be handled when the device is stopped or removed, and under what conditions the deferred IRPs are to be dispatched for processing.

Detach the PDO on Remove

A device that sets up its PDO with **KPnpDevice::SetLowerDevice** may enable an option that automatically detaches the PDO when the device is removed. This option is recommended unless handled by the device in or before its **KPnpDevice::OnRemoveDevice** handler, since the device will be deleted when the REMOVE request is handled.

Delete the Device Object

When a device is removed and has no handles opened to it, its class instance should be deleted. Class **KPnpDevice** can detect this either on a REMOVE request or a CLOSE request, and delete the device class instance if the corresponding policy option is asserted. In all cases, the underlying system device object is always deleted in the processing of the REMOVE request.

Request Power IRPs

Depending on policy settings, class **KPnpDevice** will automatically handle requesting Device Power IRPs when power state transitions are required by the system or by the state of the device. The base class enforces the correct behavior for handling System Power IRPs.

Class KPnpDevice PnP Policies

The driver writer controls the implementation of the functionality of the base class described in the previous section by asserting policy options. To assert a particular option, a driver simply sets the corresponding flag in the data member of class KPnpDevice where the policies are stored. The implementation uses structures with bit fields.

The set of PnP policies a driver writer sets up for a device are logically divided into the following groups.

General Policies – This set includes some general options related to management of the device object.

Query Remove Policy – This set controls how the device is to respond to IRP_MN_QUERY_REMOVE_DEVICE.

Query Stop Policy – This set controls how the device is to respond to IRP_MN_QUERY_STOP_DEVICE. The set of options is identical to the set available for a remove query.

Deferred Requests Policy – This set controls when requests are to be deferred and how to handle the queue of deferred requests when various events occur.

PDO Policy – This set determines the subset of the PnP minor functions for which the base class is responsible for sending the request to the underlying PDO.

Completion Policy – This set determines the subset of the PnP minor functions for which the base class is responsible for IRP completion. In some cases, the base class completes the IRP by passing it to the PDO.

The data member of **KPnpDevice** that holds the PnP policy is declared as follows:

```
PNP_POLICY m_Policies;
```

Where type PNP_POLICY has the following definition:

```
struct PNP_POLICY
{
GENERAL_POLICY m_GeneralPolicy;
QUERY_POLICY m_QueryStopPolicy;
QUERY_POLICY m_QueryRemovePolicy;
HOLD_POLICY m_HoldPolicy;
PDO_POLICY m_ForwardToPdoPolicy;
COMPLETION_POLICY m_CompletedByDriverWorks;
};
```

The component structures are described below. To set a policy option, the code looks like this:

```
m_Policies.m_HoldPolicy.m_HoldIfStopped = TRUE;
```

Initially, no policy options are asserted. To set up the "standard policy" the constructor of a subclass of **KPnpDevice** can call base class member **SetPnpPolicy**. The standard policy utilizes many of the base class PnP features and is appropriate for many kinds of devices. However, the driver writer must examine the set of options in the standard policy and determine which of those are appropriate for the device to be supported. Furthermore, several options in the standard policy require action by the driver in order to work correctly, so the developer must keep in mind the characteristics of the standard policy at all times.

General Policy

This set includes some general options related to management of the device object.

Data type GENERAL_POLICY has the following definition:

```
struct GENERAL_POLICY
{
FLAG m_FailRequestsIfRemoved:1;
FLAG m_FailRequestsIfNotStarted:1;
FLAG m_DeleteOnRemoveIfNotOpen:1;
FLAG m_WaitForSafeRemove:1;
FLAG m_DetachPdoOnRemove:1;
FLAG m_EnableDisableInterfaces:1;
ULONGLONG m_SafeRemoveTimeout;
};
```

The flags have the following meanings.

- **m_FailRequestsIfRemoved**

  This flag directs the base class to fail requests with STATUS_DELETE_PENDING if the device has been removed. The device is in the removed state if it has received IRP_MN_REMOVE_DEVICE. Note that IRPs with major functions IRP_MJ_CLOSE, IRP_MJ_CLEANUP and IRP_MJ_SHUTDOWN are not affected by this policy.

- **m_FailRequestsIfNotStarted**

  This flag directs the base class to fail requests with STATUS_UNSUCCESSFUL if the device is not started. This option applies to all IRPs except IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_PNP, IRP_MJ_POWER, IRP_MJ_CLEANUP, and IRP_MJ_SHUTDOWN.

- **m_DeleteOnRemoveIfNotOpen**

  This flag directs the base class to delete the device class instance after IRP_MN_REMOVE_DEVICE has been processed, provided that there are no open handles. Furthermore, the base class checks this policy option after IRP_MJ_CLOSE is processed, and deletes the device class instance if it is in the REMOVED state, and there are no open handles. The underlying system device object is always deleted in the processing of IRP_MN_REMOVE_DEVICE.

- **m_WaitForSafeRemove**

  This flag directs the base class to wait for completion of outstanding I/O requests before returning on IRP_MN_REMOVE_DEVICE. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below.

- **m_DetachPdoOnRemove**

  This flag directs the base class to detach the PDO after IRP_MN_REMOVE_DEVICE is processed by the subclass and the request has been forwarded to the PDO. In order to use this option, the device must call **KPnpDevice::SetLowerDevice** in its constructor.

- **m_EnableDisableInterfaces**

  This flag directs the base class to enable any device interfaces that the object has registered when IRP_MN_START_DEVICE is received and disable them when IRP_MN_STOP_DEVICE is received. Member function **RegisterInterface** registers an IRP-based device interface for a device instance.

- **m_SafeRemoveTimeout**

  This variable specifies how long the system should wait for requests to complete before a remove. The default value is zero, which indicates that no timeout is in effect, i.e., the system will wait indefinitely. Units are 100 nanoseconds. This is always a relative value, and should be input as a positive number.

Query Stop / Query Remove Policies

Policy options for IRP_MN_QUERY_REMOVE_DEVICE and IRP_MN_QUERY_STOP_DEVICE are stored in a structure defined as follows.

```
struct QUERY_POLICY
{
FLAG m_FailIfOpen:1;
FLAG m_FailIfOutstandingIo:1;
FLAG m_CancelDeviceQueue:1;
FLAG m_CancelCurrentIrp:1;
FLAG m_WaitForOutstandingIo:1;
FLAG m_UseTimeoutOnWait:1;
ULONGLONG m_WaitTimeOut;
};
```

Although the policy options for the two requests are the same, the semantics of the two requests differ. A request to stop the device means the system wants to continue using the device, but it must be stopped temporarily for reconfiguration. A request to remove the device occurs when the device is being removed from the configuration and is not expected to resume operation.

Policy options `m_FailIfOpen` and `m_FailIfOutstandingIo` are less accommodating to the system in that no attempt is made to cancel a pending I/O or to wait a period for it to complete. These are considered more appropriate for IRP_MN_QUERY_STOP, since it might be unacceptable for a device that is expected to continue after reconfiguration to lose requests.

The flags have the following meanings.

- **m_FailIfOpen**

  This flag directs the base class to fail the request immediately with STATUS_DEVICE_BUSY if there are any handles open. The base class counts open handles by incrementing a counter when the handler for IRP_MJ_CREATE returns STATUS_SUCCESS and decrementing it when the handler for IRP_MJ_CLOSE returns STATUS_SUCCESS.

- **m_FailIfOutstandingIo**

  This flag directs the base class to fail the request immediately with STATUS_DEVICE_BUSY if there are any outstanding requests. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below.

- **m_CancelCurrentIrp**

  This flag directs the base class to cancel the IRP that the device is currently processing, provided the subclass handler (if any) for IRP_MN_REMOVE_DEVICE (virtual member function **OnQueryRemoveDevice**) returns STATUS_SUCCESS. Setting this option presumes the device has a **StartIo** handler and uses the device's embedded IRP queue for serializing processing of I/O requests. The base class calls member function **CurrentIrp** to obtain the current IRP.

- **m_CancelDeviceQueue**

  This flag directs the base class to cancel all requests in the device's embedded IRP queue, provided that the subclass handler (if any) for IRP_MN_QUERY_REMOVE_DEVICE (virtual member function **OnQueryRemoveDevice**) returns STATUS_SUCCESS. Setting this option presumes that the device has a **StartIo** handler and uses the device's embedded IRP queue for serializing processing of I/O requests.

- **m_WaitForOutstandingIo**

  This flag directs the base class to wait for any outstanding I/O to complete prior to returning to calling the PDO or returning to the I/O Manager, provided the subclass handler (if any) for IRP_MN_REMOVE_DEVICE (virtual member function **OnQueryRemoveDevice**) returns STATUS_SUCCESS. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below. The base class checks this policy option after processing options `m_CancelCurrentIrp` and `m_CancelDeviceQueue`.

- **m_UseTimeoutOnWait**

  If policy option `m_WaitForOutstandingIo` is set, this flag directs the base class to wait with a timeout. The timeout period is taken from data member `m_WaitTimeOut` and is units of 100 nanoseconds. If the wait times out, the base class attempts to cancel all outstanding requests.

Completion Policy

The completion policy for a device determines the set of PnP minor functions for which the base class is responsible for completing the IRP. In some cases, the base class might complete the IRP by passing it to the PDO, so if any of these options are enabled, the subclass constructor should inform the base class of its PDO by calling SetLowerDevice.

The set of PnP minor functions to which the completion policy applies is the same set as for the PDO policy.

- IRP_MN_START_DEVICE
- IRP_MN_STOP_DEVICE
- IRP_MN_REMOVE_DEVICE
- IRP_MN_SURPRISE_REMOVAL
- IRP_MN_QUERY_REMOVE_DEVICE
- IRP_MN_QUERY_STOP_DEVICE
- IRP_MN_CANCEL_REMOVE_DEVICE
- IRP_MN_CANCEL_STOP_DEVICE
- IRP_MN_QUERY_PNP_DEVICE_STATE

Basically, when an option is enabled, the base class takes the responsibility for completing the IRP. Use of these options is recommended, because the base class knows when the IRP should be completed with respect to the implementation of other policy options.

The structure used to store the completion policy is defined as follows.

```
struct COMPLETION_POLICY
{
FLAG m_StartDevice:1;
FLAG m_StopDevice:1;
FLAG m_QueryRemove:1;
FLAG m_QueryStop:1;
FLAG m_CancelRemove:1;
FLAG m_CancelStop:1;
FLAG m_Remove:1;
FLAG m_QueryPnpState:1;
FLAG m_SurpriseRemoval:1;
};
```

The flags have the following meanings.

- **m_StartDevice**

  This flag directs the base class to take responsibility for completing IRP_MN_START_DEVICE. The subclass handler, **OnStartDevice**, must not complete the IRP when this option is set.

- **m_StopDevice**

  This flag directs the base class to take responsibility for completing IRP_MN_STOP_DEVICE. The subclass handler, **OnStopDevice**, must not complete the IRP when this option is set.

- **m_QueryRemove**

  This flag directs the base class to take responsibility for completing IRP_MN_QUERY_REMOVE_DEVICE. The subclass handler, **OnQueryRemoveDevice**, must not complete the IRP when this option is set.

- **m_QueryStop**

  This flag directs the base class to take responsibility for completing IRP_MN_QUERY_STOP_DEVICE. The subclass handler, **OnQueryStopDevice**, must not complete the IRP when this option is set.

- **m_CancelRemove**

  This flag directs the base class to take responsibility for completing IRP_MN_CANCEL_REMOVE_DEVICE. The subclass handler, **OnCancelRemoveDevice**, must not complete the IRP when this option is set.

- **m_CancelStop**

  This flag directs the base class to take responsibility for completing IRP_MN_CANCEL_STOP_DEVICE. The subclass handler, **OnCancelStopDevice**, must not complete the IRP when this option is set.

- **m_Remove**

  This flag directs the base class to take responsibility for completing IRP_MN_REMOVE_DEVICE. The subclass handler, **OnRemoveDevice**, must not complete the IRP when this option is set.

- **m_QueryPnpState**

  This flag directs the base class to take responsibility for completing IRP_MN_QUERY_PNP_DEVICE_STATE. The subclass handler, **OnQueryDeviceState**, must not complete the IRP when this option is set.

- m_SurpriseRemoval

  This flag directs the base class to take responsibility for completing IRP_MN_SURPRISE_REMOVAL. The subclass handler, **OnSurpriseRemoval**, must not complete the IRP when this option is set.

PDO Policy

The PDO policy determines the circumstances under which the base class is responsible for forwarding certain PnP IRPs to the PDO. This option is available for the following set of PnP minor functions.

- IRP_MN_START_DEVICE
- IRP_MN_STOP_DEVICE
- IRP_MN_REMOVE_DEVICE
- IRP_MN_SURPRISE_REMOVAL
- IRP_MN_QUERY_REMOVE_DEVICE
- IRP_MN_QUERY_STOP_DEVICE
- IRP_MN_CANCEL_REMOVE_DEVICE
- IRP_MN_CANCEL_STOP_DEVICE
- IRP_MN_QUERY_PNP_STATE

For some of the above, forwarding to the PDO is done synchronously prior to calling the subclass' handler for the IRP. For others, the call to the PDO is done asynchronously after the subclass has processed the request.

Although not required, the driver writer is encouraged to use these options because they ensure that forwarding to the PDO is done at the correct juncture with respect to the implementation of other options.

In order to use these options, the subclass constructor must call member function SetLowerDevice, which informs the base class of the PDO to which the subclass has attached itself. Although it would be possible for the base class to make this determination, the decoupling of **KPnpDevice** from lower edge classes (e.g., KPnpLowerDevice, KUsbLowerDevice) is an important aspect of the DriverWorks architecture. The requirement of calling SetLowerDevice introduces no significant overhead, and is in accordance with the design goal of decoupling upper and lower edges of the driver.

The structure used to store the PDO policy is defined as follows:

```
struct PDO_Policy
{
FLAG m_CallBeforeStart : 1;
FLAG m_CallBeforeCancelStop : 1;
FLAG m_CallBeforeCancelRemove : 1;
FLAG m_CallAfterQueryStop : 1;
FLAG m_CallAfterStop : 1;
FLAG m_CallAfterQueryRemove : 1;
FLAG m_CallAfterRemove : 1;
FLAG m_CallAfterQueryPnpState : 1;
FLAG m_CallAfterSurpriseRemoval : 1;
};
```

The flags have the following meanings:

- **m_CallBeforeStart**

  This flag directs the base class to pass IRP_MN_START_DEVICE to the PDO prior to invoking the subclass' handler, **OnStartDevice**. If the PDO returns with an error, the base class does no further processing on the IRP. The call to the PDO is made synchronously and preserves the IRP for further processing. The subclass handler must complete the IRP if it has not asserted the completion policy option `m_StartDevice`.

- **m_CallBeforeCancelStop**

  This flag directs the base class to pass IRP_MN_CANCEL_STOP_DEVICE to the PDO prior to invoking the subclass' handler, **OnCancelStopDevice**. If the PDO returns with an error, the base class does no further processing on the IRP. The call to the PDO is made synchronously and preserves the IRP for further processing. The subclass handler must complete the IRP if it has not asserted the completion policy option `m_CancelStopDevice.`

- **m_CallBeforeCancelRemove**

  This flag directs the base class to pass IRP_MN_CANCEL_REMOVE_DEVICE to the PDO prior to invoking the subclass's handler, **OnCancelRemoveDevice**. If the PDO returns with an error, the base class does no further processing on the IRP. The call to the PDO is made synchronously and preserves the IRP for further processing. The subclass handler must complete the IRP if it has not asserted the completion policy option `m_CancelRemoveDevice`.

- **m_CallAfterQueryStop**

  This flag directs the base class to pass IRP_MN_QUERY_STOP to the PDO after invoking the subclass' handler, **OnQueryStopDevice**. The base class passes the IRP to the PDO only if **OnQueryStopDevice** returns with a status code indicating success. If the driver sets this option, it normally also sets completion policy option `m_QueryStop`.

- **m_CallAfterStop**

  This flag directs the base class to pass IRP_MN_ STOP to the PDO after invoking the subclass' handler, **OnStopDevice**. The base class always passes the IRP to the PDO if this option is asserted. If the driver sets this option, it normally also sets completion policy option `m_Stop`.

- **m_CallAfterQueryRemove**

  This flag directs the base class to pass IRP_MN_QUERY_REMOVE to the PDO after invoking the subclass' handler, **OnQueryRemoveDevice**. The base class passes the IRP to the PDO only if **OnQueryRemoveDevice** returns with a status code indicating success. If the driver sets this option, it normally also sets completion policy option `m_QueryRemove`.

- **m_CallAfterRemove**

  This flag directs the base class to pass IRP_MN_REMOVE to the PDO after invoking the subclass' handler, **OnRemoveDevice**. The base class always passes the IRP to the PDO if this option is asserted. This is the first action taken after calling **OnRemoveDevice** and precedes implementation of general policies `m_WaitForSafeRemove`, `m_DetachPdoOnRemove`, and `m_DeleteOnRemoveIfNotOpen`. If the driver sets this option, it normally also sets completion policy option `m_Remove`.

- **m_CallAfterQueryPnpState**

  This flag directs the base class to pass IRP_MN_QUERY_PNP_ DEVICE_STATE to the PDO after invoking the subclass' handler, **OnQueryDeviceState**. The base class passes the IRP to the PDO only if **OnQueryDeviceState** returns with a status code indicating success. If the driver sets this option, it normally also sets completion policy option `m_QueryPnpState`.

- **m_CallAfterSurpriseRemoval**

  This flag directs the base class to pass IRP_MN_ SURPRISE_REMOVAL to the PDO after invoking the subclass's handler, **OnSurpriseRemoval**. The base class always passes the IRP to the PDO if this option is asserted.

Deferred Requests Policy

The Deferred Requests Policy is the set of policy options for deferred request processing controls when requests are to be deferred and how to handle the queue of deferred requests when various events occur. This set is stored in a structure defined as follows.

```
struct HOLD_POLICY
{
FLAG m_HoldIfRemovePending :1;
FLAG m_HoldIfStopPending :1;
FLAG m_HoldIfStopped :1;
FLAG m_CancelAllOnCleanUp :1;
FLAG m_CancelAllOnRemove :1;
FLAG m_CancelAllOnStop :1;
FLAG m_ProcessOnCancelStop :1;
FLAG m_ProcessOnCancelRemove :1;
};
```

Class **KPnpDevice** stores deferred requests in the data member declared as follows.

```
KInterlockedList<IRP> m_HoldQueue;
```

IRPs in the hold queue are not counted as outstanding requests.

The following types of IRPs are never put in the hold queue.

- IRP_MJ_CREATE
- IRP_MJ_CLOSE
- IRP_MJ_SHUTDOWN
- IRP_MJ_CLEANUP
- IRP_MJ_PNP
- IRP_MJ_POWER

The policy flags have the following meanings.

- **m_HoldIfRemovePending**

  This flag directs the base class to defer requests to the hold queue if the device has a remove operation pending, i.e., if IRP_MN_QUERY_REMOVE_DEVICE was processed successfully and no subsequent IRP_MN_REMOVE_DEVICE nor IRP_MN_CANCEL_REMOVE request has been received.

- **m_HoldIfStopPending**

  This flag directs the base class to defer requests to the hold queue if the device has a stop operation pending, i.e., if IRP_MN_QUERY_STOP_DEVICE was processed successfully and no subsequent IRP_MN_STOP_DEVICE nor IRP_MN_CANCEL_STOP request has been received.

- **m_HoldIfStopped**

  This flag directs the base class to defer requests to the hold queue if the device is stopped for reconfiguration, i.e., it has successfully processed IRP_MN_STOP_DEVICE and no subsequent IRP_MN_START_DEVICE request has been received.

- **m_CancelAllOnCleanUp**

  This flag directs the base class to cancel all requests in the hold queue when the device receives IRP_MJ_CLEANUP or IRP_MJ_SHUTDOWN. The base class performs this operation prior to calling the subclass' handler for the IRP.

- **m_CancelAllOnRemove**

  This flag directs the base class to cancel all requests in the hold queue when the device receives IRP_MN_REMOVE_DEVICE. The base class performs this operation prior to calling the subclass' handler for the IRP.

- **m_CancelAllOnStop**

  This flag directs the base class to cancel all requests in the hold queue when the device receives IRP_MN_STOP_DEVICE. The base class performs this operation prior to calling the subclass' handler for the IRP.

- **m_ProcessOnCancelStop**

  This flag directs the base class to dispatch all requests in the hold queue to their respective handlers when the device receives IRP_MN_CANCEL_STOP_DEVICE. The base class performs this operation after calling the subclass' handler for the IRP.

- **m_ProcessOnCancelRemove**

  This flag directs the base class to dispatch all requests in the hold queue to their respective handlers when the device receives IRP_MN_CANCEL_REMOVE_DEVICE. The base class performs this operation after calling the subclass' handler for the IRP.

Standard PnP Policy

When an instance of **KPnpDevice** is first constructed, all policy options are FALSE. A driver requests the standard PnP policy by calling member function **KPnpDevice::SetPnpPolicy**. The settings of the standard policy are as follows.

- General Policy

```
m_Policies.m_GeneralPolicy.m_FailRequestsIfRemoved = TRUE;
m_Policies.m_GeneralPolicy.m_FailRequestsIfNotStarted = TRUE;
m_Policies.m_GeneralPolicy.m_DeleteOnRemoveIfNotOpen = TRUE;
m_Policies.m_GeneralPolicy.m_WaitForSafeRemove = TRUE;
m_Policies.m_GeneralPolicy.m_DetachPdoOnRemove = TRUE;
m_Policies.m_GeneralPolicy.m_EnableDisableInterfaces = TRUE;
```

- Query Stop Policy

```
m_Policies.m_QueryStopPolicy.m_FailIfOpen = TRUE;
m_Policies.m_QueryStopPolicy.m_FailIfOutstandingIo = TRUE;
m_Policies.m_QueryStopPolicy.m_CancelDeviceQueue = FALSE;
m_Policies.m_QueryStopPolicy.m_CancelCurrentIrp = FALSE;
m_Policies.m_QueryStopPolicy.m_WaitForOutstandingIo
= FALSE;
m_Policies.m_QueryStopPolicy.m_UseTimeoutOnWait = FALSE;
m_Policies.m_QueryStopPolicy.m_CancelHoldQueueIfTimeout
= FALSE;
```

- Query Remove Policy

```
m_Policies.m_QueryRemovePolicy.m_FailIfOpen = TRUE;
m_Policies.m_QueryRemovePolicy.m_FailIfOutstandingIo = FALSE;
m_Policies.m_QueryRemovePolicy.m_CancelDeviceQueue = TRUE;
m_Policies.m_QueryRemovePolicy.m_CancelCurrentIrp = FALSE;
m_Policies.m_QueryRemovePolicy.m_WaitForOutstandingIo = TRUE;
m_Policies.m_QueryRemovePolicy.m_UseTimeoutOnWait = FALSE;
m_Policies.m_QueryRemovePolicy.m_CancelHoldQueueIfTimeout
= FALSE;
```

- Deferred Requests Policy

```
m_Policies.m_HoldPolicy.m_HoldIfRemovePending = TRUE;
m_Policies.m_HoldPolicy.m_HoldIfStopPending = TRUE;
m_Policies.m_HoldPolicy.m_HoldIfStopped = TRUE;
m_Policies.m_HoldPolicy.m_CancelAllOnCleanUp = TRUE;
m_Policies.m_HoldPolicy.m_CancelAllOnRemove = TRUE;
m_Policies.m_HoldPolicy.m_CancelAllOnStop = FALSE;
m_Policies.m_HoldPolicy.m_ProcessOnCancelStop = TRUE;
m_Policies.m_HoldPolicy.m_ProcessOnCancelRemove = TRUE;
```

- PDO Policy

```
m_Policies.m_ForwardToPdoPolicy.m_CallBeforeStart = TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallBeforeCancelStop
= TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallBeforeCancelRemove
= TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterQueryStop = TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterStop = TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterQueryRemove
= TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterRemove = TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterQueryPnpState
= TRUE;
m_Policies.m_ForwardToPdoPolicy.m_CallAfterSurpriseRemoval
= TRUE;
```

- Completion Policy

```
m_Policies.m_CompletedByDriverWorks.m_StartDevice = TRUE;
m_Policies.m_CompletedByDriverWorks.m_StopDevice = TRUE;
m_Policies.m_CompletedByDriverWorks.m_QueryRemove = TRUE;
m_Policies.m_CompletedByDriverWorks.m_QueryStop = TRUE;
m_Policies.m_CompletedByDriverWorks.m_CancelRemove = TRUE;
m_Policies.m_CompletedByDriverWorks.m_CancelStop = TRUE;
m_Policies.m_CompletedByDriverWorks.m_Remove = TRUE;
m_Policies.m_CompletedByDriverWorks.m_Stop = TRUE;
m_Policies.m_CompletedByDriverWorks.m_QueryPnpState = TRUE;
m_Policies.m_CompletedByDriverWorks.m_SurpriseRemoval= TRUE;
```

Class KPnpDevice Power Policies

The driver writer controls the implementation of the functionality of the base class described in previous sections by asserting policy options. To assert a particular option, a driver simply sets the corresponding flag in the data member of class **KPnpDevice** where the policies are stored. The implementation uses structures with bit fields.

The set of Power policies that a driver writer sets up for a device are logically divided into groups:

General Power Policies – This set includes some general options related to power management of the device object.

Query Power Policy – This set controls how the device is to respond to IRP_MN_QUERY_POWER.

Power Hold Policy – This set controls how the device is to respond to I/O requests for various power states.

Wait Wake Policy – This set controls when Wait_Wake requests are to be sent and when they are to be cancelled.

Device Sleep Policy – This set determines how the device is to handle I/O requests while transitioning to a lower power state.

Power Call Policy – This set determines the subset of the Power minor functions for which the base class is responsible for forwarding the IRP to the underlying PDO.

The data member of **KPnpDevice** that holds the Power policy is declared as follows:

```
POWER_POLICY m_PowerPolicies;
```

Where type POWER_POLICY has the following definition:

```
struct POWER_POLICY
{
GENERAL_POWER_POLICY m_GeneralPolicy;
QUERY_POLICY m_QueryPowerPolicy;
POWER_HOLD_POLICY m_HoldPolicy;
POWER_CALL_POLICY m_PowerCallByDriverWorks;
QUERY_POLICY m_DeviceSleepPolicy;
WAIT_WAKE_POLICY m_WaitWakePolicy;
};
```

The component structures are described below. To set a policy option, the code looks like this:

```
m_PowerPolicies.m_HoldPolicy.m_HoldIfSleepPending = TRUE;
```

Initially, no policy options are asserted. To set up the “standard power policy ”, the constructor of a subclass of **KPnpDevice** may call base class member **SetPowerPolicy**. The standard power policy utilizes many of the base class Power features, and is appropriate for many kinds of devices. However, the driver writer must examine the set of options in the standard power policy, and determine which of those are appropriate for the device to be supported. Furthermore, several options in the standard power policy require action by the driver in order to work correctly, so the developer must keep in mind the characteristics of the standard power policy at all times.

General Power Policy

This set includes some general options related to power management of the device object.

Data type GENERAL_POWER_POLICY has the following definition:

```
struct GENERAL_POWER_POLICY
{
FLAG m_PowerPolicyOwner :1;
FLAG m_UsePowerSequence :1;
FLAG m_WaitWakeEnabled :1;
FLAG m_PowerUpForIO :1;
FLAG m_PowerUpOnSO :1;
FLAG m_PowerUpOnCreate :1;
FLAG m_PowerDnOnClose :1;
FLAG m_GetDeviceCapabilities :1;
};
```

The flags have the following meanings:

- **m_PowerPolicyOwner**

  This flag tells the base class to be the power policy owner for the device.

- **m_UsePowerSequence**

  This flag tells the base class to acquire power sequence values from the bus driver before device set power IRPs.

- **m_WaitWakeEnabled**

  This flag tells the base class to enable support for IRP_MN_WAIT_WAKE.

- **m_PowerUpForIO**

  This flag tells the base class to request a device power on IRP when an I/O request is received if the device is not in the fully powered state.

- **m_PowerUpOnS0**

  This flag tells the base class to request a device power on IRP when a system set power IRP to **SystemPowerWorking** is received if the device is not in the fully powered state.

- **m_PowerUpOnCreate**

  This flag tells the base class to request a device power on IRP when an IRP_MJ_CREATE request is received if the device is not in the fully powered state.

- **m_PowerDnOnClose**

  This flag tells the base class to request a device power off IRP when an IRP_MJ_CLOSE request is received if the device is not in the fully powered state.

- **m_GetDeviceCapabilities**

  This flag tells the base class to call **GetDeviceCapabilities** in the base class' PnP dispatch handler for IRP_MN_START_DEVICE.

Device Sleep Policy

This set of policies controls base class behavior for requests to put the device in a sleep state. These policy options are stored in a structure defined as follows:

```
struct QUERY_POLICY
{
FLAG m_FailIfOpen :1;
FLAG m_FailIfOutstandingIo :1;
FLAG m_CancelDeviceQueue :1;
FLAG m_CancelCurrentIrp :1;
FLAG m_WaitForOutstandingIo :1;
FLAG m_UseTimeoutOnWait :1;
FLAG m_CancelHoldQueueIfTimeout :1;
};
```

The flags have the following meanings:

- **m_FailIfOpen**

  This flag does not apply to power management.

- **m_FailIfOutstandingIo**

  This flag does not apply to power management.

- **m_CancelCurrentIrp**

  This flag directs the base class to cancel the IRP that the device is currently processing when the device is requested to transition to a sleep state. Setting this option presumes that the device has a **StartIo** handler, and uses the device's embedded IRP queue for serializing processing of I/O requests. The base class calls member function CurrentIrp to obtain the current IRP.

- **m_CancelDeviceQueue**

  This flag directs the base class to cancel all requests in the device's embedded IRP queue when the device is requested to transition to a sleep state. Setting this option presumes that the device has a **StartIo** handler, and uses the device's embedded IRP queue for serializing processing of I/O requests.

- **m_WaitForOutstandingIo**

  This flag directs the base class to wait for any outstanding I/O to complete prior to returning to calling the PDO or returning to the I/O Manager. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below. The base class checks this policy option after processing options `m_CancelCurrentIrp` and `m_CancelDeviceQueue`.

- **m_UseTimeoutOnWait**

  If policy option `m_WaitForOutstandingIo` is set, this flag directs the base class to wait with a timeout. The timeout period is taken from data member `m_WaitTimeOut`, and is in units of 100 nanoseconds. If the wait times out, the base class attempts to cancel all outstanding requests.

Power Call Policy

This set of policy options determines which Power requests are to be forwarded to the PDO by the base class. This set is stored in a structure defined as follows:

```
struct POWER_CALL_POLICY
{
FLAG m_SetPower :1;
FLAG m_QueryPower :1;
FLAG m_WaitWake :1;
FLAG m_PowerSequence :1;
};
```

The call to the PDO is done asynchronously after the subclass has processed the request.

The flags have the following meanings:

- **m_SetPower**

  This flag directs the base class to forward requests with minor function IRP_MN_SET_POWER to the PDO after invoking the subclass handler, **OnSetPower**.

- **m_QueryPower**

  This flag directs the base class to forward requests with minor function IRP_MN_QUERY_POWER to the PDO after invoking the subclass handler, **OnQueryPower**.

- **m_WaitWake**

  This flag directs the base class to forward requests with minor function IRP_MN_WAIT_WAKE to the PDO after invoking the subclass handler, **OnWaitWake**.

- **m_PowerSequence**

  This flag directs the base class to forward requests with minor function IRP_MN_POWER_SEQUENCE to the PDO after invoking the subclass handler, **OnPowerSequence**.

Wait Wake Policy

This set of policies defines the base class behavior for supporting IRP_MN_WAIT_WAKE. In particular, these settings control when the base class requests and cancels IRP_MN_WAIT_WAKE. This set is stored in a structure defined as follows:

```
struct WAIT_WAKE_POLICY
{
FLAG m_SendDeviceSleep :1;
FLAG m_SendSystemSleep :1;
FLAG m_SendStartDevice :1;
FLAG m_CancelStopDevice :1;
FLAG m_CancelRemoveDevice :1;
FLAG m_CancelDeviceWake :1;
};
```

The following policy flags direct the base class when to request from the Power Manager that an IRP_MN_WAIT_WAKE be sent to the device stack:

- **m_SendDeviceSleep**

  This flag directs the base class to request an IRP_MN_WAIT_WAKE during the handling of IRP_MN_SET_POWER prior to powering the device to an intermediate sleep state. The IRP_MN_WAIT_WAKE request will only be sent if the device power state supports wake up.

- **m_SendSystemSleep**

  This flag directs the base class to request an IRP_MN_WAIT_WAKE during the handling of IRP_MN_SET_POWER prior to powering the system to a sleep state. The IRP_MN_WAIT_WAKE request will only be sent if the device supports wake up from the requested system power state.

- **m_SendStartDevice**

  This flag directs the base class to request an IRP_MN_WAIT_WAKE during the handling of the PnP IRP_MN_START_DEVICE. This indicates that the device can send a wake-up signal at any time.

The following policy flags direct the base class when to cancel an IRP_MN_WAIT_WAKE that was previously requested by the device:

- **m_CancelStopDevice**

  This flag indicates that a pending IRP_MN_WAIT_WAKE should be cancelled prior to completing the PnP IRP_MN_STOP_DEVICE.

- **m_CancelRemoveDevice**

  This flag indicates that a pending IRP_MN_WAIT_WAKE should be cancelled prior to completing the PnP IRP_MN_REMOVE_DEVICE.

- **m_CancelDeviceWake**

  This flag indicates that a pending IRP_MN_WAIT_WAKE should be cancelled prior to completing an IRP_MN_SET_POWER to power down the device when the requested device power state is below the minimum power state to send the wake-up signal.

Power Hold Policy

This set of policy options controls when requests are to be deferred, and how to handle the queue of deferred requests when various events occur. This set is stored in a structure defined as follows:

```
struct POWER_HOLD_POLICY
{
FLAG m_HoldIfPowerOffPending :1;
FLAG m_HoldIfSleepPending :1;
FLAG m_HoldIfSleeping :1;
FLAG m_HoldIfOff :1;
FLAG m_CancelAllOnPowerOff :1;
FLAG m_ProcessOnPowerOn :1;
FLAG m_WaitForOutstandingIO :1;
ULONGLONG m_OutstandingIOTimeout;
};
```

Class **KPnpDevice** stores deferred requests in the data member declared as follows:

```
KInterlockedList<IRP> m_HoldQueue;
```

IRPs in the hold queue are not counted as outstanding requests.

The following types of IRPs are never put in the hold queue:

- IRP_MJ_CREATE
- IRP_MJ_CLOSE
- IRP_MJ_SHUTDOWN
- IRP_MJ_CLEANUP
- IRP_MJ_PNP
- IRP_MJ_POWER

The policy flags have the following meanings:

- **m_HoldIfPowerOffPending**

  This flag directs the base class to defer requests to the hold queue if a power off operation is pending, i.e., if IRP_MN_QUERY_POWER was processed successfully and a subsequent IRP_MN_SET_POWER has yet to be received.

- **m_HoldIfSleepPending**

  This flag directs the base class to defer requests to the hold queue if a transition to an intermediate sleep state is pending, i.e., if IRP_MN_QUERY_POWER was processed successfully and a subsequent IRP_MN_SET_POWER has yet to be received.

- **m_HoldIfSleeping**

  This flag directs the base class to defer requests to the hold queue if the device is in an intermediate sleep state.

- **m_HoldIfOff**

  This flag directs the base class to defer requests to the hold queue if the device is in a power off state.

- **m_CancelAllOnPowerOff**

  This flag directs the base class to cancel all requests in the hold queue when the device receives a power off request.

- **m_ProcessOnPowerOn**

  This flag directs the base class to dispatch all requests in the hold queue to their respective handlers when the device returns to the fully powered state PowerDeviceD0.

- **m_WaitForOutstandingIO**

This flag directs the base class to wait for any outstanding I/O to complete prior to returning to calling the PDO or returning to the I/O Manager. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below. The base class checks this policy option after processing options `m_CancelCurrentIrp` and `m_CancelDeviceQueue`.

m_OutstandingIOTimeout

This data member is the timeout period for policy option `m_WaitForOutstandingIo`, and is in units of 100 nanoseconds. If the wait times out, the base class attempts to cancel all outstanding requests.

Query Power Policy

Policy options for IRP_MN_QUERY_POWER are stored in a structure defined as follows:

```
struct QUERY_POLICY
{
FLAG m_FailIfOpen :1;
FLAG m_FailIfOutstandingIo :1;
FLAG m_CancelDeviceQueue :1;
FLAG m_CancelCurrentIrp :1;
FLAG m_WaitForOutstandingIo :1;
FLAG m_UseTimeoutOnWait :1;
ULONGLONG m_WaitTimeOut;
};
```

The system will send an IRP_MN_QUERY_POWER request to all devices prior to powering down the system.

The flags have the following meanings:

- m_FailIfOpen

  This flag directs the base class to fail the request immediately with STATUS_DEVICE_BUSY if there are any handles open. The base class counts open handles by incrementing a counter when the handler for IRP_MJ_CREATE returns STATUS_SUCCESS, and decrementing it when the handler for IRP_MJ_CLOSE returns STATUS_SUCCESS.

- **m_FailIfOutstandingIo**

  This flag directs the base class to fail the request immediately with STATUS_DEVICE_BUSY if there are any outstanding requests. If the driver sets this option, it must follow the guidelines for counting outstanding requests (see below).

- **m_CancelCurrentIrp**

  This flag directs the base class to cancel the IRP that the device is currently processing. Setting this option presumes that the device has a **StartIo** handler, and uses the device's embedded IRP queue for serializing processing of I/O requests. The base class calls member function CurrentIrp to obtain the current IRP.

- **m_CancelDeviceQueue**

  This flag directs the base class to cancel all requests in the device's embedded IRP queue. Setting this option presumes that the device has a **StartIo** handler, and uses the device's embedded IRP queue for serializing processing of I/O requests.

- **m_WaitForOutstandingIo**

  This flag directs the base class to wait for any outstanding I/O to complete prior to returning to calling the PDO or returning to the I/O Manager. If the driver sets this option, it must follow the guidelines for counting outstanding requests, which are detailed below. The base class checks this policy option after processing options `m_CancelCurrentIrp` and `m_CancelDeviceQueue`.

- **m_UseTimeoutOnWait**

  If policy option `m_WaitForOutstandingIo` is set, this flag directs the base class to wait with a timeout. The timeout period is taken from data member `m_WaitTimeOut`, and is in units of 100 nanoseconds. If the wait times out, the base class attempts to cancel all outstanding requests.

Standard Power Management Policy

When an instance of **KPnpDevice** is first constructed, all policy options are FALSE. A driver requests the standard Power Management policy by calling member function **KPnpDevice::SetPowerPolicy**. The BOOLEAN parameter **supportWaitWake** is used to enable policies for standard IRP_MN_WAIT_WAKE support. The settings of the standard policy are as follows:

- General Policy:

```
m_PowerPolicies.m_GeneralPolicy.m_PowerPolicyOwner = TRUE;
m_PowerPolicies.m_GeneralPolicy.m_UsePowerSequence = FALSE;
m_PowerPolicies.m_GeneralPolicy.m_WaitWakeEnabled
= supportWaitWake ;
m_PowerPolicies.m_GeneralPolicy.m_PowerUpForIO = TRUE;
m_PowerPolicies.m_GeneralPolicy.m_PowerUpOnS0 = FALSE;
m_PowerPolicies.m_GeneralPolicy.m_PowerDnOnClose = FALSE;
m_PowerPolicies.m_GeneralPolicy.m_PowerUpOnCreate = TRUE;
m_PowerPolicies.m_GeneralPolicy.m_GetDeviceCapabilities
= TRUE;
```

- Query Power Policy:

```
m_PowerPolicies.m_QueryPowerPolicy.m_FailIfOpen = TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_FailIfOutstandingIo
= TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_CancelDeviceQueue
= TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_CancelCurrentIrp
= TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_WaitForOutstandingIo =
TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_UseTimeoutOnWait = TRUE;
m_PowerPolicies.m_QueryPowerPolicy.m_CancelHoldQueueIfTimeout
= TRUE;
```

- Hold Policy:

```
m_PowerPolicies.m_HoldPolicy.m_HoldIfPowerOffPending = TRUE;
m_PowerPolicies.m_HoldPolicy.m_HoldIfSleepPending = TRUE;
m_PowerPolicies.m_HoldPolicy.m_HoldIfSleeping = FALSE;
m_PowerPolicies.m_HoldPolicy.m_HoldIfOff = FALSE;
m_PowerPolicies.m_HoldPolicy.m_CancelAllOnPowerOff = FALSE;
m_PowerPolicies.m_HoldPolicy.m_ProcessOnPowerOn = TRUE;
m_PowerPolicies.m_HoldPolicy.m_WaitForOutstandingIO = TRUE;
```

- Power Call Policy:

```
m_PowerPolicies.m_PowerCallByDriverWorks.m_SetPower = TRUE;
m_PowerPolicies.m_PowerCallByDriverWorks.m_QueryPower = TRUE;
m_PowerPolicies.m_PowerCallByDriverWorks.m_WaitWake
= supportWaitWake;
m_PowerPolicies.m_PowerCallByDriverWorks.m_PowerSequence
= FALSE;
```

- Device Sleep Policy:

```
m_PowerPolicies.m_DeviceSleepPolicy.m_FailIfOpen = TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.m_FailIfOutstandingIo
= TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.m_CancelDeviceQueue
= TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.m_CancelCurrentIrp
= TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.m_WaitForOutstandingIo
= TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.m_UseTimeoutOnWait
= TRUE;
m_PowerPolicies.m_DeviceSleepPolicy.
m_CancelHoldQueueIfTimeout = TRUE;
```

- Wait/Wake Policy:

```
m_PowerPolicies.m_WaitWakePolicy.m_SendDeviceSleep
= supportWaitWake;
m_PowerPolicies.m_WaitWakePolicy.m_SendSystemSleep
= supportWaitWake;
m_PowerPolicies.m_WaitWakePolicy.m_SendStartDevice = FALSE;
m_PowerPolicies.m_WaitWakePolicy.m_CancelStopDevice
= supportWaitWake;
m_PowerPolicies.m_WaitWakePolicy.m_CancelRemoveDevice
= supportWaitWake;
m_PowerPolicies.m_WaitWakePolicy.m_CancelDeviceWake
= supportWaitWake;
```

Guidelines for Counting Outstanding Requests

Many policy options depend on the ability of the base class to maintain a count of outstanding requests, i.e., the number of requests which have been delivered to the driver, but for which the driver still has work to do. Requests that are on the hold queue do not count as outstanding requests.

Class **KPnpDevice** automatically increments its internal counter of outstanding requests each time it dispatches any IRP to a handler. However, the base class *relies on the subclass* to assist in maintaining the count of outstanding requests. Specifically, the subclass must inform the base class each time it finishes processing a request.

There are three ways by which a driver can be considered finished with a given IRP.

- It completes the IRP.
- It passes the IRP to a lower device, without having set a completion routine.
- The IRP is passed to a lower device with a completion routine, and that completion routine executes and returns a status other than STATUS_MORE_PROCESSING_REQUIRED.

In other words, the driver is finished with the IRP when it has no commitments to further process it.

It is quite simple for the subclass to provide this support and thereby gain the advantage of automatic implementation of all policy options related to outstanding requests. The guidelines are as follows.

- **Completing an IRP.** To maintain the outstanding request count when completing an IRP, simply make the following substitutions:

Instead of . . .	Use . . .
KIrp:: Complete	KIrp:: PnpComplete
KDevice:: NextIrp	KPnpDevice::PnpNextIrp
KDeviceQueue:: CleanUp	KDeviceQueue:: PnpCleanUp
KDriverManagedQueue:: NextIrp	KDriverManagedQueue:: PnpNextIrp

In each case in the above table, the first parameter of the function in the right hand column is a pointer to the instance of a **KPnpDevice** subclass for which outstanding IRPs are being counted. For example:

```
NTSTATUS MyDevice::Read(KIrp I)
{
// process the IRP
. . .
return I.PnpComplete(this, status);
}
```

Note that **KIrp::PnpComplete** is used rather than **KIrp::Complete**, and that a pointer to the device for which requests are counted (**this**) is passed as the first parameter.

- **Passing the IRP to a lower device without a completion routine.** To maintain the outstanding request count when passing an IRP to a lower device (no completion routine set), simply make the following substitutions:

Instead of . . .	Use . . .
KLowerDevice::Call	KLowerDevice::PnpCall
KLowerDevice:: PowerCall	KPnpLowerDevice::PnpPowerCall

In each case in the above table, the first parameter of the function in the right hand column is a pointer to the instance of a **KPnpDevice** subclass for which outstanding IRPs are being counted. For example:

```
NTSTATUS MyDevice::Read(KIrp I)
{
// pass the IRP to a lower device

return pLowerDevice->PnpCall(this, I);
}
```

Note that **KLowerDevice::PnpCall** is used rather than **KLowerDevice::Call**, and that a pointer to the device for which requests are counted (**`this`**) is passed as the first parameter.

- **IRPs with completion routines.** If the IRP has a completion routine and the completion does *not* return STATUS_MORE_PROCESSING_REQUIRED, then simply call **KPnpDevice::DecrementOutstandingRequestCount** somewhere inside the completion routine. For example.

```
NTSTATUS MyDevice::ACompletionRoutine(KIrp I)
{
if (I->PendingReturned)
I.MarkPending();
. . .
DecrementOutstandingRequestCount();
return STATUS_SUCCESS;
}
```

Class KPnpLowerDevice

Overview

Class **KPnpLowerDevice** provides a model of the Physical Device Object, or PDO, for use by a WDM device driver. The PDO is the operating system's representation of the physical device, which it uses for configuration and power management.

When an instance of **KPnpLowerDevice** is created or initialized by a driver, it attaches a device object to the Physical Device Object (PDO). The attachment operation returns another system device object address, which points to the device object which was actually attached. This is referred to as the Top Of Stack device. The Top of Stack device might be different from the PDO. Device attachments form a singly linked list, so at most one device can attach to a second. If the Top of Stack device is already attached to the PDO, then a second device that attempts to attach to the PDO will instead attach to the Top of Stack device.

When the system calls the **AddDevice** entry point of a minidriver, the minidriver usually creates a Functional Device Object (FDO) that it uses to model and control the physical device. The FDO is usually represented by an instance of a subclass of **KPnpDevice**. The driver writer typically embeds an instance of **KPnpLowerDevice** (or subclass thereof) in the FDO to represent the PDO. Note that there is a default constructor for **KPnpLowerDevice** and a member function **Initialize** which make it unnecessary to initialize the lower device object in the constructor list; rather, the constructor can just call **Initialize**.

The scenario is different for certain minidrivers, such as **HID** minidrivers. In this case, the class driver above the minidriver creates the FDO and attaches the PDO. Therefore, the minidriver must not create a second system device object to represent the FDO and must not attempt to attach any additional device objects to the PDO. The class driver above the minidriver supplies the system device object pointers for the PDO and FDO to the minidriver in the FDO device extension. The minidriver accepts these pointers and uses them to passively initialize an instance of **KPnpLowerDevice** to represent the PDO. We say ‘passively’ to suggest it merely stores the supplied device-object pointers, rather than making system calls to allocate or attach new device objects. The passive initialization is accomplished using a second form of member function **Initialize**.

The property retrieval functions utilize **IoGetDeviceProperty**.

Plug and Play Resource Management

In WDM systems, the operating system takes the responsibility of assigning hardware resources to devices. Types of hardware resources include address ranges (either in the I/O space or in the memory space), interrupt request signals (IRQs), and legacy system DMA channels. The operating system analyzes the requirements of the devices that are present and dynamically configures the system to best accommodate all devices. This differs markedly from the NT 4.0 style of resource management, in which the operating system was a passive arbitrator of system resources and required each driver to actively request usage of specific resources.

WDM specifies that resources be associated with a device when the device is started (i.e., it receives IRP_MN_START_DEVICE), and disassociated when the device is stopped (i.e., it receives IRP_MN_STOP_DEVICE). These events correspond to the invocation of **KPnpDevice::OnStartDevice** and **KPnpDevice::OnStopDevice.**

When a device is started, the device is sent an IRP containing a pointer to a CM_RESOURCE_LIST (see **KIrp::AllocatedResources** and **KIrp::TranslatedResources)**, which the driver can use to generate a **KResourceAssignment** object. This object can then be queried for specific addresses, IRQs, etc.

The operating system can stop the device for dynamic reconfiguration. When **OnStopDevice** is called, the driver must disconnect interrupts, destroy memory or I/O ranges, and otherwise disassociate itself from any resources that were assigned to it when it was started.

Making a WDM Driver Callable by a VxD

Overview

Class **KVxDInterface** enables a WDM driver running on Windows 98 to receive VxD control messages. This is the recommended way for WDM drivers to communicate with VxDs. The constructor calls a VMM service that adds a Device Data Block (DDB) to the system's VxD list. The DDB contains a field that points to a control procedure. The control procedure is the function that the system calls to deliver a VxD control message.

A VxD interface is sometimes implemented in a special "mapper" driver. The mapper driver provides the interface to the VxD and passes requests to a pure WDM driver. The pure WDM driver runs unchanged on Windows 2000. The advantage to this approach is that the WDM driver does not need to carry the extra VxD code on platforms where it is not used. In any case, the constructor checks the platform on which it is running and fails if not running on Windows 98. Therefore, it is not strictly necessary to create a mapper; a single driver that detects the platform on which it is running can conditionally create instances of this object as necessary.

Suppose you have a device that needs to export a VxD interface:

```
class MyDevice : public KPnpDevice {
. . .

protected:
// declare the object
KVxDInterface m_vxd;
// declare handler
MEMBER_VXDCONTROLDISPATCHER ( MyDevice, VxDControl )
. . .
};
```

In the class constructor, you initialize the object:

```
MyDevice::MyDevice( . . . args . . . )
{

BOOLEAN Success;
Success = m_vxd.Initialize(
"MYDEVICE",
LinkTo(VxDControl),
this
);
}
```

And the control message handler looks like this:

```
ULONG MyDevice::VxDControl(
ULONG ControlMessage,
ULONG Edi,
ULONG Esi,
ULONG Ebx,
ULONG Edx,
ULONG Ecx,
ULONG* pCarryBitReturn
)
{
// handle the control message
// clear carry when returning to caller
*pCarryBitReturn = 0;
// store value in EAX on retrun to caller
return value;
}
```

WDM Device Interfaces

The WDM introduces the concept of *device interfaces*. A device interface is an object that provides a defined set of IRP-based services and is instantiated in a particular device object.

The abstract functionality of a device interface is referred to as an *interface class*. For example, suppose an application needs the services of a Toaster, but does not know the name of a particular device that provides the standard services that all Toaster devices support. On WDM systems, the application can query the operating system for the names (actually, symbolic links) of all devices that have registered instances of the Toaster interface class.

In order to query for the devices that provide a particular interface class, the application must know the GUID that identifies the class. Once an application has obtained a symbolic link to a device that supports the interface class, it can then use **CreateFile** to gain access to the device.

At the driver level, new system services enable drivers to declare that an interface class is manifested in a particular device. When a WDM driver creates a device, it can register one or more device interfaces for each interface class the device provides. To register a device interface, the driver calls **KPnpDevice::RegisterInterface**, passing the GUID that identifies the interface class. This service creates and returns a symbolic link that uniquely identifies the device interface. The system composes the symbolic link from the class GUID and a serial number that it maintains in the registry and increments each time a new interface of that class is registered. The device object to which the symbolic link refers is the physical device object (PDO), not the functional device object (FDO) created when the driver calls **KPnpDevice::RegisterInterface.** This constructor of **KPnpDevice** makes this call if the caller passes the GUID of a device interface class. In order to receive IRPs sent by an application that opens the symbolic link, the driver *must* attach the FDO to the PDO, as is normal. Prior to WDM, drivers were responsible for naming the symbolic links of each device they created. In the WDM environment, the symbolic link is to the underlying PDO and its name is determined by the system, based on the GUID of the interface class.

When a device is activated (i.e., it receives IRP_MJ_PNP with the minor function IRP_MN_START_DEVICE), it enables its interfaces by calling **KPnpDevice::EnableInterfaces**. Similarly, when the device is stopped or removed, it must call **KPnpDevice::DisableInterfaces.** Policy settings enable automatic implementation of this functionality.

Using Device Interfaces from the Application Level

DriverWorks provides two classes for application level support of device interfaces: **CDeviceInterfaceClass** and **CDeviceInterface**. These classes are declared and implemented in file `devintf.h`, which resides in subdirectory **.\include** of the **DriverWorks** installation.

Class **CDeviceInterfaceClass** wraps a device information set containing information about all device interfaces of a particular class.

An application can use an instance of **CDeviceInterfaceClass** to obtain one or more instances of **CDeviceInterface**. Class **CDeviceInterface** abstracts a single device interface. Its member function **DevicePath()** returns a pointer to a pathname, which can be passed to **CreateFile** to open the device.

Here is a small sample showing the most basic use of these classes.

```
extern GUID TestGuid;

HANDLE OpenByInterface(
GUID* pClassGuid,
DWORD instance,
PDWORD pError
)
{
CDeviceInterfaceClass DevClass(pClassGuid, pError);
if (*pError != ERROR_SUCCESS)
    return INVALID_HANDLE_VALUE;

CDeviceInterface DevInterface(&DevClass,
instance,
pError);
if (*pError != ERROR_SUCCESS)
    return INVALID_HANDLE_VALUE;
cout   << "The device path is "
<< DevInterface.DevicePath() << endl;
```

```
HANDLE hDev;
hDev = CreateFile(
DevInterface.DevicePath(),
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL,
OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,
NULL
);
if (hDev == INVALID_HANDLE_VALUE)
    *pError = GetLastError();
return hDev;
}
```

WDM Power Management

Power management is an important aspect of WDM drivers. Its goals are similar to those of the Plug and Play subsystem. That is, it contributes to a responsive, adaptive, and dynamic machine environment.

The Power IRP is similar to IRP_MJ_PNP, in that a single major function, IRP_MJ_POWER is overloaded with a set of minor functions, each having its own independent purpose. The four subfunctions of the power IRP are:

- IRP_MN_SET_POWER
- IRP_MN_WAIT_WAKE
- IRP_MN_POWER_SEQUENCE
- IRP_MN_QUERY_POWER

IRP_MN_SET_POWER is sent by the system to a device to inform it of a system or device power state change (see **KIrp::PowerStateTyp**e and **KIrp::PowerStateSetting**.) In response to a system power state change, most drivers request a device power state change which results in an **IRP_MN_SET_POWER** sent to the device for a device power state change.

Device power states range from D0 (fully awake) to D3 (fully off). If the driver uses the default power policy, the system and device **IRP_MN_SET_POWER** requests will be handled by the base class **KPnpDevice**. The subclass can override **KPnpDevice::OnDeviceSleep** to save any required hardware context that will be lost in the low power state or to disconnect interrupts. The subclass can override **KPnpDevice::OnDevicePowerUp** to restore any required hardware context that was saved prior to going to a low power state or to re-connect interrupts. The handler for this IRP is **KPnpLowerDevice::OnSetPower**. In rare cases, the driver can override this if it needs specialized power management behavior. Whenever passing a power IRP to the PDO, the driver must use **KPnpLowerDevice::PnpPowerCallDriver**. The system requires a special function because it must serialize power operations.

IRP_MN_WAIT_WAKE is never sent to most drivers. Rather, a device is responsible for requesting that the system send this IRP to the device stack under certain conditions and for cancelling a pending **IRP_MN_WAIT_WAKE** request. A driver that controls a device that can wake-up the system can request this IRP when the device is capable of wake-up (typically when the system or device is transitioning to a low power state.) When a device requests this IRP, the IRP will be seen by the entire device stack. When the IRP is received by the device, it must forward it to the PDO. The bus driver will hold the IRP pending until a bus specific wake-up signal is received. Then the IRP will be completed. A good example of this type of driver is a modem that is to wake the system when it rings. When the bus driver observes the assertion of the bus wake-up signal, it determines which device is requesting service and completes the WAIT_WAKE IRP for that device. A callback function supplied when requesting the IRP will be invoked and typically requests an **IRP_MN_SET_POWER** to change the device power state to D0. Class **KPnpDevice** implements most of the required behavior and is controlled by power policy settings.

IRP_MN_POWER_SEQUENCE enables a driver to track power state changes and avoid unnecessary operations. This is a means to optimize performance. More documentation is available in the DDK.

IRP_MN_QUERY_POWER is usually, but not always, sent by the system to a device prior to **IRP_MN_SET_POWER** for a system power state change to a lower power state. This request gives the device an opportunity to either refuse the power state change or to prepare itself for a subsequent change in power. If the driver uses the default power policy, the system **IRP_MN_QUERY_POWER** requests will be handled by the base class **KPnpDevice** based on power policy settings. The handler for this IRP is **KPnpLowerDevice::OnQueryPower**. In rare cases, the driver can override this if it needs specialized power management behavior.

Each class derived from **KPnpDevice** must override virtual member function **DefaultPower**, a default handler for the power IRP. The base class handlers for all the power IRP minor functions invoke this member. Here is the code for a simple default handler:

```
NTSTATUS MyDevice::DefaultPower(KIrp I)
{
I.IndicatePowerIrpProcessed();
I.CopyParametersDown();
return m_Pdo.PnpPowerCall(this, I);
}
```

Adding WMI Support to a Driver

Adding WMI support to a driver is a multi-step process. Each stage in that process is detailed in the following sections:

Identify or Create the CIM Classes for Data that the Driver will Publish

The system defines a set of generic block classes for kernel mode drivers, and you can find their descriptions in the DDK under `wmicore.mof`. If your driver publishes custom blocks that are not in this file, then your driver must include a MOF file that describes those blocks.

Creating a simple MOF description for a driver's exported data is only slightly more difficult than defining a C-style struct. For example, suppose you want to have a CIM class that publishes data about the operation of your driver, and the driver has attributes "BuffersProcessed" and "LineErrorsDetected". In addition, suppose that these quantities can be expressed with 32-bit signed integers. Then the CIM class might look like this:

```
[
WMI, Dynamic, Provider ("WMIProv"),
guid("{67526BDD-C27E-22E2-CEA9-11B1DA17CF3E}"),
GuidName1("MY_MACHINE_MEASUREMENTS_GUID"),
HeaderName("MY_MACHINE_MEASUREMENTS"),
Description("Measurement information for my machine")
]
class MyMachineMeasurementInformation : MyMachine
{
boolean Active;
[
key
]
string InstanceName;
[
WmiDataId(1),
Description("number of buffers the driver has handled"),
Read
]
sint32 BuffersProcessed;
[
WmiDataId(2),
Description("number of line errors the driver has detected"),
Read
]
sint32 LineErrorsDetected;
};
```

The first section in square brackets ([ ]) encloses the class attributes. All WDM providers must have `WMI, Dynamic, Provider("WMIProv")`, followed by the GUID. You can use `guidgen.exe` to generate a GUID for each new class of block that your driver defines. The remaining class attributes are recommended.

The class definition itself resembles a C++ class definition. The keyword class is followed by the class name (**MyMachineMeasurementInformation**), which is followed by a colon (:) and the base class name. The base class may be an empty class whose purpose is to provide a unique namespace for all the classes associated with the device.

Inside the curly braces ({ }) are the item identifiers. The first two, **Active** and **InstanceName** are required of all WDM providers. All classes used by WDM providers have identical code for these two items.

Following the required fields are the item identifiers for the class. The item qualifiers are enclosed in square brackets, and precede the declaration of the item. In this sample, both items are read-only. Note that qualifier **WmiDataId** is required in order to enable WMI to access the fields successfully. The sample class above has no method items, but if it did, it would use **WmiMethodId** rather the **WmiDataId** for each of the method items.

Define C Structures that Correspond to the CIM Classes

A driver uses instances of such C structures to store data that will be read from and written to by WMI clients. For CIM classes that exist in the system file wmicore.mof, the corresponding C structures are defined in `wmidata.h`. You can examine the corresponding classes in these two files to get an idea of the relationship.

The mapping from CIM fields to C/C++ fields is straightforward, as shown in the following table:

CIM Type	C/C++ Type
String	class KWmiString
boolean	BOOLEAN
sint8	CHAR
uint8	UCHAR
sint16	SHORT
uint16	USHORT
sint32	LONG
uint32	ULONG
sint64	LONGLONG
uint64	ULONGLONG
datetime	WCHAR[25]

It is critical to remember that field alignment is assumed to be on eight-byte boundaries. Accordingly, you may need to use #pragma pack to ensure the correct packing.

Using the CIM definition we gave earlier, the corresponding C struct would be declared as follows:

```
#pragma pack(push, 8)
struct MyMachineMeasurementBlock
{
   LONG m_BuffersProcessed;
   LONG m_LineErrorsDetected;
};
#pragma pack(pop)
```

WMI supports variable-sized blocks, which do not map neatly onto a C structure. In such cases, the driver writer specializes the methods that query and modify the block. For blocks that have string items, DriverWorks provides class **KWmiString**. This class makes it easier to handle WMI strings.

Add Instances of Template Class KWmiDataBlock

Class **KWmiDataBlock** is a template class that wraps around the C structure that you define to correspond to a **WMI block**. Typically, you embed one or more instances of **KWmiDataBlock** as data members in the class that you derive from **KPnpDevice**.

KWmiDataBlock is a template with three parameters:

- The type (usually a struct) that models the CIM class of the block,
- The method class; i.e., the class owning the methods associated with the block. This is usually the class derived from **KPnpDevice**. For blocks with no methods, this can be unspecified, as the default parameter will suffice.
- The framework interface class. By default, this is **KStdWmiInterface**. Almost all drivers will leave this unspecified, and take the default. It is provided so that drivers that do not use the standard DriverWorks framework classes can still use the WMI classes.

Note: When you embed a block in a device object, you must invoke the constructor of the block in the member initializer list of the device object.

Suppose you have a class **MyDeviceClass** derived from **KPnpDevice**, and you have a WMI block modeled by struct **MyMachineMeasurementBlock** (from the earlier example). The device class declaration looks like this:

```
class MyDeviceClass : public KPnpDevice
{
// . . . usual stuff . . .
// declare the WMI block as a data member
// of the device class
   KWmiDataBlock<MyMachineMeasurementBlock>
m_MeasurementBlock;
};
```

It is perfectly allowable to have multiple blocks of the same type or of different types in a device object.

When a block has methods associated with it, a second template parameter is required (as described in the next section).

Identify the Method Object and Declare the Array of Method Members

This applies to blocks that have methods.

WMI methods are procedures in your driver that WMI clients can call. A method is associated with a particular WMI block. These methods can accept input parameters and provide output parameters.

The DriverWorks implementation of WMI requires that WMI methods be mapped to member functions of some class. For a given block type, all methods are mapped to members of the same class. You specify the class that implements the methods as the second parameter to template **KWmiDataBlock**. While it is entirely up to the driver writer which class is to serve as the method class, the device class is usually the most natural.

To fully specify the object that is to supply the methods for a block, the object type (named in the block's template parameter list) is necessary but not sufficient. In addition, your driver must supply a pointer to an instance of the method object type. The driver passes this pointer to the block's constructor.

If the parameter that specifies the method object is omitted in the call to the constructor, the default value of NULL directs DriverWorks to use the device class object as the method object.

To implement methods, there are five steps:

1. For custom MOF files, declare the methods in the class definitions. An example is shown below. The DDK provides complete documentation of the syntax.
2. Identify the class that will implement the methods. Normally, this is your device class, i.e., the class you derived from **KPnpDevice**.
3. For each block type, declare the array of method functions. Details of how to do this are given in the next section.
4. Add code to call **KWmiContext::SetMethods** when the device is started, stopped, or removed.
5. Declare and implement the member functions that serve as the WMI methods.

Add Code to Register and Deregister a Device's WMI Blocks

A driver registers the blocks for a device when the device is started, and deregisters them when the device is stopped or removed.

Member functions of template class **KWmiContext** implement the registration and deregistration operations. DriverWorks automatically creates an instance of this class when WMI blocks are constructed. In the normal case, when a driver creates instances of classes derived from **KPnpDevice**, public data member **KPnpDevice::m_Wmi** points to the **KWmiContext** object for the device. As a result, it is simple to invoke the registration and deregistration functions inside the handlers for IRP_MN_START_DEVICE (**OnStartDevice**), IRP_MN_STOP_DEVICE (**OnStopDevice**), and IRP_MN_REMOVE_DEVICE (**OnRemoveDevice**).

Here is an example:

```
NTSTATUS MyDevice::OnStartDevice(KIrp I)
{
   // . . . other usual stuff . . .
   // Tell DriverWorks about the blocks for this device
   status = m_Wmi->Register(*this, L"MofResource");
}
NTSTATUS MyDevice::OnRemoveDevice(KIrp I)
{
// . . . other usual stuff . . .
// Tell DriverWorks we are done with the
// blocks for this device
   m_Wmi->Deregister();
}
```

Add a Handler For IRP_MJ_SYSTEM_CONTROL

Whenever an application makes a WMI request to a kernel mode provider, the WMI subsystem translates the request to an IRP of type IRP_MJ_SYSTEM_CONTROL. Therefore, any driver that is to support WMI must have a handler for this IRP.

To add a handler for this IRP to your driver, edit the `function.h` file for the driver, and add

```
#define DRIVER_FUNCTION_SYSTEM_CONTROL
```

If you are not using DEVMEMBER_DISPATCHERS inside the declaration of your device class(es), then you must add a declaration of the member function, like this:

```
NTSTATUS SystemControl(KIrp I);
```

Finally, add the code for the handler itself. Here is what it should look like:

```
NTSTATUS MyDevice::SystemControl(KIrp I)
{
   if ( m_Wmi != NULL )
   {
     return m_Wmi->DispatchSystemControl(*this, I);
   }
   else
   {
     I.ForceReuseOfCurrentStackLocationInCalldown();
     return m_Lower.PnpCall(this, I);
   }
}
```

In the event that no WMI blocks were instantiated, **KPnpDevice** data member `m_Wmi` will be NULL, so it is a good idea to check for this. In the normal case, DriverWorks takes responsibility for taking the appropriate action. The result could be a call to a member function, such as Query, of some block, or the invocation of a WMI method.

Specialize Members of Template Instances as Necessary

DriverWorks provides automatic dispatching of IRP_MJ_SYSTEM_CONTROL. The following table shows the mapping of certain minor function codes to member functions of class **KWmiDataBlock**:

Minor Function Code	Member Function Invoked
IRP_MJ_QUERY_ALL_DATA and IRP_MJ_QUERY_SINGLE_INSTANCE	Query
IRP_MN_CHANGE_SINGLE_INSTANCE	Set
IRP_MN_CHANGE_SINGLE_ITEM	SetItem

In some cases, **KWmiDataBlock** requires no additional code for proper operation. In other cases, the driver writer must provide specializations of the member functions.

The implementation of **SetItem** must always be specialized, because the template has no knowledge of the individual data items inside the block.

For most blocks that have a fixed size, the template's implementations of **Set** and **Query** are sufficient, and the driver writer does not have to provide any additional code. This statement is based on the assumption that the driver actively maintains the data held in the structures wrapped by the **KWmiDataBlock** instances. If not, then specialization is required.

Blocks of variable size, such as those that contain strings, require specialization of **Query** and Set.

Modify the Build Procedure

If your WMI blocks are not of a system defined CIM class, then the MOF file that you created to define the custom CIM class(es) must be compiled and attached to your driver as a resource.

Normally, drivers already have a resource (with extension .rc) for the purpose of specifying the driver version information. The MOF resource declaration in the resource file has the following form:

```
<tag> MOFDATA <name of compiled MOF file>
```

where <tag> is the name that you pass to Register when you register your blocks. A line of this form must appear in the resource file.

Here is an example:

```
MofResource MOFDATA myblocks.bmf
```

In this example, the tag is `MofResource`, and the name of the compiled MOF file is `myblocks.bmf`.

Using the DDK's BUILD Utility

If you build your driver with BUILD.EXE, then you just need to add the .mof file to the SOURCES= list in the SOURCES file, e.g.:

```
SOURCES=mymain.cpp myaux.cpp myblocks.mof
```

The DDK makefiles provide all the required logic for correct processing.

Using the Visual Studio Environment

If you are using Microsoft Visual Studio 6.0 to build your driver, then follow these steps:

- Add the MOF file to the project. From the Project menu, select **Add To Project | Files**, then specify or browse to the MOF file. Click **OK**.
- (optional) In the Workspace window, drag the MOF file to the Resource Files folder.
- From the **Project** menu, select **Settings**. In the left pane of the Settings dialog, locate and select the MOF file. Once you have selected it, click on the **Custom Build** tab in the right pane.
- Set the command for the custom build step to this line:

  ```
  $(BASEDIR)\bin\x86\mofcomp -B:$(TargetName).bmf
  $(InputPath)
  ```

- Set the outputs for the custom build step to:

  ```
  $(TargetName).bmf
  ```

Note: You must make the changes to the project settings for both the Checked and Free configurations of the driver.

Classes for Human Interface Device Support

The Human Interface Device (HID) specification describes a protocol for transmitting data between a computer and a class of peripheral devices. The HID specification was originally developed as a part of the USB specification, but has been adopted as a bus independent protocol for WDM.

The intent is that devices using the HID protocol are devices under human control, such as mice, keyboards, and joysticks. These devices share the characteristics of requiring low bandwidth, being event driven, and having structured input formats. For each device, there is a *report descriptor* that describes the format and content of data that the device generates. Using the report descriptor, the operating system can parse input from the device and relay it to applications that have requested it.

Although it is primarily used for input devices, HID allows transfers both to and from a peripheral device.

At the center of the HID architecture of WDM is HIDCLASS.SYS, which provides the interfaces used by applications to access HID devices. Subordinate to HIDCLASS are the HID minidrivers. Each minidriver responds to requests from HIDCLASS to supply reports from a HID device to the system. These requests enter the minidriver as IRPs having major functions IRP_MJ_PNP, IRP_MJ_POWER and IRP_MJ_INTERNAL_DEVICE_CONTROL. The architecture limits the work of the minidriver to supporting a small, well-defined set of operations.

DriverWorks provides two classes for developing HID minidrivers: **KHidMiniDriver** and **KHidDevice**.

Writing an HID Minidriver

Class KHidMiniDriver

Class **KHidMiniDriver** is the class you use to form the framework of a HID minidriver. However, you do not use this class directly; you must define a subclass.

KHidMiniDriver is a small and simple class. It overrides **KDriver** virtual member function **DriverEntry**, so its subclasses are *not* required to do so.

The other overridable virtual member of class **KHidMiniDriver** is **Register**, which registers the driver with HIDCLASS.SYS. **DriverEntry** calls this member function. While the member function is overridable, the base class implementation should be sufficient for most drivers.

The important virtual member function for subclasses of class **KHidMiniDriver** is **AddDevice**. A subclass *must* override this member in order to respond to obtain the Functional Device Object (FDO) pointer that HIDCLASS provides for its devices. Remember to #define DRIVER_FUNCTION_ADD_DEVICE in `function.h`.

The main module of a HID minidriver will have a code sequence like the following:

```
#define VDW_MAIN
// Note: HID minidrivers include khid.h
//rather than vdw.h
#include <khid.h>

#include "myhiddriver.h" // the driver class
#include "myhiddevice.h"/ / the device class

//////////////////////////////////////////////////
// Begin INIT section code
#pragma code_seg("INIT")
DECLARE_DRIVER_CLASS(MyHidDriver, NULL)
```

Class KHidDevice

Class **KHidDevice**, a subclass of **KPnpDevice**, is a model for a device that provides HID reports to the system. Its purposes are to:

- Provide a framework for dispatching internal device control requests and Plug and Play requests to overridable virtual member functions
- Implement default processing of many requests that all HID minidrivers must handle

KHidDevice is used as a base class; you must define a subclass in order to make use of it.

Unlike most subclasses of **KDevice**, creating an instance of **KHidDevice** does not create a new system device object. Rather, the instance inherits a system device object (the FDO) from HIDCLASS.SYS. When a HID minidriver registers with HIDCLASS, HIDCLASS takes the driver's **AddDevice** entry point so it can control device creation. HIDCLASS defines a structure for the device extension of the system device object it creates, enabling the HID minidriver to locate the PDO and top of stack devices. This is all handled transparently by **KHidDevice**. The key is simply to pass the FDO pointer that is passed to **AddDevice** to the constructor of the subclass of **KHidDevice**.

The class implements three Plug and Play handlers that override virtual member functions of **KPnpDevice**, namely **OnQueryId**, **OnRemoveDevice**, and **OnQueryCapabilities**. In each case, the handler provides default processing for the request. This is sufficient for most HID devices, but the subclass can override any or all of them.

The class provides dispatching of all the internal device control requests that HIDCLASS sends to a minidriver, and, for some of them, the class implements useful default processing. For the remainder, the base class implementation simply passes the request down the device stack. The implementor of the subclass must examine the documentation for each of the virtual member functions and determine which ones the subclass should override.

Universal Serial Bus (USB) Drivers

The following sections describe USB drivers.

Classes for USB Support

DriverWorks supports development of drivers that are USB clients, i.e., drivers that need to send requests over a USB to some device.

From the client driver's perspective, any USB device is conceptually comprised of three classes of components:

- **Logical Device**. This is the logical component with which the USB bus driver communicates in order to configure and control the device. It corresponds to the default control pipe or "endpoint 0" as described in the USB specification. All USB devices support a common set of functions accessible via commands to the Logical Device layer of the USB protocol stack. These functions include access to the device descriptor and setting the configuration.

  Class **KUsbLowerDevice** abstracts this functionality. By instancing this class, a device driver creates an interface to the upper edge of the system USB bus driver.

- **Interfaces**. All USB devices have one or more interfaces. An interface is a grouping of the device's endpoints which provides a particular function. (This is conceptually similar to COM interfaces, in that the interface represents a specific, well-defined set of operations.) Each interface of a device presents a different subset of the device's capabilities. The device driver may enable multiple interfaces of a device provided that they do not use the same endpoints.

The USB specification suggests the following example as an illustration of the utility of interfaces: "... an ISDN device might be configured with two interfaces, each providing 64 kbs bi-directional channels that have separate data sources or sinks on the host. Another configuration might present the ISDN device as a single interface, bonding the two channels into one 128 kbs bi-directional channel."

Class **KUsbInterface** abstracts this functionality. A driver creates an instance of this class for each interface with which it needs to interact.

- **Endpoints.** Endpoints are the individual providers and consumers of data on the physical device. Physically, this could map to an I/O register on the device. The connection between an endpoint and the driver is referred to as a pipe. The USB specification defines four types of pipes: Control, Interrupt, Bulk, and Isochronous. Each data transfer conducted by a device driver utilizes a particular pipe.

  Class **KUsbPipe** abstracts this functionality. A driver creates an instance of this class for each pipe through which it needs to transfer data. However, drivers do not create an instance for the default control pipe (endpoint 0). **KUsbLowerDevice** abstracts this pipe.

Class KUsbLowerDevice

Class **KUsbLowerDevice** is the central class for implementing a USB client driver. It enables a driver to access the default control pipe of a USB device, to configure the device, and to deliver various control and status requests to the device. This class corresponds to the logical device layer of the USB protocol stack, which is responsible for routing communication between the USB and the device endpoints.

Class **KUsbLowerDevice** is a subclass of **KPnpLowerDevice**, and accordingly inherits its member functions. The base class provides access to the properties of the underlying physical device.

An instance of class **KUsbLowerDevice** (or subclass thereof) represents endpoint 0 (the control pipe). You do not instantiate an instance of **KUsbPipe** for endpoint 0.

When the system detects a USB device, it creates a device object to represent that physical device in the system. This device object is called the Physical Device Object, or PDO. The system determines which driver is responsible for the device, and calls that driver's **AddDevice** entry point, passing a pointer to the PDO as a parameter.

Typically, **AddDevice** creates a Functional Device Object, or FDO, that corresponds to the PDO which is passed in. The role of the FDO is to enable applications and higher level drivers to send I/O requests to the USB device via the client driver. The FDO is an instance of a subclass of **KPnpDevice**.

To create the binding between the FDO and the PDO, the FDO embeds a data member of type **KUsbLowerDevice**. During construction (or in member function Initialize), the FDO is attached to the PDO, thereby inserting itself in the stack of drivers that handle IRPs for the PDO.

Class KUsbInterface

Class **KUsbInterface** abstracts a USB Interface, as described in chapters 5 and 9 of the USB specification. An interface is a collection of endpoints, which are data sources and sinks. In other words, an interface is a subset or grouping of the data producing or data consuming components of a USB Device. A given device may have several interfaces. Interfaces that include the same endpoint cannot be configured simultaneously.

A driver uses instances of **KUsbInterface** to manage access to a device's pipes. The relationship between **KUsbPipe** and **KUsbInterface** is analogous to the relationship between **KUsbInterface** and **KUsbLowerDevice**, i.e., an instance of **KUsbInterface** must be initialized before a driver can initialize a **KUsbPipe** object. The purpose of the class is therefore more structural than functional; few of the members actually interact with the physical device. Instead, the class provides a means for the driver to access information about the interface and its pipes.

Initialization of a **KUsbInterface** object occurs in the constructor, or in member function Initialize. Initialization requires an instance of **KUsbLowerDevice** from which the initializing function registers the **KUsbInterface** object with the instance of **KUsbLowerDevice**. A driver must never attempt to initialize an instance of **KUsbInterface** until the **KUsbLowerDevice** has been initialized. This means that the driver must make a successful call to the constructor or initialize member of **KUsbLowerDevice** prior to initializing any **KUsbInterface** object.

Legacy Note

Device Drivers that use USBENDPOINT structures to describe their USB device should follow different rules for initialization of **KUSbInterface** objects. These drivers must never attempt ot initialize an instance of **KUsbInterface** until the physical device has been configured. This means that the driver must make a successful call to **KUsbLowerDevice::Configure** prior to initializing any **KUsbInterface** object.

Class KUsbPipe

Class **KUsbPipe** abstracts a USB pipe, as described in chapters 5 and 9 of the USB specification. A pipe is a connection between the host and an endpoint. Each endpoint is an independently addressable consumer or provider of data. A pipe therefore represents a logically distinct context for exchange of information between the host and the device. In a given configuration of a device, each pipe is associated with a particular interface.

Initialization of a **KUsbPipe** object occurs in the constructor, or in member function Initialize. Initialization requires an instance of **KUsbLowerDevice** from which the initializing function registers the **KUsbPipe** object with the instance of **KUsbLowerDevice**. A driver must never attempt to initialize an instance of **KUsbPipe** until after it has initialized the corresponding **KUsbLowerDevice** object.

The USB specification defines four types of pipes: Control, Bulk, Interrupt, and Isochronous. The characteristics of each are described fully in the USB specification. Class **KUsbPipe** has member functions for building USB Request Blocks (URBs) for each kind of transfer. For a given pipe object, only the member function which builds an URB of the type appropriate for the pipe may be called. Once a driver has built an URB for a pipe, it calls base class member function **SubmitUrb** to deliver the request to the system USB bus driver.

IEEE 1394 Drivers

Note: 1394 support is only available in the WDM libraries.

IEEE 1394, often called *FireWire*, is a high-speed serial bus that supports speeds of 100, 200, and 400 megabits/second. Devices may be hot-plugged into the bus at any time. The 1394 bus can handle up to 64 devices, or nodes, plugged into it at any one time. The nodes communicate as peers, and, at bus reset, one device is arbitrarily elected as the bus manager node. The computer is but a node on the bus – without any special privileges.

Address Space

The IEEE 1394 bus contains a large address space, similar to memory, which is divided into portions, one for each node on the bus. The address space may be queried by specifying a device, and a 48-bit offset into the device's portion of the address space.

Asynchronous and Isochronous Transfers

The 1394 bus supports two modes of transfer: asynchronous and isochronous. Out of each packet on the bus, a small portion is reserved for asynchronous transfers, and the rest is available for isochronous transactions.

An asynchronous transfer is one in which every packet requires an acknowledgement from the target device. The packets sent generally perform read, write, or lock operations on the device's address space. Asynchronous transactions are slower than isochronous transactions, but the data is guaranteed to reach to the target device. Asynchronous transfers are also considerably simpler to set up.

An isochronous transfer is capable of sending data over the bus very quickly, in a streaming fashion. Each isochronous transfer reserves a specific number of bytes per frame, or bandwidth, which it will use to continuously transmit data. One isochronous transfer may be used either for listening or for talking.

There are 64 isochronous channels, any of which may be reserved by a particular transfer object. A talking transfer sends data over the bus to a specific channel on another device. A listening transfer waits to receive data on a particular channel. Data sent by a talking transfer may be received by zero or more listeners.

Bus Resets

Whenever a new device is plugged into the 1394 bus, or when an existing device is removed from the bus, a bus reset occurs. In a bus reset, the bus returns to its initial state, and must rediscover all nodes on the bus. At bus reset, the bus driver updates a reset generation count, necessary for most asynchronous requests, and all isochronous bandwidth and channels are freed.

In DriverWorks, **KPnpDevice::OnBusReset** is called to notify the driver of the bus reset. An isochronous device should immediately try to reallocate its resources in order to continue functioning. An asynchronous driver should update its generation count.

IEEE 1394 Bus Requests (IRBs)

A 1394 WDM driver communicates to the bus using IRBs. The IRBs are sent to the bus driver, and from there, used to carry out a request on the bus. DriverWorks class KIrb allows the driver to quickly construct an IRB. **K1394LowerDevice::SubmitIrb** allows the driver to submit the IRB to the lower device.

Support for Asynchronous and Isochronous Transfers

Note: Please see the DriverWorks On-line Help Topics, *How to Write an Asynchronous 1394 Driver* and *How to Write an Isochronous 1394 Driver* for a step-by-step explanation on how to generate a 1394 transfer.

Isochronous Support

For isochronous transfers, DriverWorks provides the classes **K1394IsochChannel** and **K1394IsochBandwidth** in order to allocate and free the isochronous bandwidth and channels, the physical resources needed to perform the isochronous transfer.

K1394IsochResource allows the driver to allocate a DDK resource object, which specifies certain options, such as which channel to use and whether the transfer should be used for talking or listening. **K1394IsochTransfer** provides the functionality to start and stop a transfer, and to add and cancel buffers from a transfer object. Each K1394IsochTransfer object must be initialized with a valid K1394IsochResource object. In fact, there is a one-to-one correspondence between instances of K1394IsochResource and K1394IsochTransfer.

The transfer object contains a list of buffers, which will be either transmitted over the bus or filled with data, depending on the direction of the transfer. **K1394IsochBufferList** provides an easy way to set up a list of buffers to add to the isochronous transfer. The transfer pauses if its buffer list becomes empty.

Asynchronous Support

Most asynchronous transactions can be made by creating read or write IRBs using **class KIrb**, and submitting them through **K1394LowerDevice::SubmitIrb**. In this case, the driver reads and writes to the device's address space.

It is also possible for the driver to allocate an address range in the computer's address space so that the device may read and write to the driver's address range. Classes **K1394AddressRange** and **K1394AddressFifo** are used to store handles to the address range. Class **K1394AsyncClient** handles allocating the address range, and contains virtual callback routines to notify the driver when the device performs an operation on the address range.

WDM Streaming Drivers

The following sections describe WDM Streaming drivers.

Classes for Stream Minidrivers

The Kernel Streaming Classes simplify the design and implementation of WDM Streaming Architecture. These classes encapsulate the C convention calls provided with WDM into a powerful C++ framework.

DriverWorks has four classes that aid in the development of stream minidrivers:

- **Class AVStream** is Microsoft's next generation multimedia class driver. This is the only multimedia class driver that Microsoft will evolve. Existing minidrivers under stream class and port class continue to work in Microsoft® Windows® XP. However, Microsoft now supports the older stream class driver only on an as-is basis. Vendors may still write audio drivers using port class. Vendors developing new drivers that would have run under stream class may wish to consider instead writing to AVStream. AVStream is especially recommended for vendors who are writing software filters or drivers for new technology areas or new device types.

The AVStream subsystem features a sizeable set of supported object types such as **KSDEVICE**, **KSFILTER**, **KSPIN**, etc. These objects are created by the Windows Kernel Subsystem (KS) on behalf of AVStream minidrivers and characterized by their static data properties and event processing requirements. The main job of an AVStream minidriver is basically to register the set of properties describing minidriver's KS capabilities and provide event processing for KS events.

DriverWorks provides a compact set of base classes (one for each distinct KS object type), where both properties and behavior of each KS object is clearly defined in one place (class). The framework's job is to construct the KS descriptors based on the compile-time properties of the classes representing the KS object types. Each base class (KsXxx) defines all properties and event handlers pertained to given object type. For instance, KsPin class defines the set of all pin-related default data properties and event handlers for KSPIN objects. Driver writer uses KsPin as a base class and (re)defines only those properties and event handlers specific to the given minidriver. DriverWorks ships with 2 sample drivers, VCap and HwCap, functionally similar to the DDK samples but implemented using the DriverWorks AVStream C++ framework.

DriverWorks provides 3 base classes — **KsDevice**, **KsFilter** and **KsPin** — for each of the core KS object types **KSDEVICE**, **KSFILTER,** and **KSPIN** accordingly. Each of the base classes defines a default set of appropriate data properties (such as formats, data ranges, etc.) and event handlers. The user publicly derives from the base classes to define actual data properties and event handlers of interest. The framework uses the C++ template techniques to figure the actual layout of user class and construct appropriate device descriptors and dispatch tables to be registered with the KS subsystem.

Refer to the DriverWorks on-line help for more detailed information on the AVStream class.

- **Class KStreamMinidriver** is the WDM filter device driver interface to the operating system. Derived from KDriver, it operates in conjunction with the WDM streaming module to provide user mode components with a DirectShow compliant filter graph object. Windows Plug and Play constructs a WDM filter device object when streaming hardware device is installed or when a kernel-mode driver (streaming filter) is loaded. **KStreamMinidriver** provides a default implementation for basic *WDM* filter device object commands, and dispatches commands to the appropriate **KStreamAdapter** object. You define a class derived from **KStreamMinidriver** and override methods that your hardware device supports. A single **KStreamMinidriver** derived object can support multiple **KStreamAdapter** objects.

- **Class KStreamAdapter** controls an individual "hardware board". **KStreamAdapter** is a base class called directly by the **KStreamMinidriver** object. Although most implementations directly control physical hardware, it is possible to create **KStreamAdapter** derived classes that are pure software components or that layer on top of other hardware drivers (see our sample). Most commands sent to class **KStreamMinidriver** are sent directly to the appropriate **KStreamAdapter** object. The base class implementation of **KStreamAdapter** includes default implementations for *all* commands, usually returning STATUS_NOT_IMPLEMENTED. Your **KStreamAdapter** object usually includes control of device specific properties and should *contain* the **KStream** derived classes you support. Much of the **KStreamAdapter** work is in initializing and opening stream objects, setting and retrieving device properties, and handling hardware interrupts.

- **Class KStream** provides control and data handling for an individual media stream. Where the **KStreamMinidriver** and **KStreamAdapter** objects provide control over the device driver and hardware, **KStream** is where stream control, stream properties, and stream data buffers are managed. Once a **KStream** derived class is opened, stream specific data and control flow directly to that stream through dispatch callbacks that were registered with the WDM stream class driver when the **KStreamAdapter** was created. The current stream state (STOPPED, PAUSED, RUNNING) is managed entirely in the **KStream** class. The **KStream** class provides default implementations for *all* stream commands, returning STATUS_NOT_IMPLEMENTED for most commands. Developers override the appropriate virtual methods in a **KStream** derived class to implement appropriate functionality.

The stream minidriver classes provide a structured interface to the WDM Streaming architecture as illustrated in the diagram below.

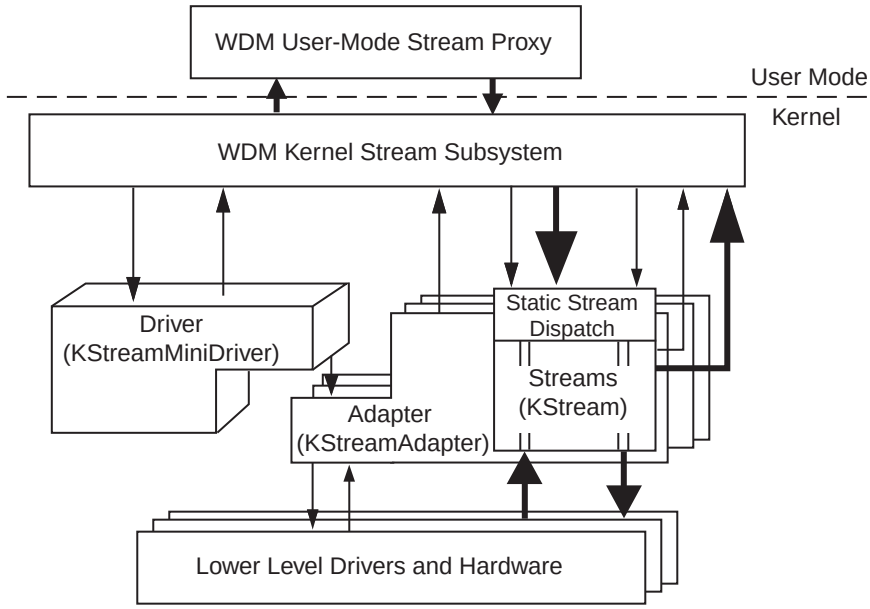


Figure 4-1. WDM Streaming Architecture

General

In this model, applications enumerate WDM streaming driver filters just as they do any other DirectShow filter. The WDM User-Mode Stream Proxy presents a standard DirectShow filter for every WDM Streaming Filter Driver in the system. The proxy uses Device I/O Control commands to the WDM Kernel Stream Subsystem to interact with WDM Streaming Filter Driver as a means of enumerating and setting properties, controlling the filter, and streaming data to/from the filter object. The WDM Streaming architecture provides the filter framework, greatly simplifying driver implementation.

A small set of well-defined command and data calls and callbacks comprise the interface between the WDM Kernel Stream Subsystem and a WDM Streaming Filter Driver. DriverWorks stream classes implement these interfaces directly, allowing developers to focus effort directly on the handling of physical devices, command processing, and data transfer.

Commands to drivers, adapters, and streams are contained in a structure called a Service Request Block (SRB). An SRB contains all the information a filter driver needs to complete a request. Filter drivers can provide implementations that allow several SRBs to be in process at one time or can choose a more simplified approach that processes a single SRB to completion. SRBs targeted at specific adapter objects are managed independently of SRBs targeted at individual streams on that adapter. The DriverWorks stream classes implement a dispatching mechanism and methods for all the defined SRB command codes in driver, adapter, and stream objects, allowing developers to implement commands by simply overriding these methods.

Finally, the DriverWorks stream classes simplify callbacks to the WDM Kernel Stream Subsystem by providing abstracted callback methods as part of the **KStreamAdapter** and **KStream** classes.

Flow of Control

The WDM Kernel Stream Subsystem instantiates a filter driver when Windows Plug and Play identifies a hardware device or filter device for that driver. At that point, the Kernel Stream Subsystem loads the driver file and calls the **DriverEntry** method in the **KStreamMinidriver** subclass.

Creating the Filter Driver

DriverEntry must register the device's parameters with the Kernel Stream Subsystem by calling the **KStreamMinidriver::Register** method. This registration process provides the WDM Kernel Stream Subsystem with the address of entry points for commands, hardware interrupts, timeouts, and the size of objects, such as the filter driver's **KStreamAdapter** derived class. With the registration process complete, **DriverEntry** returns with the results of the registration process.

Once registered, the WDM Kernel Stream Subsystem calls the **KStreamMinidriver::ReceivePacket** handler with device/adapter SRBs to process. Developers supply an **OnCreateAdapter** and optionally and **OnDestroyAdapter** method which are called when an SRB_INITIALIZE_DEVICE and SRB_UNINITIALIZE_DEVICE commands are received. **OnPagingOutDriver** can be overridden to perform any work needed when the system pages out the filter driver. All other SRBs specify an adapter object and are passed directly to the **KStreamAdapter::ReceivePacket** method for processing.

Creating the Adapter

When **OnCreateAdapter** is called, it must construct the **KStreamAdapter** derived object in the memory supplied by the *WDM* Kernel Stream Subsystem and then call the Initialize method on that object (see sample). The Initialize method performs any special initialization for the **KStreamAdapter**. This is especially useful for initializing the properties associated with all stream objects contained in that adapter object (see sample).

After **OnCreateAdapter** returns, the WDM Kernel Stream Subsystem will send an SRB containing the SRB_INITIALIZATION_COMPLETE command. The Adapter processes this command in its **OnInitializationComplete** method by registering all the controls it contains and then calling the Register method to register the filter driver instance.

Once registered, the adapter is ready to process all SRB requests.

When an adapter is open, it processes an SRB request for the properties of controls, information regarding streams, and opening and closing streams. For stream oriented requests (info, properties, open and close), the **KStreamAdapter** derived object determines exactly which stream the request is targeted at and then calls a method on that stream to complete the request.

Opening a Stream

KStreamAdapter::OnOpenStream (SRB_OPEN_STREAM) opens the stream object. It calls **KStream::Initialize** to obtain information regarding the stream and exposed methods that the WDM Kernel Stream Subsystem will use to pass control and data requests to the stream, then calls the **OnOpen** method in the **KStream** derived class, which validates the data format requested, and then performs any special open processing.

Stream Control and Data

As soon as a stream is open, the system calls the **KStream** members ReceiveData, ReceiveControl, DispatchClock, DispatchTimer, and DispatchEvent to communicate with the stream object. These methods act as a dispatch layer, identifying the stream (by the getting the stream pointer from the SRB) and then calling the appropriate stream for processing. Developers override methods in the **KStream** derived object to implement read, write, buffer handling, clock handling, and event handling functions.

Summary of Initialization Logic

The **OnCreateAdapter** member of **KStreamMinidriver** must construct the adapter object.

After construction, **OnCreateAdapter** calls **KStreamAdapter** member **Initialize**. This is a pure virtual function, and, as such, all subclasses of **KStreamAdapter** must implement it.

KStreamAdapter::Initialize must initialize its streams by calling **KStream::InitializeStream** for each stream. This is a pure virtual member function of **KStream**.

Each subclass of **KStream** must implement **InitializeStream** and this function must call **InitializeBaseClass.** If necessary, this function can be overridden.

Closing Streams Adapter Shutdown

Streams are started and stopped through the adapter object. When running, requests are handled directly by the stream through **ReceiveData** and **ReceiveControl**.

Once stopped, a stream is closed by an SRB_CLOSE_STREAM processed in the adapter. **KStreamAdapter::OnCloseStream** identifies the appropriate stream and then calls that stream's **OnClose** method to complete processing.

When all streams are closed, the adapter object itself might be destroyed (SRB_UNINITIALIZE_DEVICE). The object derived from **KStreamMinidriver** may override the default implementation of **OnDestroyAdapter** to provide special shutdown processing.

More Information

Detailed information on SRB processing and WDM filter drivers can be found in the WDM DDK section *"Notes on Writing StreamClass Minidrivers"*.

Also, see the KStream Overridables help topic for a list of overridable members of the stream classes.

Steps for Building a Stream Minidriver

Follow these steps to build a stream minidriver using the DriverWorks stream classes.

1. Define a class derived from **KStreamMinidriver**. This class *must* provide a DriverEntry method and an **OnCreateAdapter** method. Refer to the on-line documentation for those methods to identify what features they must implement.

2. Define a class derived from **KStreamAdapter**. This class must include all streams and controls as member variables (primarily for memory management). This class *must* implement an Initialize method and a GetStream method. Refer to the on-line documentation for those methods to identify the features they must implement.

3. Define one or more classes derived from **KStream**. This class must include all attributes of the stream and must be prepared to handle all control and data requests.

4. Identify the structures associated with the properties and topology of your controls and streams and implement them as members of the control/stream objects. In some ways, this is the most difficult part. Although you can construct the properties required in response to command requests, it is often easiest to use the predefined macros provided with the WDM DDK to construct the appropriate properties, pin definitions, and topologies. For example, the Video Capture sample defines these as static members of its control and stream objects.

5. Override methods in your Driver, Adapter, and Stream objects to implement SRB specific processing required by your device. You will find that stepping through the Video Capture sample can help you in understanding the flow of control, format negotiation, and data processing in your driver.

6. Modify the `.INF` file supplied with the Video Capture sample and begin debugging.

Chapter 5
DriverNetworks Object Model

- Introduction
- NDIS Miniport Driver Framework
- NDIS Connection-Oriented Miniport Driver Framework
- NDIS Miniport Call Manager Framework
- NDIS Intermediate Driver Framework
- NDIS Protocol Driver Framework
- NDIS Transport Driver Framework
- TDI Client (DriverSockets) Framework
- DriverSockets vs. WinSock

Introduction

Frameworks define how sets of classes are arranged to form a working component. DriverNetworks defines the following class frameworks:

- NDIS Miniport Drivers
- NDIS Intermediate Drivers
- NDIS Protocol and Transport Drivers
- TDI Clients

The DriverNetworks framework resides between the system NDIS and TDI interfaces. It delegates system callbacks into client's classes derived from *adapter* base classes supplied by DriverNetworks.

For NDIS drivers, the delegation is provided by *wrapper* template classes, such as KNdisWrapper. *Wrapper* classes provide a set of static methods-callbacks, which are called directly from NDIS. Each callback is accompanied by a *context* parameter, which is usually a 'this' pointer to an adapter class instance. The wrapper then calls into the appropriate member function of the adapter class, which is referred to as a *handler*.

Handler methods must have certain prototypes called *signatures*. The handler signatures, the naming conventions, and the callback context closely resemble the conventions for appropriate MiniportXxxx and ProtocolXxxx handlers defined in the NDIS specification.

Wrapper classes are C++ templates parameterized by the *adapter* class. This mechanism provides the driver writer with greater flexibility to implement the adapter's handler methods. The handlers might be declared as either virtual or non-virtual, or inline members of the adapter class, as long as their signatures are compliant with the framework's requirements. For example, declaring the handler's inline yields the same run time efficiency as would be returned by a straight C implementation of the handler (e.g., in *free* builds).

KNdisMiniAdapter is the base class for all adapter classes. NDIS Miniport drivers derive their *adapter* class from KNdisMiniAdapter and provide the handler methods called on various NDIS events. In addition, KNdisMiniAdapter exposes methods to access various NDIS services that are routinely requested by NDIS Miniport drivers.

KNdisFilterAdapter is the base class for *filter* or *intermediate* adapters. It is derived from the KNdisMiniAdapter class and provides a number of virtual handler functions that the NDIS Intermediate driver writer implements to monitor network packets and status indications traveling between the *real* protocol and the NIC(s).

KTdiClient is the base class for kernel mode *socket* classes. TDI clients utilize the Transport Driver Interface to access protocols, such as TCP/IP.

NDIS miniport *adapter* classes derive from the **KNdisMiniAdapter** class and declare the required signatures for handler methods, such as **Initialize()**, **Halt()**, and **Reset()**. They might either be virtual or non-virtual, or inline members of the derived class. The naming conventions for the handlers closely match those used in the DDK. For example, **Initialize()** corresponds to **MiniportInitialize()**, and **Halt()** corresponds to **MiniportHalt()**. The processing context and IRQL levels for the handlers are the same as for the corresponding **MiniportXxxx()** handlers in the DDK.

The UML class diagram in depicts the basic relationships between the **DriverNetworks** framework classes and classes implemented by the driver writer.

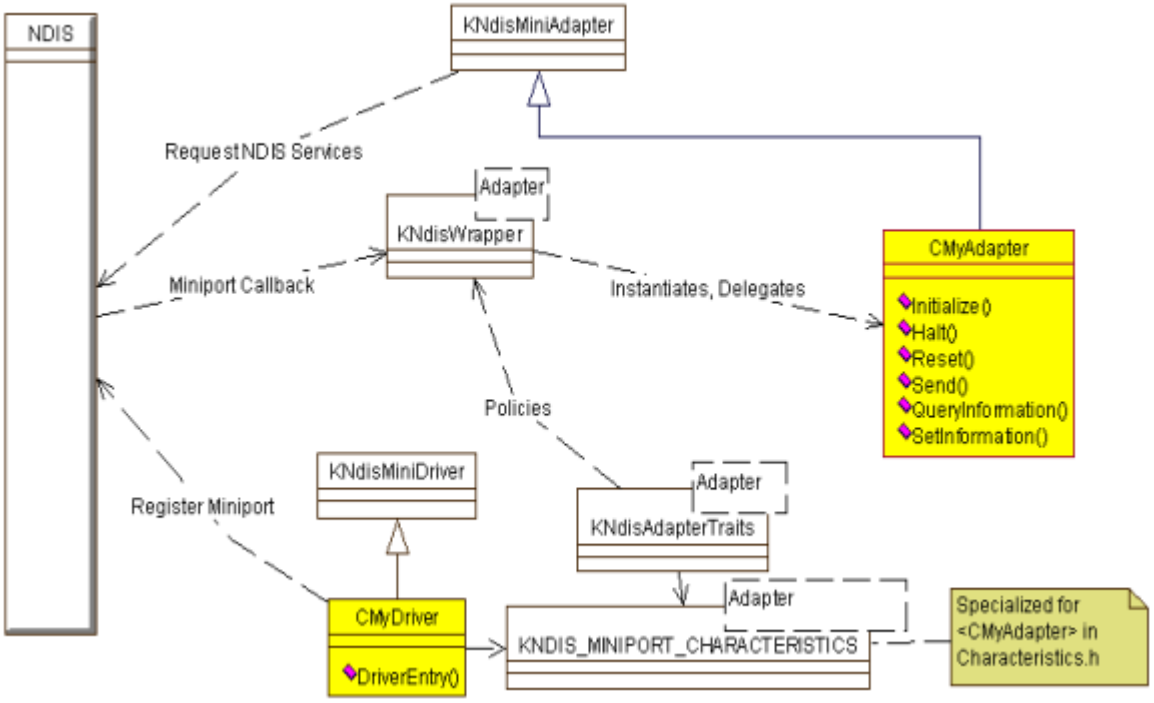


Figure 5-1. Basic Relationships between Framework Classes and Driver Writer Classes

The driver writer provides two main classes:

- KNdisMiniDriver-derived driver class
- KNdisMiniAdapter-derived *adapter* class

Note: The Network Driver Wizard application automatically generates both classes for NDIS driver project.

Upon a driver load, the framework calls the adapter's **DriverEntry()** method, which registers the adapter with NDIS. The registration is handled by the helper template class **KNDIS_MINIPORT_CHARACTERISTICS**, which passes NDIS a set of handlers that the adapter supports. The default set of handlers along with other miniport characteristics are specified in the helper template class, KNdisAdapterTraits. Each miniport driver project includes a header file, `Characteristics.h`, which optionally redefines some of those characteristics (e.g., the set of handlers) for the specific adapter.

After the adapter registration has completed, NDIS might start calling the adapter's handlers. The handlers are called from NDIS via the framework template class, KNdisWrapper. For each adapter class A, the framework instantiates an implementation of KNdisWrapper<A> that *forwards* NDIS callbacks to the appropriate handlers defined at A. It is not necessary for the driver writer to implement the **QueryInformation()** and **SetInformation()** handlers, which both process NDIS OIDs. In DriverNetworks, the framework implements OID processing via *OID maps*.

Any instances of a KNdisMiniAdapter-derived class are automatically created by the DriverNetworks framework. As a result, there is never a need to call a `new()` for the adapter. The framework is responsible for registering with NDIS, intercepting the original NDIS **MiniportInitialize()** callback, and creating an instance of the adapter class. After the instance is created, the framework passes control to the **Initialize()** handler of the derived class. The Initialize handler performs the necessary custom steps, such as claiming hardware resources and initializing the NIC. A successful return from the **Initialize()** handler enables the adapter for the system.

The destruction of KNdisMiniAdapter-derived objects is also handled by the framework. The **Halt()** handler is called immediately before the destruction. The adapter class performs a hardware-specific shutdown, resource deallocation, etc. The **Halt()** handler is typically called when the miniport is disabled via the Control Panel or on system shutdown.

NDIS Miniport Driver Handler Signatures

```
class CMyAdapter : public KNdisMiniAdapter {
 public:
 // Initialization and shutdown
   NDIS_STATUS Initialize(IN OUT KNdisMedium& Medium, IN KNdis-
Config& Config) ;
    VOID Halt(VOID);
    NDIS_STATUS Reset(OUT PBOOLEAN AddressingReset);
    void Shutdown(VOID);
// OID requests
    NDIS_STATUS QueryInformation(
          IN NDIS_OID Oid,
          IN PVOID InformationBuffer,
          IN ULONG InformationBufferLength,
          OUT PULONG BytesWritten,
          OUT PULONG BytesNeeded
          ) ;
  NDIS_STATUS SetInformation(
          IN NDIS_OID Oid,
          IN PVOID InformationBuffer,
          IN ULONG InformationBufferLength,
          OUT PULONG BytesRead,
          OUT PULONG BytesNeeded
          ) ;
// Sending packets
    VOID SendPackets(
        IN PPNDIS_PACKET PacketArray,
        IN UINT NumberOfPackets
        ) ;
  VOID ReturnPacket(IN PNDIS_PACKET Packet);

 // overridables for interrupt-driven miniports
   BOOLEAN CheckForHang() ;
   VOID DisableInterrupt() ;
   VOID EnableInterrupt() ;
   VOID HandleInterrupt() ;
   VOID Isr(OUT PBOOLEAN InterruptRecognized,
         OUT PBOOLEAN QueueMiniportHandleInterrupt ) ;
 };
```

NDIS Connection-Oriented Miniport Driver Framework

An NDIS connection-oriented miniport driver is an NDIS miniport driver that handles the additional responsibilities of creating and deleting virtual connections, activating and deactivating virtual connections, sending packets on virtual connections and accepting protocol-initiated requests to get or set information. A connection-oriented miniport driver provides these services as an interface between connection-oriented protocols and the NIC hardware.

The DriverNetworks framework actively supports the abstractions of connection-oriented miniport adapter classes and virtual connection classes.

NDIS connection-oriented miniport adapter classes derive from the **KNdisMiniAdapter** class and declare the required signatures for handler methods such as **Initialize()**, **Halt()**, and **Reset()**. They might be virtual or non-virtual, or inline members of the derived class. The naming convention for the handlers closely matches those used in the DDK. For example **Initialize()** corresponds to **MiniportInitialize()**, and **Halt()** corresponds to **MiniportHalt()**. The processing context and IRQL levels for the handlers are the same as for the corresponding **MiniportXxxx()** handlers in the DDK.

Figure 5-2 depicts the relationships between the DriverNetworks framework classes and classes implemented by the driver writer.

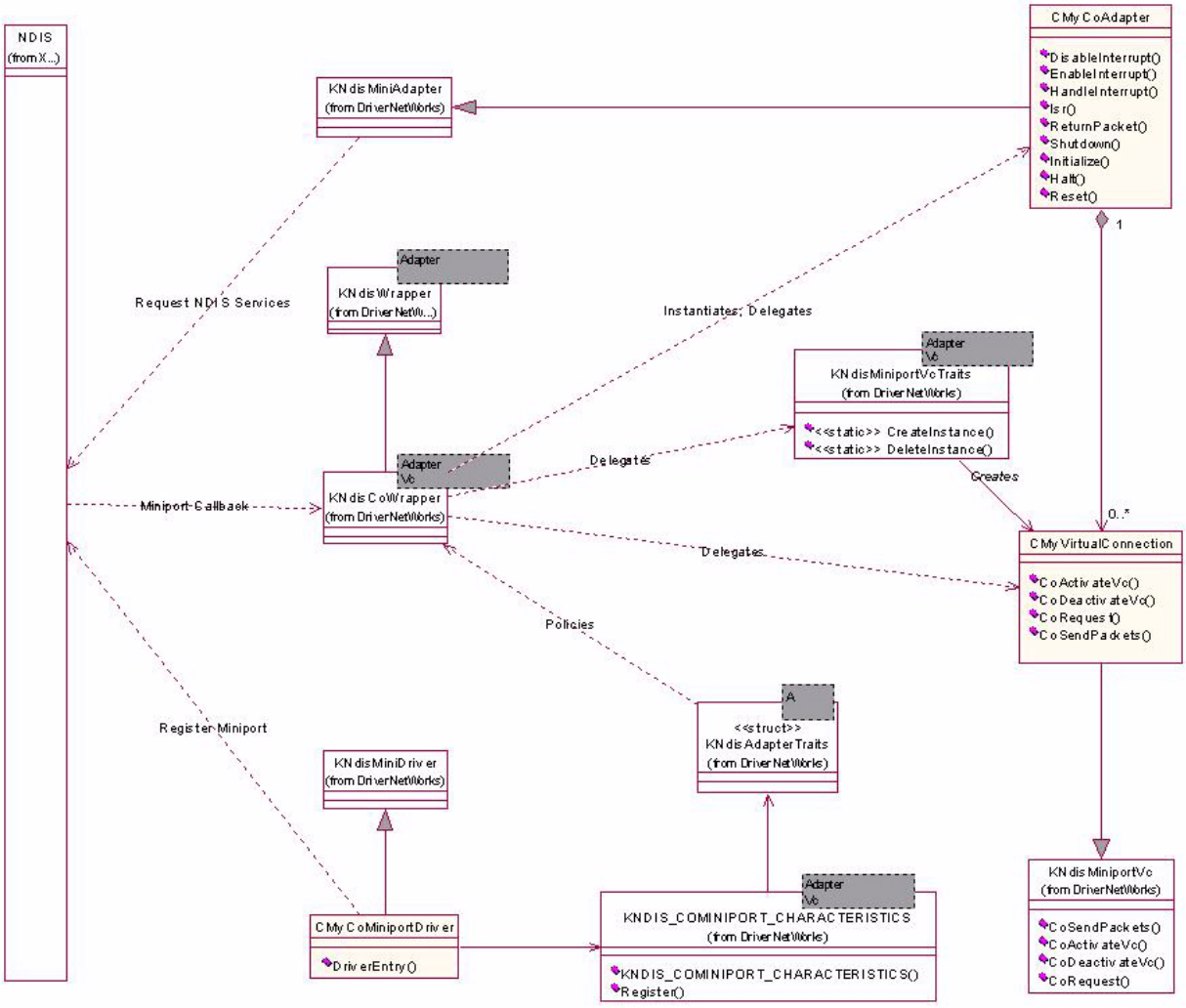


Figure 5-2. UML Class Diagram: Connection-Oriented Miniport Driver

The driver writer provides three main classes:

- KNdisMiniDriver-derived driver class
- KNdisMiniAdapter-derived adapter class
- KNdisMiniportVc-derived Virtual Connection class

Note: The Network Driver Wizard application automatically generates all three classes for an NDIS Connection-Oriented Miniport Driver project.

Upon a driver load, the framework calls the adapter's **DriverEntry()** method, which registers the adapter with NDIS. The registration is handled by the template class **KNDIS_COMINIPORT_CHARACTERISTICS**, which passes NDIS a set of handlers that the adapter supports. The default set of handler along with other miniport characteristics are specified in the helper template class KNdisAdapterTraits. Each connection-oriented miniport driver project includes a header file, `Characteristics.h`, which optionally redefines some of those characteristics (e.g., the set of handlers) for the specific adapter.

After the adapter registration has completed, NDIS might start calling the adapter's handlers. The handlers are called from NDIS via the framework template class KNdisCoWrapper. For each adapter class A and virtual connection class C, the framework instantiates an implementation of KNdisCoWrapper<A,C> that forwards NDIS callbacks to the appropriate handlers defined at A or C. It is not necessary for the driver writer to implement the CoRequest handler, which processes NDIS OIDs. In DriverNetworks, the framework implements OID processing via OID maps.

Any instances of a KNdisMiniAdapter-derived class are automatically created by the DriverNetworks framework. As a result there is never a need to explicitly allocate an adapter object. The framework is responsible for registering with NDIS, intercepting the original NDIS **MiniportInitialize()** callback, and creating an instance of the adapter class. After the instance is created, the framework passes control to the **Initialize()** handler of the derived class. The handler performs the necessary custom steps, such as claiming hardware resources and initializing the NIC. A successful return from the **Initialize()** handler enables the adapter for the system.

The destruction of KNdisMiniAdapter-derived objects is also handled by the framework. The **Halt()** handler is called immediately before the destruction. The adapter class performs hardware-specific shutdown, resource de-allocation, etc. The **Halt()** handler is typically called when the miniport is disabled via the Control Panel or on system shutdown.

Connection-oriented miniport virtual connection classes derive from the **KNdisMiniportVc** class and declare the required signatures for handler methods such as **CoActivateVc()**, **CoDeactivateVc()**, and **CoSendPackets()**. The creation and destruction of virtual connection objects is initiated by NDIS. For a given adapter class A and virtual connection class C creatable by adapter objects of class A, the framework instantiates an implementation of KNdisMiniportVcTraits<A,C> and relies on the static **CreateInstance()** and **DeleteInstance()** methods of such instantiations to control the creation and deletion of virtual connections. The framework intercepts the original NDIS **MiniportCoCreateVc()** callback and calls **KNdisMiniportVcTraits<A,C>::CreateInstance()** to create a virtual connection. After the virtual connection is created, the framework forwards all subsequent activity on the virtual connection to the handlers of the KNdisMiniportVc-derived class. Similarly, the framework intercepts the original NDIS **MiniportCoDeleteVc()** callback and calls **KNdisMiniportVcTraits<A,C>::DeleteInstance()** to delete a virtual connection.

The driver writer can control the creation of virtual connections by overriding the methods of the KNdisMiniportVcTraits class. The default implementation of the static method **KNdisMiniportVcTraits<A,C>::CreateInstance()** allocates a new virtual connection object from the NDIS pool and constructs it. The default implementation of the static method **KNdisMiniportVcTraits<A,C>::DeleteInstance()** destroys such a previously allocated object and returns the underlying storage to the NDIS pool.

NDIS Miniport Call Manager Framework

An NDIS miniport call manager is a connection-oriented NDIS miniport driver that also provides call management services to connection oriented protocols. For such drivers, the interface between the call manager and the connection-oriented miniport driver is internal to the driver and opaque to NDIS.

The DriverNetworks framework actively supports the abstractions of miniport call manager adapter classes and virtual connection classes.

NDIS miniport call manager adapter classes derive from the **KNdisMiniAdapter** class and declare the required signatures for handler methods such as **Initialize()**, **Halt()**, and **Reset()**. They might be virtual or non-virtual, or inline members of the derived class. The naming convention for the handlers closely matches those used in the DDK. For example **Initialize()** corresponds to **MiniportInitialize()**, and **Halt()** to **MiniportHalt()**. The processing context and IRQL level for the handlers are the same as for the corresponding **MiniportXxxx()** handlers in the DDK.

Figure 5-3 depicts the relationships between the DriverNetworks framework classes and classes implemented by the driver writer

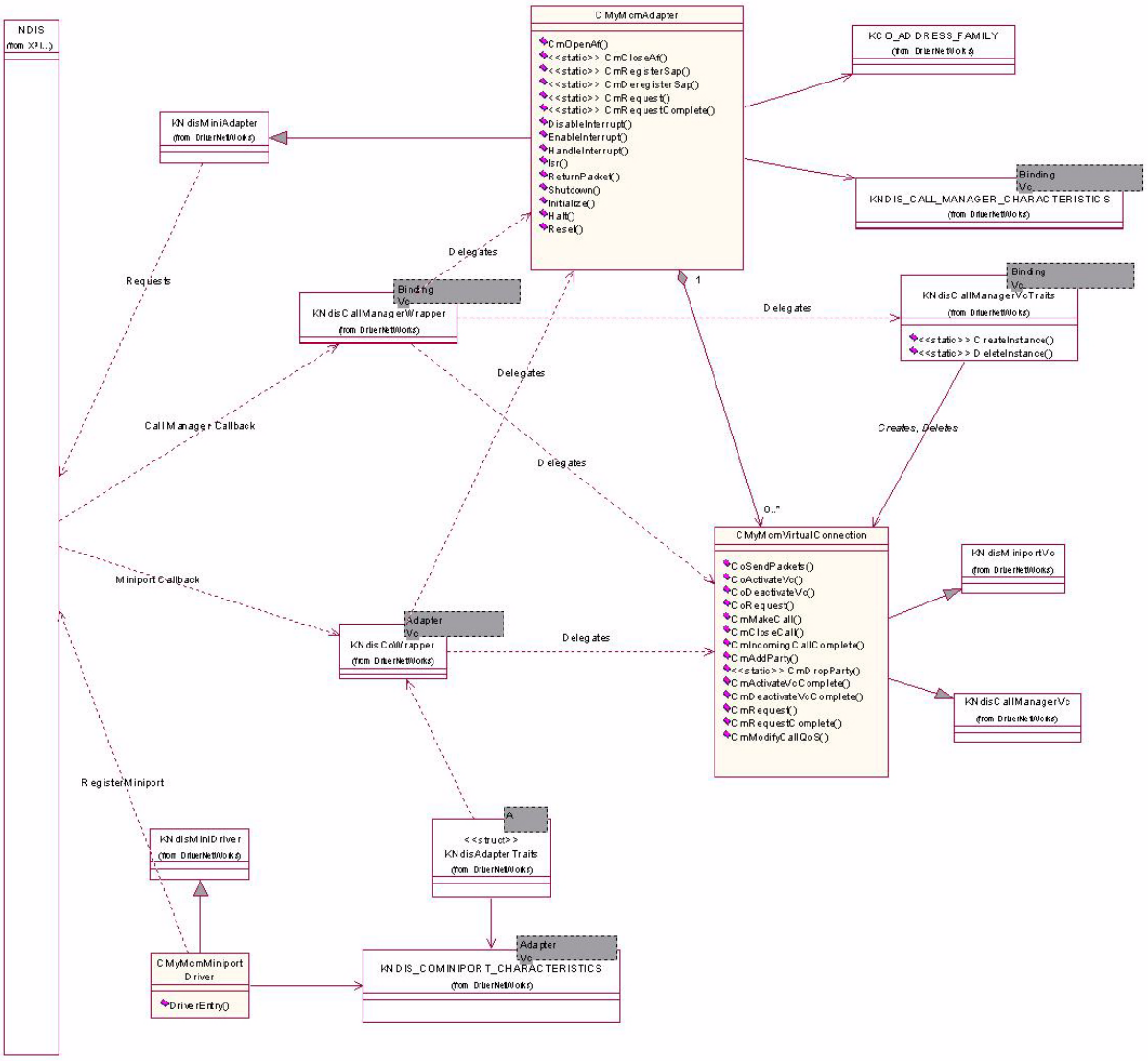


Figure 5-3. UML Class Diagram: Miniport Call Manager

The driver writer provides three main classes:

- KNdisMiniDriver-derived class
- KNdisMiniAdapter-derived class
- KNdisMiniportVc and KNdisCallManagerVc-derived class

Note: The Network Driver Wizard application automatically generates all three classes for an NDIS Miniport Call Manager project.

Upon a driver load, the framework calls the adapter's **DriverEntry()** method, which registers the adapter with NDIS. The registration is handled by the template class **KNDIS_COMINIPORT_CHARACTERISTICS**, which passes NDIS a set of handlers that the adapter supports. The default set of handler along with other miniport characteristics are specified in the helper template class KNdisAdapterTraits. Each connection-oriented miniport driver project includes a header file, `Characteristics.h`, which optionally redefines some of those characteristics (e.g., the set of handlers) for the specific adapter.

After the adapter registration has completed, NDIS might start calling the adapter's handlers. The handlers are called from NDIS via the framework template class KNdisCoWrapper. For each adapter class A and virtual connection class C, the framework instantiates an implementation of KNdisCoWrapper<A,C> that forwards NDIS callbacks to the appropriate handlers defined at A or C. It is not necessary for the driver writer to implement the CoRequest handler, which processes NDIS OIDs. In DriverNetworks, the framework implements OID processing via OID maps.

Any instances of a KNdisMiniAdapter-derived class are automatically created by the DriverNetworks framework. As a result there is never a need to explicitly allocate an adapter object. The framework is responsible for registering with NDIS, intercepting the original NDIS **MiniportInitialize()** callback, and creating an instance of the adapter class. After the instance is created, the framework passes control to the **Initialize()** handler of the derived class. The handler performs the necessary custom steps, such as claiming hardware resources and initializing the NIC. A successful return from the **Initialize()** handler enables the adapter for the system.

The framework also makes use of the adapter's **Initialize()** handler to register the call management capabilities of the adapter. The framework initializes and registers **NDIS_CALL_MANAGER_CHARCTERISTIC** and **CO_ADDRESS_FAMILY** structures with a call to **NdisMCmRegisterAddressFamily()** during the handling of the adapter's **Initialize()** method. The registration passes NDIS the required set of handlers that the call manager supports.

A successful return from the **Initialize()** handler enables the adapter for the system. NDIS might then start calling the call manager's handlers. The handlers are called from NDIS via the framework template KNdisCallManagerWrapper. For each adapter class A and virtual connection class C, the framework instantiates an implementation of KNdisCallManagerWrapper<A,C> that forwards NDIS callbacks to the appropriate handlers defined at A or C.

The destruction of KNdisMiniAdapter-derived objects is also handled by the framework. The **Halt()** handler is called immediately before the destruction. The adapter class performs hardware-specific shutdown, resource de-allocation, etc. The **Halt()** handler is typically called when the miniport is disabled via the Control Panel or on system shutdown.

Miniport Call Manager virtual connection classes derive from both KNdisMiniportVc and KNdisCallManagerVc classes and declare the required signatures for handler methods such as **CoActivateVc()**, **CoDeactivateVc()**, **CoSendPackets()**, **CmMakeCall()**, and **CmCloseCall()**. The creation and destruction of virtual connection objects is initiated by NDIS. For a given adapter class A and virtual connection class C creatable by adapter objects of class A, the framework instantiates an implementation of KNdisCallManagerVcTraits<A,C> and relies on the static **CreateInstance()** and **DeleteInstance()** methods of such instantiations to control the creation and deletion of virtual connections. The framework intercepts the original NDIS **ProtocolCoCreateVc()** callback and calls **KNdisCallManagerVcTraits<A,C>::CreateInstance()** to create a virtual connection. After the virtual connection is created, the framework forwards all subsequent activity on the virtual connection to the handlers of the KNdisMiniportVc and KNdisCallManagerVc derived class. Similarly, the framework intercepts the original NDIS **ProtocolCoDeleteVc()** callback and calls **KNdisCallManagerVcTraits<A,C>::DeleteInstance()** to delete a virtual connection.

The driver writer can control the creation of virtual connections by overriding the methods of the KNdisCallManagerVcTraits class. The default implementation of the static method **KNdisCallManagerVcTraits<A,C>::CreateInstance()** allocates a new virtual connection object from the NDIS pool and constructs it. The default implementation of the static method **KNdisCallManagerVcTraits<A,C>::DeleteInstance()** destroys such a previously allocated object and returns the underlying storage to the NDIS pool.

NDIS Intermediate Driver Framework

Tip: *The easiest way to become familiar with the framework is to start the Network Driver Wizard, generate a skeleton NDIS miniport or NDIS IM driver project, and explore the generated source code using the Visual Studio class browser.*

NDIS Intermediate (IM) drivers are layered between NDIS protocol drivers, such as TCP/IP, and NDIS miniport drivers, talking to real NICs. IM drivers intercept packets and status indications traveling between NDIS protocol and miniport layers. Intermediate drivers expose both a virtual adapter(s) that binds to protocol drivers in the system and a protocol edge which binds to miniport drivers in the system. There are two basic types of Intermediate drivers in Windows 2000 and Windows XP; NDIS Filter drivers and NDIS Mux drivers.

NDIS Filter drivers expose one virtual adapter for each physical adapter to which it binds on the lower edge. The system automatically inserts the intermediate filter between all protocols and the physical adapters. The upper edge binding interface and the lower edge binding interface are the same.

Figure 5-4 is a sample configuration of an installed filter in the system. All protocols that were bound to physical adapters are now bound to the virtual adapters exposed by the filter driver.

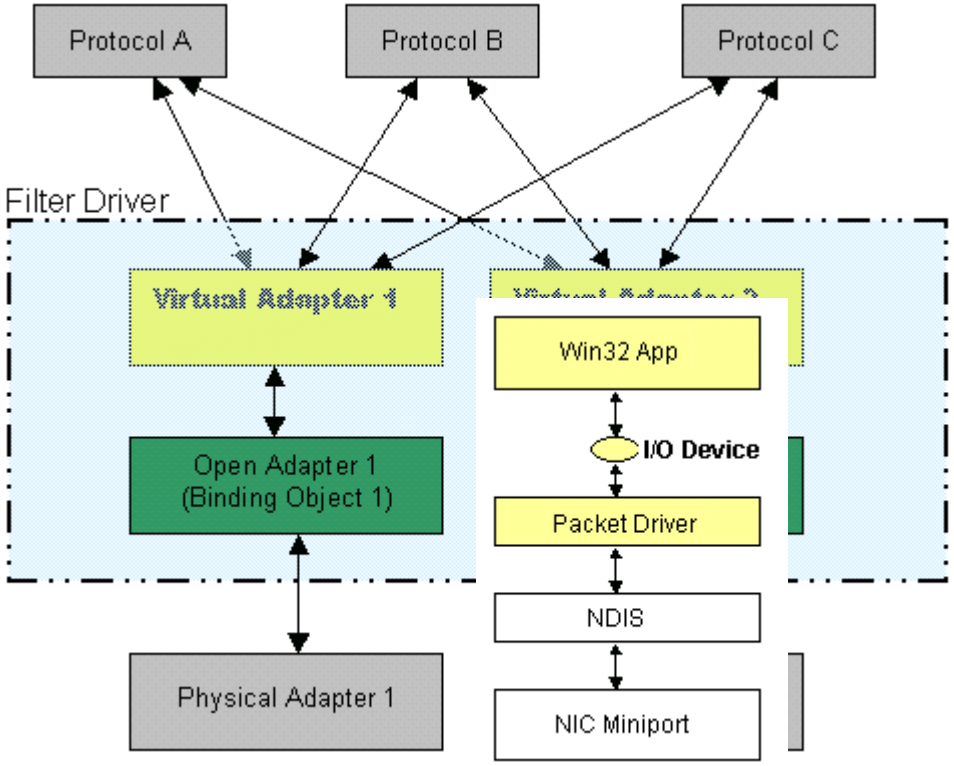


Figure 5-4. Sample of Installed System Filter

NDIS Filter Driver

A programmer develops an NDIS Filter driver to implement some sort of packet filtering. The driver could examine the data buffers of the packet and decide to copy, modify, or drop the packet. Examples of NDIS Filter drivers are packet monitoring, packet scheduling, load balancing, and firewall drivers.

NDIS Mux Driver

NDIS Mux drivers allow more flexibility to the driver writer if required. The upper and lower binding interfaces do not have to be the same. So the driver can convert one type of interface to another. The driver can expose more than one virtual adapter per physical adapter in the system (N virtual adapters to one physical adapter.) This flexibility comes at the cost of the driver managing the complicated internal binding arrangements. Also, a user mode component called a Notify Object must be used. (See the related sections “Understanding the Notify Object Framework” and “NDIS Intermediate (MUX) Drivers.”)

An example would be a driver that performs LAN emulation for some other medium. Or the driver could multiplex all physical adapters into one virtual adapter (1 to N.) An example would be a driver that performs load balancing of three heterogeneous physical adapters but exposes only one adapter to the system.

Figure 5-5 is a sample configuration for an NDIS Mux Driver (1 to N).

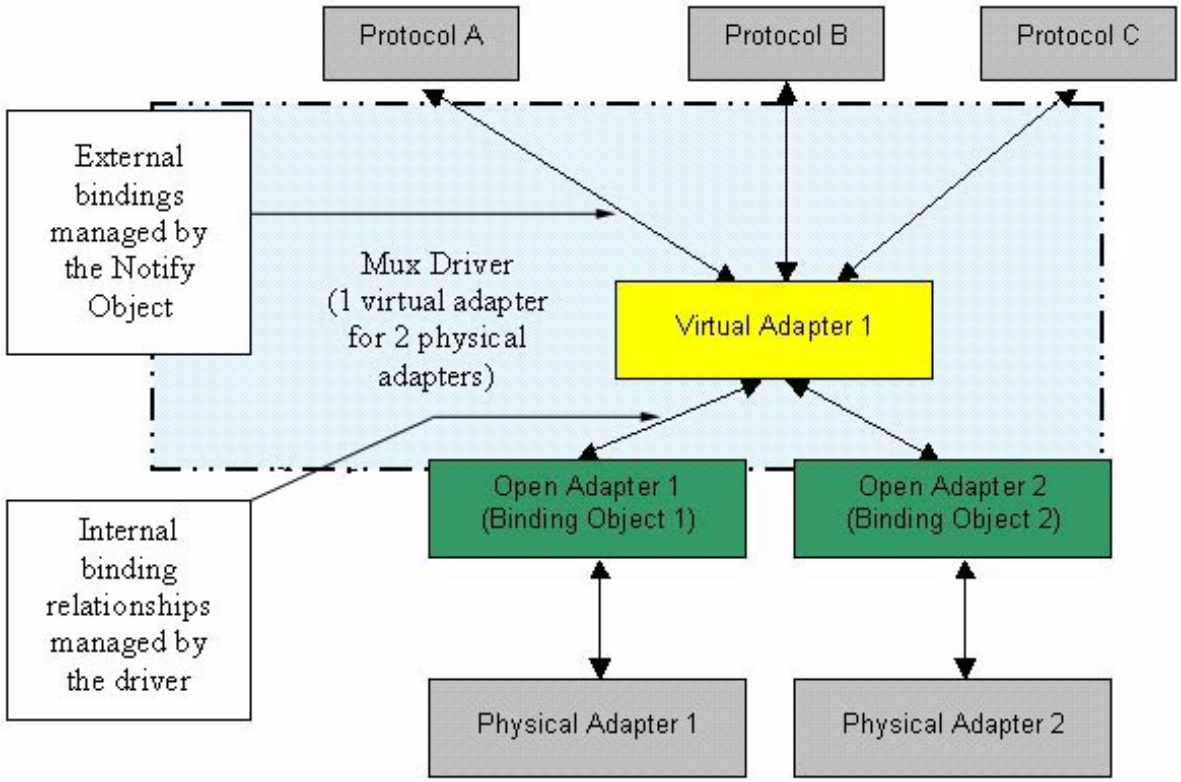


Figure 5-5. NDIS Mux Driver (1 to N) Sample Configuration

The NDIS Mux driver exposes a virtual adapter for the two physical adapters on the system. The protocols bind to the one virtual adapter. The Mux driver chooses which physical adapter to route packets.

Figure 5-6 shows a sample configuration of an NDIS Mux driver (N to 1). The NDIS MUX driver exposes two virtual adapters for one physical adapter in the system. The Notify Object allows only Protocol A and Protocol B to bind to Virtual Adapter 1 exposed by the driver. Protocol B and Protocol C are allowed to bind to Virtual Adapter 2.

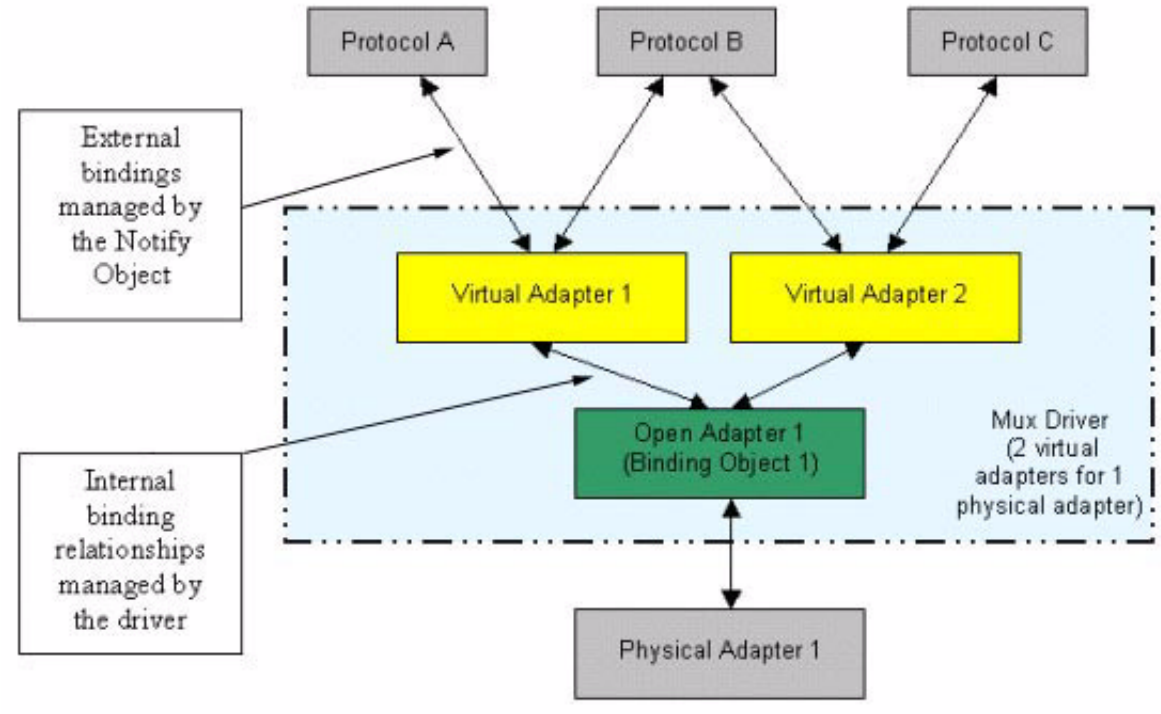


Figure 5-6. NDIS Mux Driver (N to 1) Sample Configuration

NDIS Intermediate (Mux) Drivers

DriverNetworks generic intermediate (NDIS Mux) drivers derive an adapter class from the **KNdisMiniAdapter** class and declare the required signatures for handler methods, such as **Initialize()**, **Halt()**, and **Send()**. These drivers also derive a binding class from the **KNdisProtocolBinding** class and declare the required signatures for handler methods, such as **Receive()**. The naming convention for the handlers closely match those from the DDK. For example, **Initialize()** corresponds to **MiniportInitialize()** and **Receive()** corresponds to **ProtocolReceive()**.

The UML class diagram in Figure 5-7 depicts the basic relationships between the **DriverNetworks** framework classes and classes implemented by the driver writer.

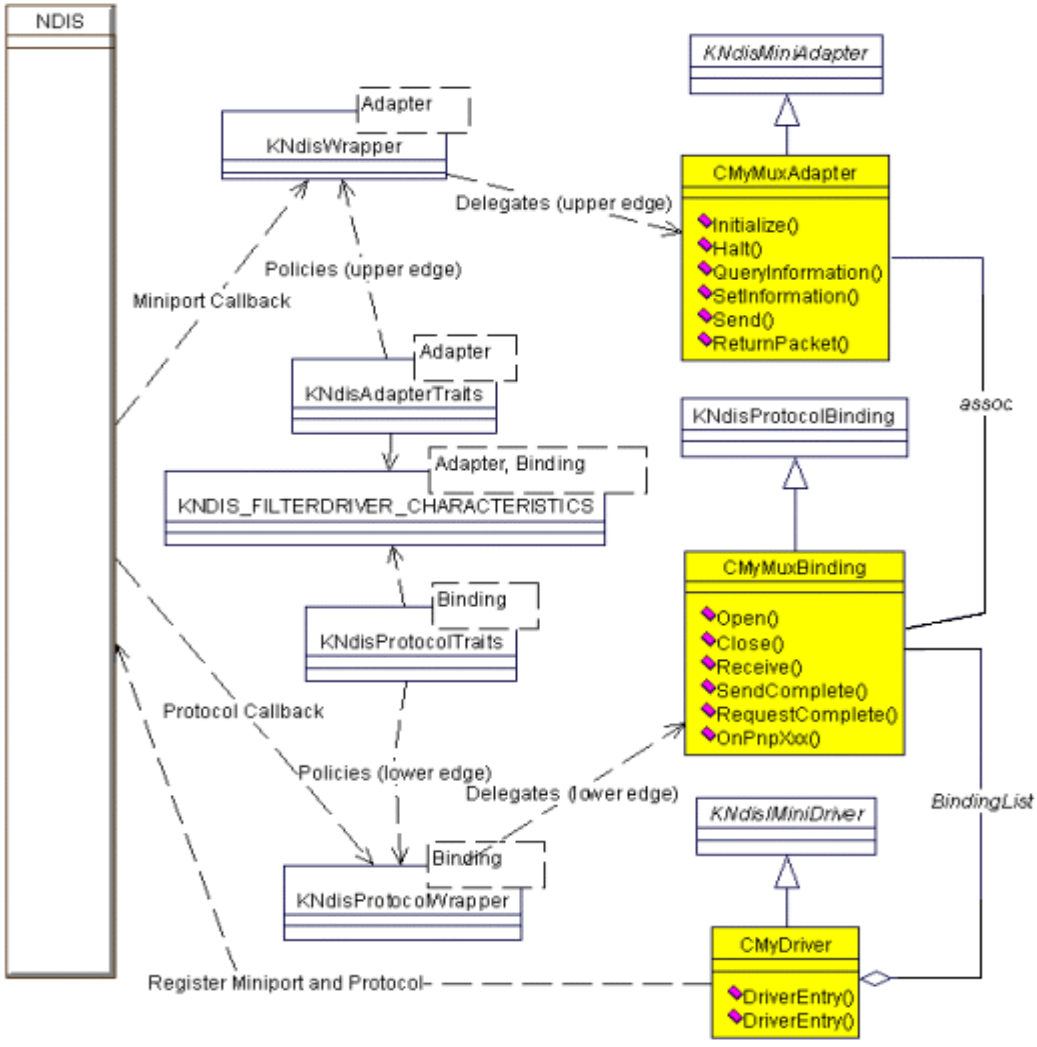


Figure 5-7. NDIS Intermediate (Mux) Drivers

The driver writer provides the following main classes:

- **KNdisiMiniDriver**-derived driver class
- **KNdisMiniAdapter**-derived adapter class.
- **KNdisProtocolBinding**-derived binding class.

Note: The **Network Driver Wizard** application automatically generates these classes for the NDIS Generic (MUX) Intermediate driver project.

Upon a driver load, the framework calls the driver class' **DriverEntry()** method, which registers the adapter with NDIS. The registration is handled by the helper template class **KNDIS_FILTERDRIVER_CHARACTERISTICS**, which passes to NDIS a set of handlers that the adapter supports. IM drivers dually act as both protocols for underlying real miniports and miniports for real upper-layer protocols. Thus, IM drivers must register two sets of NDIS handlers: MiniportXxxx and ProtocolXxxx. The default sets of handlers along with other driver characteristics are specified in the helper template classes, **KNdisAdapterTraits** and **KNdisProtocolTraits**. Each intermediate driver project includes a header file, `Characteristics.h`, which optionally redefines some of those characteristics (e.g., the set of handlers) for the specific adapter.

The intermediate adapter handlers are called from the wrapper class **KNdisWrapper**. **KNdisWrapper** delegates higher-layer NDIS requests issued by the real protocol, which is exactly the same way as for the miniport framework. On the other hand, **KNdisProtocolWrapper** delegates indications posted by the real NIC. The delegation from the real NIC is implemented via the **KNdisProtocolBinding**-derived class, which abstracts the NDIS binding object.

The **KNdisMiniAdapter**-derived **Send()** handler intercepts the packets coming from the real protocol down to the network. The **KNdisProtocolBinding**-derived **Receive()** handler intercepts packets indicated by the real NIC miniport to the protocols. In both cases, the driver writer can allocate a fresh packet descriptor to repackage the data to be sent or received.

The driver must maintain the internal binding relationships between Adapter and Binding objects. In a 1 to N Mux driver, there will be multiple **KNdisProtocolBinding**-derived objects for the **KNdisMiniAdapter**-derived object. The driver must “choose” which **KNdisProtocolBinding**-derived object to send a packet in its **KNdisMiniAdapter**-derived **Send()** handler. This scenario can be useful for a load balancing or failover situation. In an N to 1 Mux driver, there will be multiple **KNdisMiniAdapter**-derived objects for the **KNdisProtocolBinding**-derived object. The driver must “choose” which **KNdisMiniAdapter** object to indicate a packet. The wizard generates code which will use the packet filter of the adapter that the protocol set for each virtual adapter. So this means that the driver manages the internal bindings of the intermediate driver by choosing which path a packet should take. The wizard-generated code implements a simple one-to-one relationship between Adapters and Bindings.

NDIS Filter Drivers

DriverNetworks filter classes derive from the KNdisFilterAdapter class and implement the virtual **OnXxx()** handlers, such as **OnInitialize()** and **OnSend()**, called from the base.

The UML class diagram in Figure 5-8 depicts the basic relationships between the **DriverNetworks** framework classes and classes implemented by the driver writer.

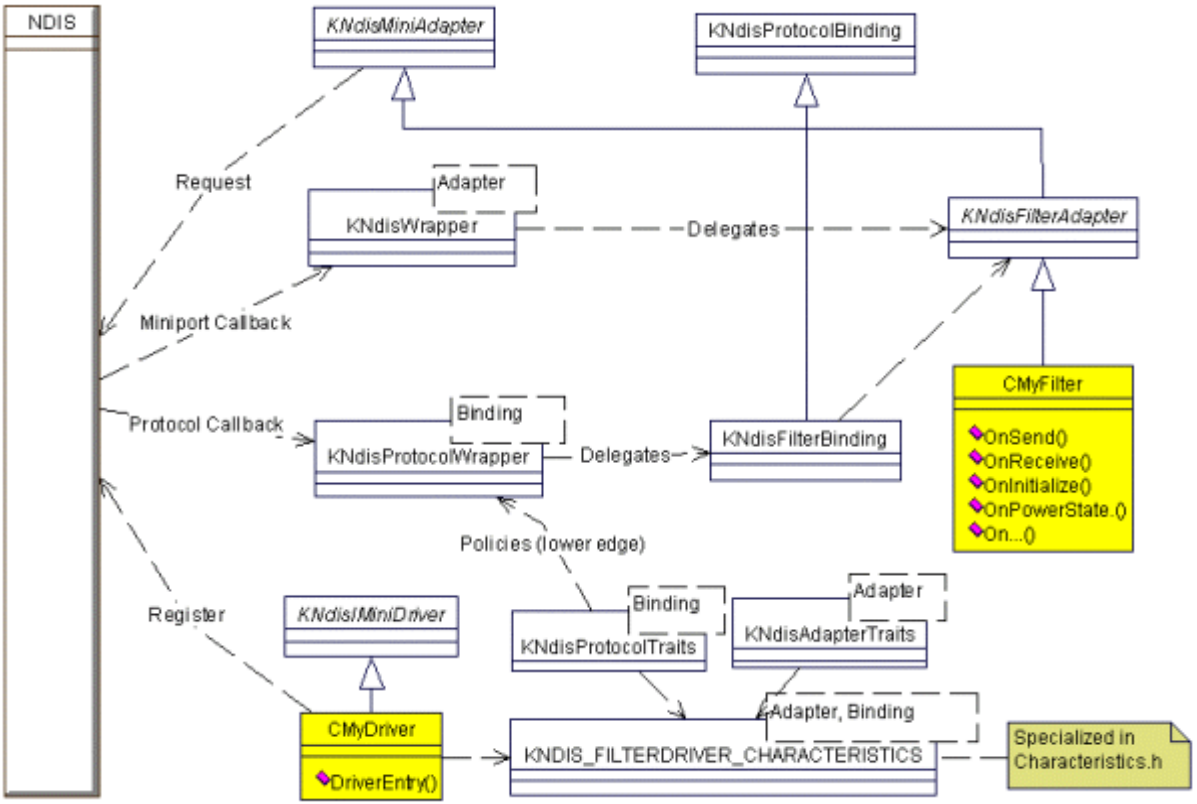


Figure 5-8. NDIS Filter Drivers

The driver writer provides two main classes: a KNdisIMiniDriver-derived driver class and a KNdisFilterAdapter-derived adapter class. The Network Driver Wizard application automatically generates both classes for the NDIS driver project.

Upon a driver load, the framework calls the adapter's **DriverEntry()** method, which registers the adapter with NDIS. The registration is handled by the helper template class KNDIS_FILTERDRIVER_CHARACTERISTICS, which passes to NDIS a set of handlers that the adapter supports. IM drivers dually act as both protocols for underlying *real* miniports and miniports for *real* upper-layer protocols. Thus, IM drivers must register two sets of NDIS handlers: MiniportXxxx and ProtocolXxxx. The default sets of handlers along with other driver characteristics are specified in the helper template classes, KNdisAdapterTraits and KNdisProtocolTraits. Each filter driver project includes a header file, `Characteristics.h`, which optionally redefines some of those characteristics (e.g., the set of handlers) for the specific adapter.

The filter adapter registration presents a rather complicated interaction between NDIS and the IM driver. The sequence is handled by the framework. When the framework detects a possible binding between a higher layer protocol and a lower-layer NIC, it instantiates a KNdisFilterAdapter derived class (e.g., CMyFilter) and calls its **OnInitialize()** handler. Returning NDIS_STATUS_SUCCESS from **OnInitialize()** effectively forges the *real-protocol » filter » real NIC* binding. From that point on, the IM driver is situated on the way of all packets and status indications traveling between the real protocol and miniport.

The filter adapter handlers are called from two wrapper classes, KNdisWrapper and KNdisProtocolWrapper. KNdisWrapper delegates higher-layer NDIS requests issued by the real protocol, which is exactly the same way as for the miniport framework. On the other hand, KNdisProtocolWrapper delegates indications posted by the real NIC. The delegation from the real NIC is implemented via another framework class, KNdisFilterBinding, which abstracts the NDIS binding object.

The **OnSend()** handler intercepts the packets coming from the real protocol down to the network. The **OnReceive()** handler intercepts packets indicated by the real NIC miniport to the protocols. In both cases the filter adapter is presented with both original packet descriptor and a fresh packet descriptor. The fresh packet descriptor is then used to *repackage* the data to be sent or received and forward it accordingly down to the network or up to the protocols. Additional **OnXxx()** handlers intercept some other important events, such as **OnPowerState()**, which can be monitored by the filter driver.

The destruction of KNdisFilterAdapter-derived objects is also handled by the framework. The **OnHalt()** handler is called immediately before the destruction. The filter class performs a driver-specific shutdown, resource deallocation, etc. The **OnHalt()** is typically called when the filter (or underlying NIC) is disabled via the Control Panel or on system shutdown.

NDIS Filter Driver Handler Signatures

The client's filter class declares and implements the following methods-handlers. Mandatory handlers appear in bold:

```
class CMyFilterAdapter : public KNdisFilterAdapter {
 public:
    // First call on a new object: a chance to accept or reject
    the binding
     NDIS_STATUS OnInitialize(const KNdisMedium&, KNdis-
    Config&);

     // Last call on the object about to be destroyed
     VOID OnHalt();

     // Resetting the NIC:
     NDIS_STATUS OnReset();

     // Intercepting upper-layer send. MUST be implemented
     NDIS_STATUS OnSend(const KNdisPacket& Original,
         KNdisPacket& Repackaged
         ) ;

// Notifying on privately submitted Tx packet completions:
 VOID OnSendComplete(PNDIS_PACKET, NDIS_STATUS) ;

 // Notifying on return on privately submitted Rx packet indica-
tions:
 VOID OnReturnPacket(IN PNDIS_PACKET);

 // Intercepting upper-layer OID requests:
 NDIS_STATUS OnQuery(KNdisRequest&, NDIS_STATUS Returned-
Status);
 NDIS_STATUS OnSet(KNdisRequest&);

 // Lower-edge: delegated from KNdisProtocolWrapper<Binding>
 // Intercetpting lower-layer receive. MUST be implemented.
 NDIS_STATUS OnReceive(
     const KNdisPacket& Original,
     KNdisPacket& Repackaged
     ) ;
```

```
// Intercepting partial receive indication:
NDIS_STATUS OnReceive(IN PVOID /*HeaderBuffer*/,
    IN OUT UINT /*HeaderBufferSize*/,
    IN PVOID /*LookAheadBuffer*/,
    IN OUT UINT /*LookaheadBufferSize*/,
  IN UINT /*PacketSize*/
    ) ;

// Intercepting lower-layer Status indications
NDIS_STATUS OnStatusIndication(
    NDIS_STATUS Status,
    IN OUT PVOID*,
    UINT*
    );
void OnStatusIndicationComplete();

// Intercepting protocol reconfigurations (PnP)
NDIS_STATUS OnReconfiguration(PCWSTR /*Section*/);
// Intercepting NIC power change
void OnPowerState(NDIS_DEVICE_POWER_STATE NewState,
    BOOLEAN bRequestNotNotification);

};
```

NDIS Protocol Driver Framework

Tip: *The easiest way to become familiar with the framework is to start the Network Driver Wizard, generate a skeleton NDIS miniport or NDIS IM driver project, and explore the generated source code using the Visual Studio class browser.*

DriverNetworks supports development of NDIS 5.0 Protocol drivers for the Windows 2000 and XP platforms. The NDIS Protocol drivers come in two main flavors: packet drivers and transport drivers.

Packet drivers are layered between the I/O Manager and NDIS Miniport drivers. At the lower edge the packet driver registers with the NDIS library as an NDIS Protocol. This allows the packet driver to bind to the existing network interface adapters registered as NDIS Miniports. The NDIS library facilitates the process of establishing bindings between NDIS protocol and miniport instances. Once a binding is established the packet driver can send and receive packets to the NIC over this binding.

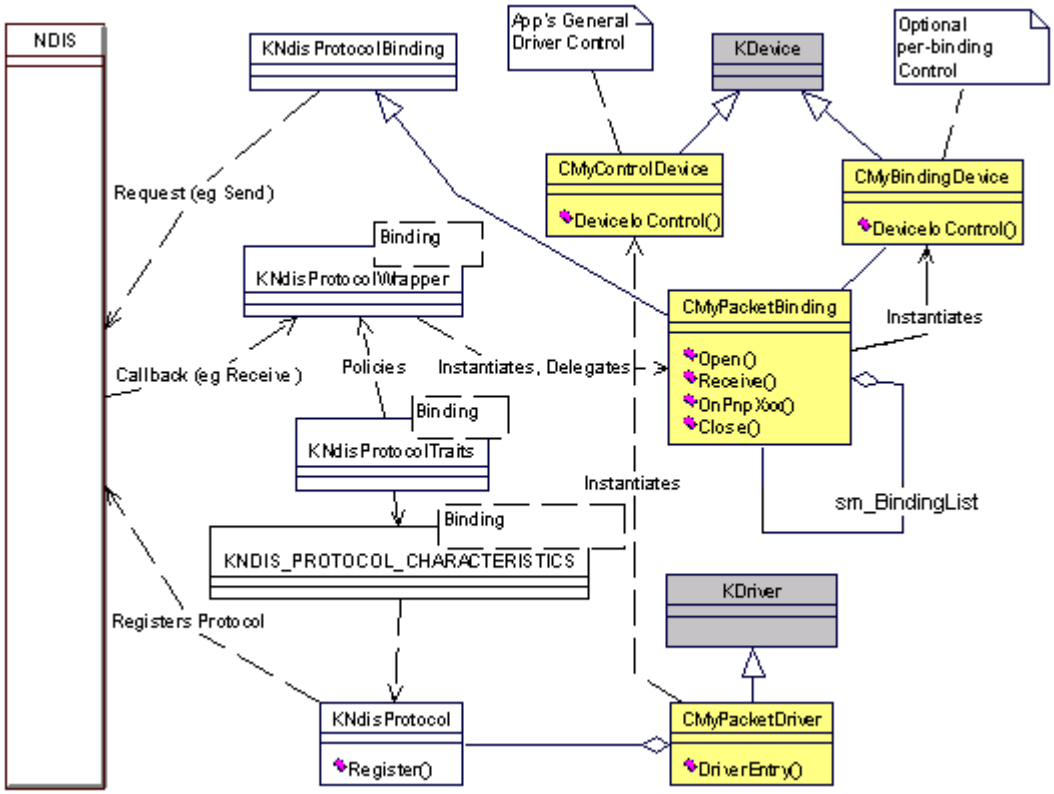


Figure 5-9. NDIS Protocol Packet Driver

At the upper edge the packet driver exposes a conventional WDM I/O device interface. This interface (usually, a proprietary one) allows Win32 applications to talk to the driver and, thus, send and receive packets directly to/from the bound NICs, and control the NICs in some application-specific fashion.

Packet Drivers

A programmer develops a packet driver to implement some custom way of accessing and controlling the network adapters on the system. A typical example of a packet driver would be a network monitor. A network monitor driver puts the bound NIC(s) into the "promiscuous mode," which allows it to intercept all the packets passing over the LAN segment. The packets are then picked up by a controlling application, which parses and displays them to the user. **DriverNetworks** ships with the sample packet driver, **NmPacket**, which demonstrates an implementation of such driver.

Transport Drivers

Transport drivers (also known as TDI transports) are a more complex type of protocol drivers. Transport drivers are layered between NDIS and TDI (Transport Driver Interface) subsystems. At the lower edge transport drivers register with NDIS and establish bindings to the underlying miniports pretty much the same way as packet drivers do. At the upper edge, however, transport drivers implement the I/O device interface compliant to the TDI specification. The TDI specification defines a set of standardized internal I/O control interfaces for connection-oriented and connectionless transport services. This allows higher layer drivers, known as TDI clients, gain access to the network in a transport-independent fashion.

An example of a TDI Client would be a file system redirector. The TDI layer allows the TDI client to execute a higher level protocol over any TDI-compliant transport. The example of such transport would be the Microsoft `tcpip.sys` driver.

In addition to the network services transport drivers participate in the TDI Plug-and-Play system. A transport driver registers its dynamic internal objects (such as transport addresses) with the TDI and forwards PnP and PM events from the bound NICs up to the TDI subsystem. The TDI subsystem, in turn, notifies PnP-enabled TDI clients on these changes in the system state.

Figure 5-10 shows a diagram of an NDIS Protocol Transport Driver.

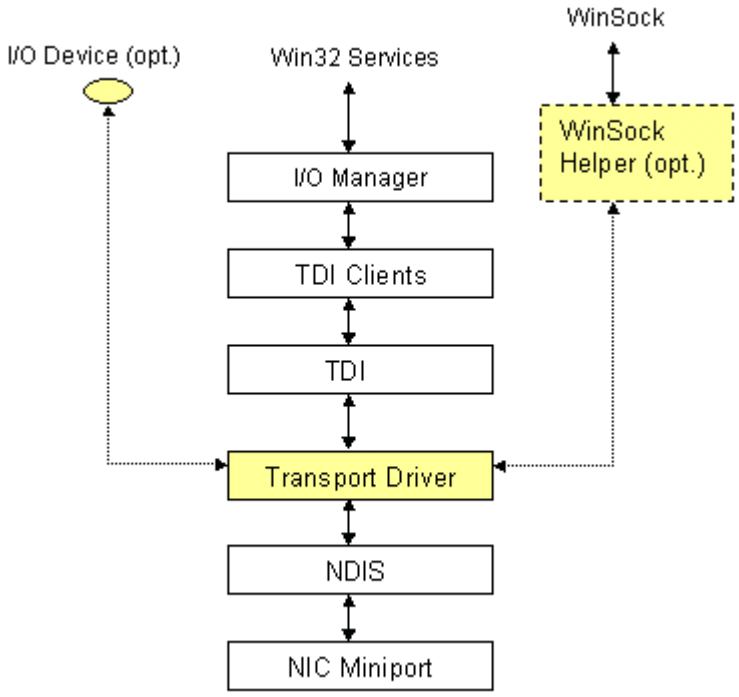


Figure 5-10. NDIS Protocol Transport Driver

Optionally, the transport driver might come with support for WinSock. By providing a so-called WinSock helper DLL, the transport driver makes its services accessible via the standard WinSock API to Win32 applications.

Why Would One Need a Transport Driver?

A programmer develops a transport driver to implement a transport (layer 4) protocol and make the protocol available to the existing TDI clients and, perhaps, Winsock-compliant Windows applications. For instance, one might want to implement a customized (e.g. encrypted) version of the UDP. Or, add an implementation of a standard transport protocol not shipped with Windows. Of course, one should use discretion whether implementing a transport driver would be the best way of getting the required functionality.

The good news is that **DriverNetworks** makes it much easier to implement either kind of NDIS protocol driver.

NDIS Packet Driver Framework

The packet driver represents a **DriverWorks** "NT4-style" driver, which makes use of the **DriverNetworks** class framework. Figure 5-11 displays the UML diagram of a typical driver implementation.

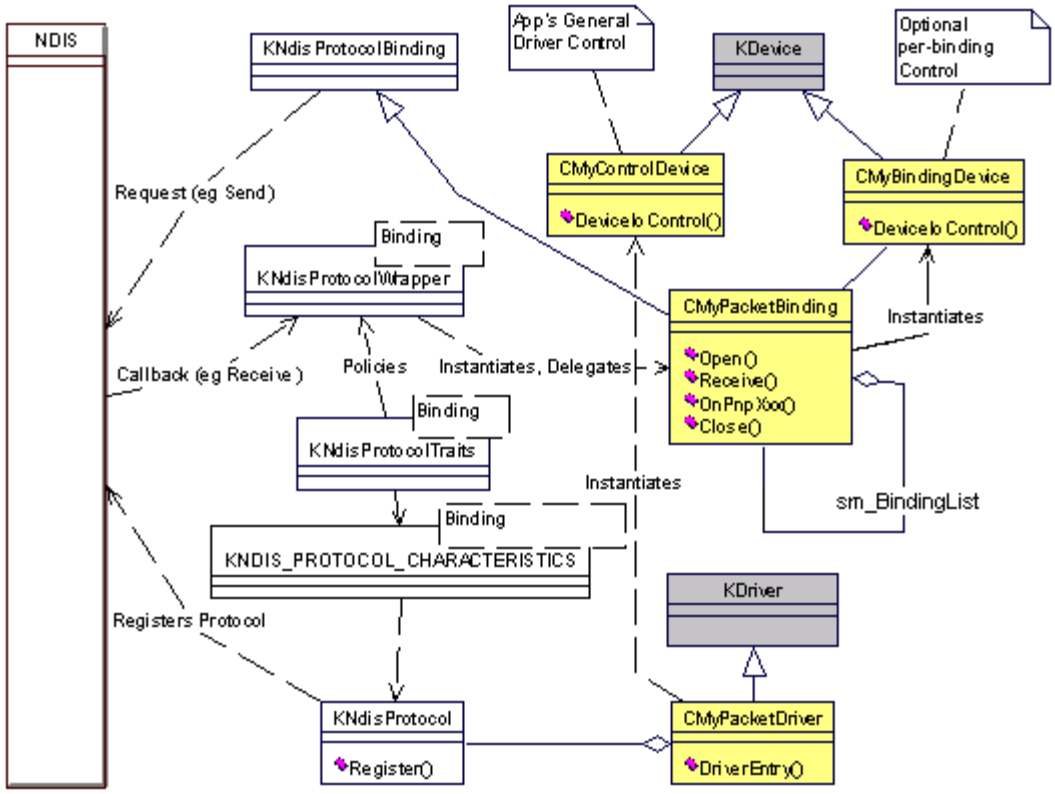


Figure 5-11. NDIS Packet Driver Framework

The driver writer provides the following main classes:

- As for all **DriverWorks** drivers, there is a **KDriver**-derived class, **CMyPacketDriver**. It is a singleton instantiated by the **DriveWorks** framework during driver initialization.
- A DriverWorks **KDevice**-derived class, **CMyControlDevice**, which exposes a device object. This device object let the user mode applications access the driver and its binding objects using a proprietary device
  I/O interface.

- A **KNdisProtocolBinding**-derived class, **CMyPacketBinding**. Instances of this class represent NDIS binding objects. Each binding object represents a logical association between your driver and underlying NIC miniport. The driver writer implements certain handler methods, such as **Receive()**, to process network events indicated from the NIC miniport. The base class provides a number of service methods, such as **Send()** to pass packets and control actions to the NIC; optionally, another **DriverWorks KDevice**-derived class, **CMyBindingDevice** each instance of which gets associated with a certain binding. Those devices provide a per-binding I/O control interface to the driver.

Tip: *The best way to study the underpinning of the driver is to generate a packet driver project using the* ***Network Driver Wizard*** *and examine the class hierarchy in the Visual Studio IDE.*

Upon the driver load, the framework calls the driver's **CMyPacketDriver::DriverEntry()** method, which (a) creates the **CMyControlDevice** instance, and (b) registers itself as an NDIS protocol. The latter is accomplished by registering the set of **KNdisProtocolWrapper<CMyPacketBinding>** handlers defined by the instantiation of **KNDIS_PROTOCOL_CHARACTERISTICS** template. The set of handlers and some other aspects of the framework behavior are controlled by policies defined in **KNdisProtocolTraits** template. The driver writer can redefine the default policies by specializing them for **KNdisProtocolTraits< CMyPacketBinding>** implementation; file `Characteristics.h` in the driver project hosts the set of such specializations (if any).

WinNT4, Win2K, and WinXP support plug-and-play NDIS protocols. This means that the network bindings get created and destroyed dynamically as the underlying NICs appear and disappear. As a network adapter gets detected by the system, NDIS calls into a **KNdisProtocolWrapper**-registered **ProtocolOpenAdapter** method, which, in turn, instantiates an instance of **CMyPacketBinding** and transfers control into **CMyPacketBinding::Open()** method. At this point, if the driver decides to bind to the NIC it returns NDIS_STATUS_SUCCESS, and the framework proceeds with opening the NIC of interest. When the binding is finally established (or the attempt has failed) the framework calls **CMyPacketBinding::OpenComplete()**. At this point, if the binding has been successfully established, the driver might further allocate resources to start using the binding. For instance, the driver might want to create a new device object (**CMyBindingDevice**) to represent this particular binding. The driver might also allocate buffer and packet pools (see **KNdisBufferPool**, **KNdisPacketPool**) to use while sending and receiving data over the binding. In addition, the driver might query certain parameters of the underlying NIC, such as its physical MAC address, line speed, etc. using NDIS OID requests.

The sequence of events in a typical use case might look as follows. Assume the application wants to send a frame with a proprietary format onto the network. The app opens a handle to the **CMyBindingDevice** using Win32's **CreateFile()**. Then it issues a **DeviceIoControl()** passing the buffer with the frame it filled in. The driver gets control, allocates a packet descriptor from its packet pool, retrieves the MDL describing the frame from the **DeviceIoControl** IRP, chains the MDL into the packet descriptor and passes the descriptor into **KNdisProtocolBinding::Send()** method of the associated binding object. The packet is passed down to the miniport by NDIS. When the miniport completes processing the packet, it notifies NDIS, which eventually notifies the driver with transferring control into **CMyPacketBinding::SendComplete()**. At this point the driver completes the IRP, and thus, the applications **DeviceIoControl()** request.

Whenever the bound NIC gets removed from the system, the framework notifies the driver by calling **CMyPacketBinding::Close()**. At this point the driver releases the resources acquired on **Open()** and/or **OpenComplete()**, including deleting the **CMyBindingDevice** object.

Note that before a network adapter can be removed, the system notifies all the bound protocols with **NetEventQueryRemoveDevice** event. In **DriverNetworks**, this results in the invocation of **CMyPacketBinding::OnPnpQueryRemoveDevice()** handler. This handler might disable removing the network adapter (by returning NDIS_STATUS_FAILURE) if the current driver state does not allow to perform the removal safely.

NDIS Transport Driver Framework

The transport driver represents a **DriverWorks** "NT4-style" driver, which makes use of **DriverNetworks** class framework. Because of the complexity and the variety of design choices for a transport driver, **DriverNetworks** does not attempt to provide a "complete" set of transport driver's components in the form of base classes or templates. Instead, **DriverNetworks** suggests a set of loosely coupled model classes, which might comprise a typical transport driver design. These model classes (declared in file `KTransport.h`) define the conceptual interfaces which a transport's components, such as a control channel or a transport address object, expose.

Figure 5-12 displays the UML diagram of a typical driver implementation.

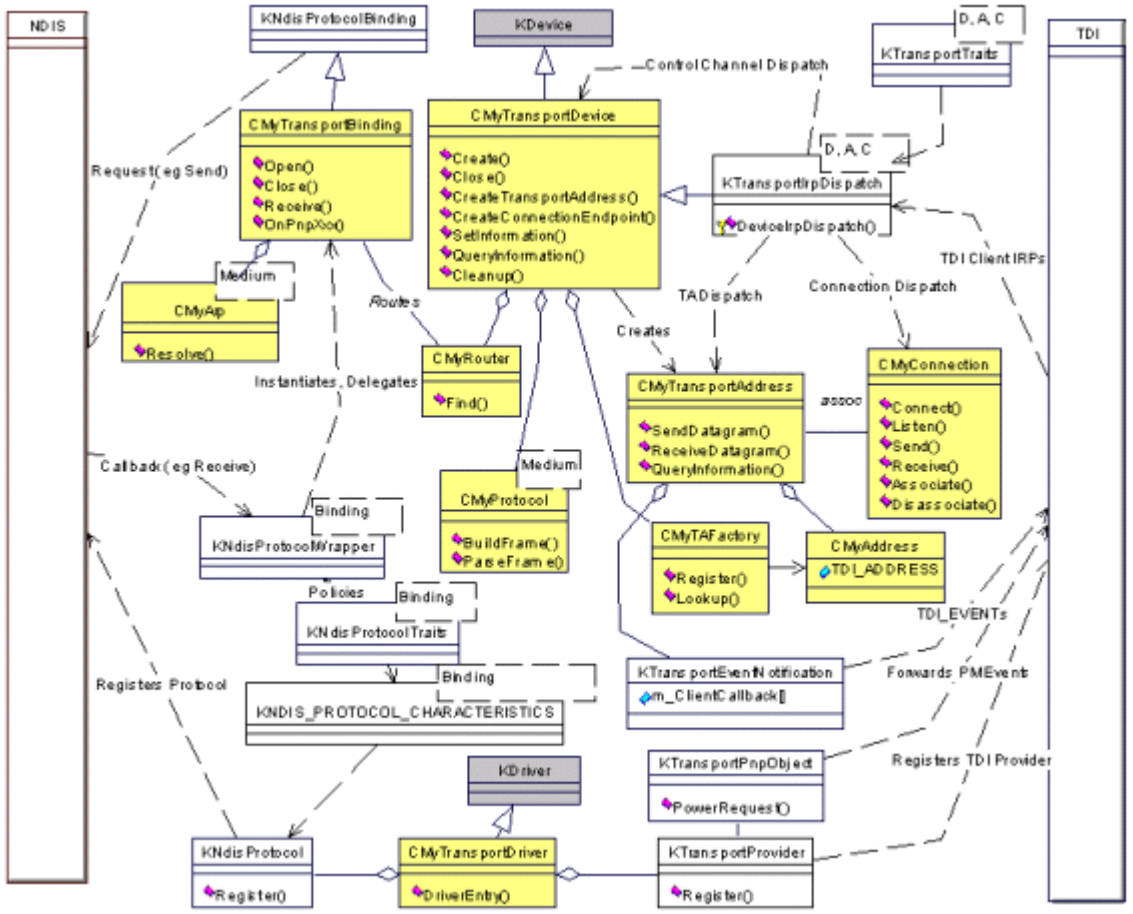


Figure 5-12. NDIS Transport Driver Framework

Note: The best way to study the underpinning of the driver is to generate a transport driver project using the **Network Driver Wizard** and examine the class hierarchy in the Visual Studio IDE.

The framework provides no implementation for the model classes and `KTransport.h` is not included in any .cpp file. **DriverNetworks**, however, uses these models to produce the transport driver reference design generated by the Network Driver Wizard tool. The driver writer in no way is forced to follow the reference design. He or she might choose to change the model interfaces, provide more efficient implementations or even get rid of some components altogether. Certain user-provided classes, however, are required. Now review both sets of classes.

The driver writer must provide the following classes:

- A DriverWorks **KDriver**-derived class, **CMyTransportDriver**. It is a singleton instantiated by the **DriverWorks** framework during driver initialization;
- A DriverWorks **KDevice**-derived class, **CMyTransportDevice**. This class wraps a device object which serves as a target for all TDI IRPs. In **DriverNetworks**, this class also implements the control channel of the transport;
- A **KNdisProtocolBinding**-derived class, **CMyTransportBinding**. Instances of this class represent NDIS binding objects. Each binding object represents a logical association between the transport driver and underlying NIC miniport. The driver writer implements the handler methods, such as **Receive()**, to process network events indicated from the NIC miniport. The base class provides a number of service methods, such as **Send()** to pass packets and control actions to the NIC;
- **CMyTransportAddress** class modeled after **KTransportAddress**. Instances of this class represent transport address objects and serve as a target for the TDI IRPs directed to TDI Transport Address Objects;
- **CMyConnectionEndpoint** class modeled after **KTransportConnection**. Instances of this class represent connection endpoint objects and serve as a target for the TDI IRPs directed to TDI Connection Endpoint Object. This class is optional if your transport supports only connectionless service;
- **CMyAddress** class modeled after **KTAddress** to represent a network address value. **CMyAddress** is an instantiation of **KTA_ADDRESS** template, which conveniently wraps the TDI **TA_ADDRESS** structure.

In addition, for any non-trivial transport, the driver writer might provide the following classes:

- **CMyArp** class modeled after **KTransportArp**. This class provides a transport-specific method of mapping network addresses to physical MAC addresses (a.k.a. Address Resolution Protocol). If the transport needs an ARP, a separate implementation of CMyArp must be provided for each NDIS medium (802.3, FDDI, etc.) the transport supports;

- **CMyRouter** class modeled after **KTransportRouter**. This class provides a transport-specific method of mapping network addresses to bindings (network interfaces). If the transport is routable, ie it can operate over a set of bound network interfaces, it must provide a way of choosing bindings based on the attributes of the outgoing packets (e.g., the destination address);
- **CMyProtocol** class modeled after **KTransportProtocol**. This is an utility class, which knows how to build and parse frames going to and from the network for each NDIS medium type the transport supports. For example, for the 802.3 medium, this class would know the format of the Ethernet frame.
- **CMyTAFactory** class modeled after **KTransportTAFactory**. This class manages the lifetime of local network addresses. Note that TDI is allowed to maintain N-to-1 mapping between transport address objects (**CMyTransportAddrees**s) and local network addresses (**CMyAddress**).

Upon the driver load, the framework calls the driver's **CMyTransportDriver::DriverEntry()** method, which (a) creates the **CMyTransportDevice** instance, and (b) registers itself as an NDIS protocol. The latter is done exactly the same way as for the packet drivers. The **CMyTransportDevice** instance gets created by instantiating the **KTransportIrpDispatch** template, which is parametrized by transport device (D), transport address (A) and connection context (C) types. **KTransportIrpDispatch** "cracks" TDI IRPs and dispatches them into appropriate instances and methods on **CMyTransportDevice**, **CMyTransportAddress** or **CMyConnectionEndpoint** accordingly.

A transport driver also registers itself with the TDI PnP system by calling **KTransportProvider::Register()**. When the transport gets bound to all possible adapters the framework calls **CMyTransportBinding:: OnPnpBindListComplete()**. At this point, the driver further configures itself based on the characteristics of the bound NICs and calls **KTransportProvider::Ready()** to declare the transport available to the PnP-enabled TDI clients.

Consider a couple of use cases with the transport driver.

Creating a Transport Address Object

A TDI client issues **IRP_MJ_CREATE** with the **FILE_FULL_EA_INFORMATION** context specifying "**TransportAddress**". **KTransportIrpDispatch< CMyTransportDevice>** parses the IRP and calls **CMyTransportDevice::CreateTransportAddress()**. This method retrieves the local address (**CMyAddress**) value from the IRP and calls **CMyTAFactory()** to register this (perhaps, new) local address. Then it instantiates a new **CMyTransportAddress** object referencing this local address and returns the pointer to it. **KTransportIrpDispatch** saves the returned transport object pointer in the file context field of the device file object and completes the IRP.

Sending a datagram: A TDI client issues **IRP_MJ_INTERNAL_DEVICE_CONTROL** with the minor function **TDI_SEND_DATAGRAM**. **KTransportIrpDispatch< CMyTransportDevice>** parses the IRP and calls **CMyTransportAddress::SendDatagram()**. This method first figures what binding instance (**CMyTransportBinding**) to choose to send the datagram over. A **CMyRouter** instance is used for this. Based on the medium type of the chosen binding, the driver uses appropriate **CMyArp** instance to resolve the destination network address into the physical MAC address. Then the driver builds the network frame packet using the appropriate instance of **CMyProtocol** and chains the user's buffer to the packet. Finally, it forwards the packet down the binding using **KNdisProtocolBinding::Send()**. When the underlying miniport completes the packet the framework calls **CMyTransportBinding::SendComplete()**. At this point the driver recycles the resources allocated for the transfer and completes the TDI client's IRP.

Notify Object Framework

A Notify Object is a COM server dll which is used by the Network class installer and Network control panel application. Notify Objects are installed for a network component based on entries made to the .inf file for the network component. Protocol and intermediate drivers can use a Notify Object to display properties in the network control panel for the user to configure. Generic Intermediate drivers are required to use a Notify Object to control the binding relationships external to the driver and to install upper edge virtual adapters that the driver exposes. For an Intermediate Filter driver, this is handled by the system. Notify Objects consist of various optionally implemented COM interfaces providing network subsystem callbacks for various events; network installation, network setup, network binding notifications, and to display properties in the UI. The Notify Object is given access to the Network Configuration Subsystem through COM interfaces. See your DDK reference for details about the Notify Object.

The UML class diagram shown in Figure 5-13 depicts the basic relationships between the **DriverNetworks** Notify Object framework classes, Notify Object Interfaces, ATL (active template library) COM classes, and classes implemented by the driver writer. The framework provides wrapper classes around the COM interfaces to simplify writing a Notify Object.

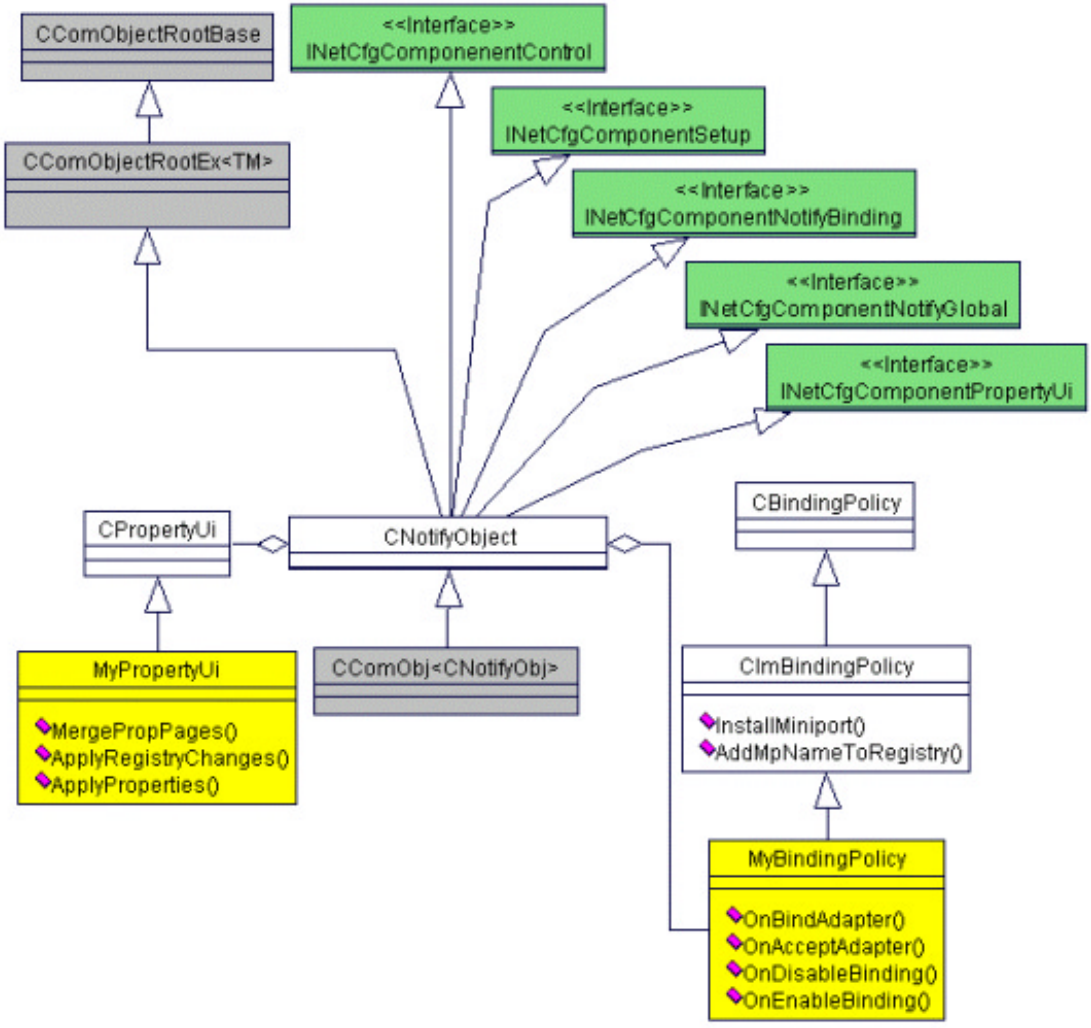


Figure 5-13. Notify Object Framework

The framework provides one main class: a **CNotifyObject** template class that has handlers for all the notifications from the Network Control Subsystem. The framework hides most of the COM details required to implement the Notify Object. The task of the driver writer is to implement some of the handlers. CNotifyObject will delegate various callbacks to template classes provided by the user. CNotifyObject requires a BindingPolicy class which provides handlers specific to managing bindings for the driver. CNotifyObject also requires a CPropertyUI class which provides handlers specific to displaying property pages in the Network Control Panel applet. The driver writer provides a CBindingPolicy derived class which has handlers required to be implemented by the framework for binding policy decisions. For generic NDIS intermediate drivers, the driver writer can derive from class CImBindingPolicy which provides value add handlers and services to make it easy to install virtual adapters, alter system bindings, and make the required registry entries. The Network Driver Wizard automatically generates the class declarations.

The Notify Object for a generic intermediate driver will receive binding notifications for each binding of its protocol edge to real miniports in the system for which there is a match between the binding interfaces specified in their .inf files. The framework will call **OnAcceptAdapter()** for each binding between our intermediate driver's protocol edge and real adapters in the system. The default behavior is to accept binding to all adapters. This method can be overridden to provide different behavior (i.e., to be choosier with the adapters to which we bind). The framework will call **OnAcceptProtocol()** for each binding between real protocols and our intermediate driver's miniport edge. Once again, the default behavior is to accept all protocols, but this method can be overridden to alter this behavior.

For each accepted real adapter, the framework will call the **OnBindAdapter()** handler whose base class implementation is to install one virtual adapter instance for the miniport portion of the intermediate driver. The virtual adapter instances are installed by the Notify Object. Also, the virtual adapter instance name has to be added to the Protocol's registry section for the real adapter name key as data for the "upper bindings" value. The **InstallMiniport()** method performs these operations. For each binding notification, the **OnBindAdapter()** handler installs a virtual miniport by calling **InstallMiniport()**. This results in a 1 to 1 generic intermediate driver, one virtual adapter instance for one real adapter in the system. The **OnBindAdapter()** handler can be overridden to implement *N to 1* or *1 to N* multiplexed configurations. The **InstallMiniport()** method can be used to install a virtual adapter. For a 1 to N case, if a virtual miniport is not to be installed for a particular real adapter, an existing virtual adapter instance name must be written to the registry for the driver binding to succeed. The **AddMpToReg()** method can be used to add an existing virtual miniport instance to the registry for a particular adapter. This instance name is important since the driver will read this registry value during Bind operations for the driver. For an N to 1 case, multiple virtual adapter instances can be installed per notification. Thus, the **OnBindAdapter()** handler can call **InstallMiniport()** n times. The framework takes care of properly handling binding notifications between the virtual miniport and protocol. For each real adapter in the system that we bind, it is the Notify Object's responsibility to alter existing binding relationships (i.e., real protocols bound to real adapters). For Intermediate Filter drivers, the system automatically disables these bindings. The framework calls the **OnDisableBinding()** handler to request the policy for disabling the binding between a real protocol and a real adapter in the system. The default base class implementation is to disable the binding. This method can be overridden to provide a different behavior.

The CNotifyObject class will store all of the system binding changes to the registry made by the Notify Object. In this way, if the network component is uninstalled, the Notify Object can "undo" all of these changes, restoring the system to its previous state.

Notify Objects can display properties in the Network Control Panel applications. By implementing the CPropertyUi template class, the Notify Object supplies a property page for displaying UI controls for configuration parameters. When the Notify Object detects that the user has changed properties, it writes these changes to the registry.

TDI Client (DriverSockets) Framework

DriverSockets is a C++ framework for kernel mode drivers requiring TCP/IP connectivity. DriverSockets abstracts the Windows Transport Driver Interface (TDI) as a compact set of classes that provide a socket-like API for client drivers.

DriverSockets hides significant differences found in the TDI client interface implementation between NT and Win9x systems. The top-level classes are defined in an OS-independent manner. As a result, the same source code will compile and work under Windows NT (NT), Windows 2000 (W2K), and Windows 95/98 (Win9x) systems. Refer to the following sections for detailed information on DriverSockets frameworks:

- Protocols and Address Families
- Programming Model
- Examples

Figure 5-14 displays the TDI Client (DriverSockets) Framework.

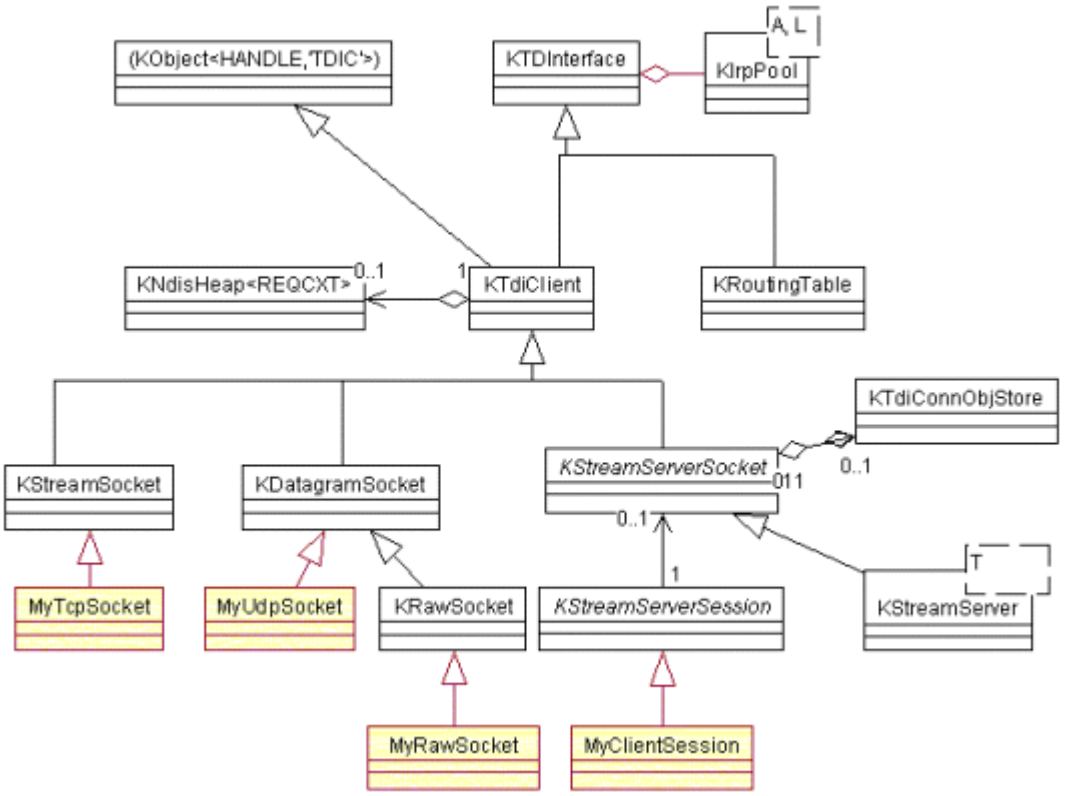


Figure 5-14. TDI Client (DriverSockets) Framework

Protocols and Address Families

DriverSockets provides two main base classes:

- KDatagramSocket - connectionless socket types (e.g., UDP)
- KStreamSocket - connection-oriented socket types (e.g., TCP)

Another base class, KStreamServerSocket, is used to facilitate multiple-session support when implementing server-side, connection-oriented sockets (kernel mode servers).

By default, DriverSockets runs over TCP and UDP transports, and, therefore uses an IPv4 addressing scheme (32-bit IP address and 16-bit port values). Generally, DriverSockets supports all the transports and address families registered with the TDI (e.g., NetBEUI). Those transports are accessed by overriding the default *szProvider* device name and providing an appropriately formed TRANSPORT_ADDRESS to the constructors for KDatagramSocket and KStreamSocket. This release, however, is primarily designed and tested for the IP-stack.

Programming Model

DriverSockets exploits an asynchronous event-driven programming model. The base classes, such as KDatagramSocket and KStreamSocket, provide a set of methods (e.g., **send()**), which request certain services from the TDI subsystem. The requests are conceptually asynchronous. For example, they typically return immediately after submitting the request (e.g., a packet to be sent) to the underlying transport.

The base classes also defines a set of overridable event handlers, which are called when a certain socket-related event has occurred. There are two types of TDI events: request completion and network indication. An example of a request completion is: when a **KStreamSocket::send()** request completes, **On_sendComplete()** is called. An example of a network indication is: when a packet is received over a TCP/IP connection, the **KStreamSocket::OnReceive()** handler is called.

The developer derives a specific socket class from the base class and implements the event handlers that process requests completions and network indications. DriverSockets runs in a context-free environment. For example, no worker-thread is associated with either request or event handler processing. The event handlers are executed at DISPATCH_LEVEL IRQL, which dictates certain limitations on how much processing may be accomplished within a handler without jeopardizing the overall system performance.

The DriverSockets programming model allows you to issue multiple requests without waiting for completion of an earlier-issued request. To keep track of the request completions, the **send()** methods in all the base classes include an extra parameter, a context pointer, which is opaque to the DriverSockets framework. The context pointer is passed to all completion handlers, so that you can correlate the completion event with the request and recycle the resources bound to this particular request.

DriverSockets Examples

The first example demonstrates the implementation of a trivial TCP-echo client.

```
class CCheckTcpEcho : public KStreamSocket {
 public:
   // ctor
   CCheckTcpEcho() : KStreamSocket() {};
   // Try to connect
   BOOLEAN IsFound(ULONG IpAddr)
   {
     m_Event.Reset();
     TDI_ADDRESS_IP addr = {htons(7), IpAddr};
     CTDI_CONNECTION_INFORMATION server(addr);
     if (connect(server) != TDI_PENDING)
       return FALSE; // TDI error
     m_Event.Wait();
     if (IsConnected()) {
       disconnect(); // drop the connection
       return TRUE;
     }
     else return FALSE; // no server found
   };
 protected:
   // Request Completions
   void On_connectComplete(
     PVOID pCxt,
     TDI_STATUS Status,
     uint ByteCount)
   {
     m_Event.Set(); // Status is TDI_SUCCESS when connected
   }
```

```
    // datd
        KNdisEvent m_Event;
};
```

CCheckTcpEcho provides a single public method, **IsFound**, which attempts to connect to an echo-server at the specified IP address. The method is assumed to run at the PASSIVE_LEVEL in a system (or worker) thread context. The method synchronizes with the handler **KStreamSocket::On_connectComplete()**, which is called when the connection attempt has completed, by using an `m_Event`.

The second example demonstrates the implementation of a trivial UDP echo socket.

```
class CCheckUdpEcho : public KDatagramSocket {
 public:
 // ctor
   CCheckUdpEcho() : KDatagramSocket() {}
 // Sending test msg to server
   BOOLEAN IsFound(ULONG IpAddr)
 {
   m_Event.Reset();
   TDI_ADDRESS_IP addr = {htons(7), IpAddr};
   CTDI_CONNECTION_INFORMATION server(addr);
   sendto(server, sm_szMessage, sizeof(sm_szMessage));
   if (m_Event.Wait(5000)) {
     return m_bReplyOk; // there was a reply
   }
   else return FALSE; // no reply
 };
 protected:
   // Datagram indication
   virtual uint OnReceive(
     uint AddressLength,
     PTRANSPORT_ADDRESS pTA,
     uint OptionsLength,
     PVOID Options,
     uint Indicated,
     uchar *Data
   )
```

```
    {
     m_bReplyOk = memcmp(sm_szMessage, Data, Indicated) ==
0;
     m_Event.Set();
     return Indicated;
    }
// data
   KNdisEvent m_Event;
   BOOLEAN m_bReplyOk;
   static char sm_szMessage[] = "My Test Message";
 };
```

CCheckUdpEcho provides a single public method, **IsFound**, which sends a short message to an echo-server at the specified IP address. The method is assumed to run at the PASSIVE_LEVEL in a system (or worker) thread context. The method synchronizes with the **KDatagramSocket::OnReceive()** handler, which is called when a datagram has arrived to the local port, by using an `m_Event`.

To review additional DriverSockets programming examples, refer to the "TDI Client (DriverSockets) Framework" on page 178 section.

DriverSockets vs. WinSock

DriverSockets provides a much lower-level programming model for network connectivity than WinSock and BSD sockets. This introduces both advantages and disadvantages.

The advantage is the flexibility. The developer can use DriverSockets to build any kind of KM driver, such as an NDIS (virtual) miniport, a higher-level standalone NT driver, a Win9x-specific VxD, or a file system filter driver.

The downside to this flexibility, however, is that the developer must contend with the asynchronous nature of the DriverSockets and provide the necessary synchronization and resource protection between the main code line and the DriverSockets event handlers. In addition, many utility features of WinSock are not yet implemented or supported in the current release.

Features of DriverSockets vs. Winsock

- Interface Layering
- Address Families, Protocols, and Socket Types
- Programming Models
- Name Resolution
- Socket Options
- Miscellaneous Helper Functions
- Multicasting
- QoS
- Plug-and-Play

Interface Layering

Winsock is implemented as a WOSA component with a DLL-provider/KMD combination layered on top of the TDI Client interface. The user calls into WinSock via the *winsock2.dll* exports.

DriverSockets is a static library built into the target driver. The target driver talks directly to a transport driver, using IRPs under NT and a function table under Win9x. All the requests (IRPs) issued to the transports are sent from the target driver's context.

Address Families, Protocols, Socket Types

Winsock supports protocols and their appropriate address families for which a WinSock provider exists. For WinSock 2, these include IP, IPX/SPX, NetBIOS, AppleTalk, ATM, and IR.

Since DriverSockets talks directly to the transport drivers, via the thin TDI layer, it can support all the registered protocols and their address families, even those not exposed via the WinSock interface. The current DriverSockets release has been tested with the IP protocols only.

Winsock and DriverSockets support connection-oriented and connectionless socket types. In DriverSockets, the connection-oriented socket types are presented by KStreamSocket and the connectionless socket types by KDatagramSocket base classes. WinSock also includes a raw socket (SOCK_RAW), that allows use of service protocols such ICMP from the application. DriverSockets provides the base class KRawSocket for such purposes. The KRawSocket-derived class behaves exactly like the KDatagramSocket-derived class. The difference is in the construction and receive indications, which indicate the entire IP TSDU rather than the user's portion as with the UDP.

In supporting the server-side programming, DriverSockets transcends the traditional listen/accept model by providing a higher-level connection-oriented session framework. This framework is presented by the KStreamServer<>/KStreamServerSession class bundle.

Programming Models

Winsock offers a variety of programming models. It allows both blocking and non-blocking socket modes involving client's request processing. To facilitate network event processing, Winsock offers BSD-standard select and 4 Win32-specific *WSAAsyncSelect*, *WSAEventSelect*, *Overlapped I/O* and *Completion Port* models. These models define how to detect and process the asynchronous events generated by the underlying networking stack. They are particularly powerful in implementing Winsock-based servers processing multiple sessions.

DriverSockets, on the other hand, offers a single *asynchronous* event-driven programming model. The client's request, such as **send()**, are always non-blocking, and no thread context is associated with any of the requests. The **send()** requests include an optional context parameter, so that a single send completion callback is able to handle multiple pending requests.

The network events, such as connect and receive indications, are indicated via asynchronous callbacks implemented as overridable virtual functions, such as **OnReceive()**, of the base classes, KStreamSocket and KDatagramSocket. The event handlers run at the DISPATCH_LEVEL and should be designed to be re-enterable (e.g., they can preempt each other in the SMP environment).

Unlike Winsock, *DriverSockets does not provide any buffering for incoming data*. Thus, implementing a buffering strategy is up to the client. For stream-oriented sockets, however, DriverSockets does provide a flow control mechanism in the inbound direction by combining **OnReceive()** and **recv()** processing. For instance, returning 0 from the **OnReceive()** handler causes the transport to buffer/flow control incoming data until an explicit **recv()** request is made.

DriverSockets offers a special class framework to facilitate the design of kernel-mode, connection-oriented servers. The server framework is designed around two *bundled* base classes, KStreamServer and KStreamServerSession. KStreamServer deals with socket-related functionality, such as establishing and dropping TCP/IP connections, while KStreamServerSession deals with specific transactions over the established connection. The developer derives a specific session class from KStreamServerSession and instantiates a listener socket *KStreamServer<MySession>* to manage the sessions. The framework acts as a factory for the KStreamServerSession-derived objects.

Name Resolution

Name Resolution services translate a readable text name (such as "www.numega.com") into transport address (such as {12.6.233.20, 80 }.). Winsock supports network name resolution by expoiting DNS for TCP/IP networks and other directory services for other types of network. Winsock exposes this service via miscellaneous functions like **gethostbyname()**.

DriverSockets deals with raw transport addresses only. For IP networks, this means one has to provide a binary IP address (ULONG for IPv4 networks) and binary port number (USHORT). No name resolution is supported. Constructors for KStreamSocket, KDatagramSocket and KStreamServer require a pointer to a valid TRANSPORT_ADDRESS, or NULL for client-side sockets. DriverSockets provides a wrapper class, CIPTRANSPORT_ADDRESS, which facilitates the job of building a simple IP transport addresses.

Socket Options

Winsock allows the user to control socket behavior and attributes by using the **getsockopt()**/**setsockopt()** APIs. DriverSockets' internal class, KTdiClient, which is the base for all socket classes, provides public methods, **GetOption()** and **SetOption()**. These allow the user to control the behavior of the transport drivers. Unfortunately, while Windows NT DDK provides limited details on the options' definitions and appropriate data structures, some of them might have to be deduced from `winsock2.h` and `smpletcp.h`.

Miscellaneous Helper Functions

Winsock provides several useful *utility* functions which facilitate various functions including: data format translations, transmission of files, and duplication of sockets.

DriverSockets currently provides a limited subset of Winsock-style utility functions, including: **inet_addr()**, **inet_ntoa()**, **htonl()**, **ntohl()**, **htons()**, and **ntohs()**.

Multicasting

Winsock allows a socket to join and/or leave a multicast group by passing the IP_ADD/DEL_MEMBERSHIP option via the **setsockopt()** method. After the membership is established, a normal **sendto()**/**recvfrom()** method is used to multicast within the group.

DriverSockets exploits the same paradigm. A KDatagramSocket-derived class controls membership in a group by using methods **JoinMGroup()** and **LeaveMGroup()** which are based on the **KTdiClient::SetOption()** function. A multicast datagram is sent by the **KDatagramSocket::sendto()** function and received by the **OnReceive()** event handler.

QoS

Winsock 2 allows **WSAXxxx()** functions, such as **WSAConnect()**, to specify a *QoS* descriptor that suggests the desirable quality of service for a given request, connection, etc. Winsock translates the QoS requirements into RSVP-protocol messages, which in turn trigger and control QoS-related features of the underlying network components, such as the *802.1p* feature of the Ethernet NIC. Winsock also generates QoS-related events that the application can process. Although this functionality is currently not supported by DriverSockets, it is subject to further improvement.

Plug-and-Play

In this context, PnP is a capability which automatically discovers dynamic changes in the underlying networking stack, such as appearance and removal of transport providers.

Winsock 2 provides a pair of functions, **WSAProviderConfigChange()** and **WSAEnumProtocols()**. These functions support PnP.

DriverSockets provides the KTdiPnpClient class to support PnP on Windows 2000 and Windows XP systems.

Appendix A
Other Resources

Microsoft DDK

Short of the Windows source code, the Microsoft DDK is the definitive resource for information relating to device driver development. While many simple drivers can be written without a full and deep understanding of the underlying system architecture, the DDK documentation provides invaluable insight into the technical details you will need to know to build complex drivers. The DriverWorks documentation is designed to enhance rather than replace the DDK reference.

On-line Resources

Visit the Compuware web site for access to up-to-date technical information about driver development. The web site also includes pointers to other on-line resources for device driver developers, including newsgroups, ftp sites, and mailing lists.

The Compuware web site can be found at http://www.Compuware.com.

Index

Symbols

.cpp file 170

Numerics

802.3 medium 172

A

Accessor methods 38
Adapter 141
    base classes 141
    objects
        DMA 54
        embedded callback queue 54
AddDevice 125
AddMpToReg() method 177
Address 179, 183
    families 179, 183
        protocols socket types 183
    objects, peripheral 51
    resolution protocol 171
    space 130
APC_LEVEL 48
Assignment object 44
    class KResourceAssignment 44
Asynchronous
    and isochronous transfers 130
    model 184
    request 131
    support 132
    transfers 130
AVStream class 132

B

Base class 142
    implementation 177
BASEDIR 18
Binding
    instance 173
    notification 177
Build
    commands 21, 26
    environment shortcuts 17
    options 17
    scripts 24, 27
BUILD utility 20
Building
    a stream minidriver 139
    drivers 17, 21
    libraries 22, 25
Bundled base classes 185
Bus
    addresses 49, 51
        CPU 50
        peripheral 49
    and interrupts 52
    CPU bus 49
    drivers 68, 70, 72
        Universal Serial Bus (USB) 70
    mapping 50
    master device 55
    masters 53, 54
    multiple 49
    resets 131

C

Callback 35, 53
  query objects 44
  queue embedded in adapter object 54
  timed 60
CCheckTcpEcho 181
CCheckUdpEcho 182
CDeviceInterfaceClass 112
Characteristics.h 144, 148, 152, 159, 161
Class
  AVStream 132
  CDeviceInterfaceClass 112
  declarations 176
  drivers 69, 70, 72
    stream class driver 69
  KCommonDmaBuffer 55
  KConfigurationQuery 43
  KController 47
  KDeferredCall 52
  KDevice 39
  KDeviceQueue 45
  KDispatcherObject 57
  KDmaAdapter 54
  KDmaTransfer 55
  KDriver 34, 72
  KDriverManagedQueue 46
  KErrorLogEntry 62
  KEvent 58
  KFifo 61
  KFile 62
  KHeap 63
  KHeapClient 63
  KHidDevice 124, 125
  KHidMiniDriver 124
  KImageSection 36
  KInRange 51
  KInRegister 51
  KInterlockedList 61
  KInterrupt 52
  KInterruptSafeFifo 61
  KInterruptSafeList 61
  KIrb 132
  KIrp 37
  KList 61
  KLockableFifo 61
  KLowerDevice 41, 73
  KMemory 56
  KMemoryRange 51
  KMemoryRegister 51
  KMutex 59
  KPagedHeap 63
  KPagedHeapClient 63
  KPeripheralAddress 51
  KPnpDevice 39, 69, 72, 74, 75, 77, 79
  KPnpLowerDevice 41, 73, 74, 107
  KRegisrtyKey 43
  KResourceAssignment 44
  KSemaphore 58
  KSpinLock 57
  KStream 134, 138
  KStreamAdapter 134
  KStreamMinidriver 134
  KSystemThread 60
  KTimedCallback 60
  KTimer 60
  KUsbInterface 128
  KUsbLowerDevice 73, 127
  KUsbPipe 129
  KUstring 62
  KVxDInterface 74, 109
Closing streams adapter shutdown 138
CmCloseCall 153
CmMakeCall 153
CNotifyObject 177
CO_ADDRESS_FAMILY 152
CoActivateVc 149, 153
CoDeactivateVc 149, 153
Common buffer objects 55
  class KCommonDmaBuffer 55
Container classes 61
Controller object 47
  class KController 47
  controller_objects 47
Controlling hardware
  see objects, hardware control 49
CoRequest 148
CoSendPackets 149, 153
CPropertyUi template class 177
CreateFile 39
CreateInstance 149, 153
Creating
  the adapter 137
  the filter driver 136

D

DDK 187
  compiler support 19
  root directory 18
Deferred Procedure Call (DPC) 52
  Callback function 52
DeleteInstance 149, 153
Dequeueing an IRP 47
Destination address 172
Destructors 36

Development environment 2
Device
Data Block (DDB) 109
destructors 36
driver, new project 31
interfaces 74, 111
class 111
messages 8
objects 34, 36
anonymous devices 39
class
KDevice 39
KPnpDevice 39
CreateFile 39
deleting 76
device_object 39
ReadFile 39
symbolic links 39
unloading 36
WriteFile 39
Direct Memory Access
see DMA 53
DIRQL 48, 52
DISPATCH_LEVEL 48, 49, 52, 53
and spin locks 57
Dispatcher objects 57
class KDispatcherObject 57
event 58
semaphores 58
timed callbacks 60
Dispatching requests 35
DMA
adapter objects 54
class KDmaAdapter 54
bus masters 53
channel resources 34
objects
adapter, dma_adapter 54
dma_objects 53
dma_transfer 55
scatter/gather 54
slaved devices 53
transfer objects 57
class KDmaTransfer 55
dma_transfer 55
types 55
transfers 53, 55
Driver
code generation wizards 5
Development Kit (DDK) 15
framework 143, 154
installation 43
managed queues 46
object
callback 35
class KDriver 34
image section objects 36
unloading 36
parameters 43
subclass 34, 35
writer 34, 50
DriverEntry 35, 148
initialization entry point 34
path of registry key 43
DriverMonitor 7
DriverNetworks
description 4
libraries 25
types of drivers 14
Drivers
building 17
examples 6
DriverSockets 178, 182
introduction 178
programming model 180
vs WinSock 182
DriverWizard ix, 29
understanding 30
DriverWorks
build environment 15
class KPnpDevice 69
description 3
libraries 22
object model x, 33
types of drivers 13

E

Error log entries 62
Event 58
dispatcher object 58
log entry 62
error log entries 62
objects, class KErrorLogEntry 62
object 58
class KEvent 58
Examples KTDI 180
EzDriverInstaller 7

F

Fast mutex 59
FDO 69, 72, 107
FIFO objects 61
    class
        KFifo 61
        KInterruptSafeFifo 61
        KLockableFifo 61
    methods 61
    templates 61
File system redirector 165
Files, objects 62
    class KFile 62
Filter drivers 35
Firewall 155
FireWire 130
Flow control 184
Free builds 142
Functions, callback 47, 53

G

Generate a new driver 16

H

Halt 144, 146, 148, 150, 153
Handler 141
    methods 142
    naming conventions 143, 157
Hardware
    Abstraction Layer (HAL) 40
    low-level access 40
    objects 40
Heaps 63
    alternate heaps 63
    lookaside lists 63
    objects, class
        KHeap 63
        KHeapClient 63
        KPagedHeap 63
        KPagedHeapClient 63
    template classes 63
Helper 185
Higher-layer NDIS requests 159
Human Interface Device (HID) 75, 123
    class
        KHidDevice 125
        KHidMiniDriver 124
    minidrivers 124

I

I/O
    device interrupts, see DIRQL 48
    manager 34, 35, 37, 39, 45, 57, 84
        dispatching requests 35
    port objects 40
    registers 51
    Request Packet (IRP) 37, 40, 41, 45, 57
        and WDMs 68
        canceling 39
        class KIrp 37
        dequeuing 47
        io_request_obj 37
        Stack locations 38
    Requests (IRP)
        maintaining a count 76, 77
ICMP 183
IEEE 1394
    bus 130
    Bus Requests (IRBs) 131
    drivers 130
IM drivers 161
Image section objects 36
Initialization
    driver object 34
    entry point 34
    method 43
Initialize 146, 150, 152
Installed system filter example 154
Integrated Development Environment (IDE) 15
Interface 111, 112
    class 111
    layering 183
Intermediate drivers 154
Interrupt Service Routine (ISR) 52
    and DIRQL 52
    and IRQL 52
Interrupts 34, 40, 49, 52
    inter_obj 52
    objects
        class
            KDeferredCall 52
            KInterrupt 52
Introduction 141, 163
    protocol driver 163
    to Frameworks 141
IP TSDU 183
IPv4 addressing scheme 179
IRP 173, 183

IRQL 48, 52, 57, 60, 62
APC_LEVEL 48
DISPATCH_LEVEL 48
PASSIVE_LEVEL 48
Isochronous
support 131
transactions 130

K

K1394AddressFifo 132
K1394AddressRange 132
K1394AsyncClient 132
K1394IsochBandwidth 131
K1394IsochBufferList 132
K1394IsochChannel 131
K1394IsochResource 131
K1394IsochTransfer 131
K1394LowerDevice
SubmitIrb 131, 132
KCommonDmaBuffer class 55
KConfigurationQuery class 43
KController class 47
KDatagramSocket 179
KDeferredCall class 52
KDevice class 39
KDevice-derived class 171
KDeviceQueue class 45
KDispatcherObject 57
KDmaAdapter class 54
KDmaTransfer class 55
KDriver class 34, 72
KDriver-derived class 171
KDriverManagedQueue class 46
Kernel
mode drivers 49
and dispatcher objects 58
and file objects 62
and IRPs 37
and mapping 54
and spin locks 57
and timer objects 60
mode socket classes 142
streaming classes 132
KErrorLogEntry 62
KEvent 58
KFifo 61
KFile 62
KHeap 63
KHeapClient 63
KHidDevice class 124, 125
KHidMiniDriver class 124
KImageSection class 36
KInRange class 51
KInRegister class 51
KInterlockedList 61
KInterrupt class 52
KInterruptSafeFifo 61
KInterruptSafeList 61
KIrp class 37
KList 61
KLockableFifo 61
KLowerDevice class 41, 73
KMemory class 56
KMemoryRange class 51
KMemoryRegister class 51
KMutex 59
KMutex class 59
KMutexFast 59
KNdis 143, 154
intermediate drivers 154
miniport drivers 143
KNDIS_COMINIPORT_CHARACTERISTICS 148, 152
KNDIS_FILTERDRIVER_CHARACTERISTICS 159
KNDIS_MINIPORT_CHARACTERISTICS 144
KNdisAdapterTraits 144, 148, 152, 159
KNdisCallManagerVc 153
derived class 151
KNdisCallManagerVcTraits 153
KNdisCallManagerWrapper 153
KNdisCoWrapper 148, 152
KNdisFilterAdapter 141, 142
KNdisiMiniDriver 158
KNdisMiniAdapter 142, 143, 146, 148, 150, 152, 157, 158
derived
adapter class 147
class 151
KNdisMiniDriver 143
derived
class 151
driver class 147
KNdisMiniportVc 149, 153
derived
class 151
virtual connection class 147
KNdisMiniportVcTraits 149
KNdisProtocolBinding 158, 159
derived class 171
KNdisProtocolTraits 159
KNdisProtocolWrapper 159
KNdisWrapper 141, 144, 159
KPagedHeap 63
KPagedHeapClient 63
KPeripheralAddress class 51

KPnpDevice
class 39, 69, 72, 74, 75, 77, 79
member funtions 75
Pnp policies 79
OnBusReset 131
KPnpLowerDevice class 41, 73, 74, 107
overview 107
KRegistryKey class 43
KResourceAssignment class 44
KResourceRequest class 44
KSDEVICE 133
KsDevice 133
KSemaphore 58
KSFILTER 133
KsFilter 133
KSPIN 133
KsPin 133
KSpinLock 57
KStream class 134, 138
KStreamAdapter class 134
KStreamMinidriver class 134
KStreamServerSocket 179
KStreamSocket 179
KSystemThread 60
KTA_ADDRESS template 171
KTdiClient 142
KTdiPnpClient 186
KTimedCallback 60
KTimer 60
KTransport.h 170
KTransportAddress 171
KTransportConnection 171
KUsbInterface class 128
KUsbLowerDevice class 73, 127
KUsbPipe class 129
KUstring 62
KVxDInterface class 74, 109
overview 109

L

Lists objects 61
class
KInterlockedList 61
KInterruptSafeList 61
KList 61
methods 61
synchronization 61
templates 61
Load balancing 155
Lock, spin lock method 57
Lookaside lists 63
Lower device object
class KLowerDevice 41
KPnpLowerDevice 41
Low-level
hardware access 40
system object directories 62

M

m_Event 181
MAC addresses 171
Managing devices 34
Mapping 54
addresses 50
Memory objects 56
class KMemory 56
Microsoft DDK ix, 187
Minidriver 69, 107
Miniport
call manager 149
driver framework 143
drivers, connection-oriented 146
MiniportCoCreateVc 149
MiniportHalt 146, 150
MiniportInitialize 146, 148, 150, 152
MiniportXxxx 146, 150
Misc helper functions 185
Models 179, 184
programming 184
Modes of transfer 130
Monolithic drivers 37
Multicasting 183, 186
Mutexes 58, 59
class KMutex 59
dispatcher object 58
MUXs 157

N

Name 185
resolution 183, 185
NDIS 157, 160
(virtual) miniport 182
5.0 protocol drivers 163
binding object 159, 161
connection-oriented miniport drivers 146
filter driver 154, 155, 160
handler signatures 162
intermediate 157
drivers 141, 154
understanding the framework 154

miniport
driver 141
handler signatures 145
miniport call manager 149
mux driver 154, 156
(1 to N) 156
OID request 168
packet driver framework 167
protocol and transport drivers 141
NDIS_CALL_MANAGER_CHARCTERISTIC 152
NDIS_STATUS_FAILURE 169
NdisMCmRegisterAddressFamily 152
NetBEUI 179
Network
Driver Wizard 143
indication 179
interface adapters 163
NmPacket 164
Notify object 156, 177
framework 174
understanding 174

O

Object model x, 33
Objects 63
assignment, class KResourceAssignment 44
controller 47
deferred procedure call 52
DMA 40
driver managed queue 46
event 58
log entry 62
FIFO 61
file 62
hardware control
DMA 53
peripheral address 49
interrupt 40
inter_obj 52
lists 61
memory 56
mutexes 59
peripheral address 49
query 43
resource request 44
class KResourceRequest 44
semaphore 58
string 62
system thread 60
timed callback 60
timer 60
used during initialization
assignment objects 44
registry objects 43
resource request 44
OID maps 144
OnHalt() handler 162
On-line
help reference and how-to help ix
resources 187
OnQueryCapabilities 125
OnQueryId 125
OnReceive() handler 161
OnRemoveDevice 125
OnSend() handler 161
Open handles, maintaining a count of 77
Opening a stream 137
Overridable event handler 179

P

Packet
descriptor 161
drivers 164, 167
monitoring 155
scheduling 155
PASSIVE_LEVEL 48, 49, 182
and files 62
and system threads 60
PCI 70, 72
PDO 68, 69, 70, 74, 82, 84, 87, 107
and bus drivers 70
forwarding requests to 76
Peripheral address objects
class
KInRange 51
KMemoryRange 51
KPeripheralAddress 51
I/O addresses 50
KInRegister 51
KMemoryRegister 51
mapping addresses 50
periph_address 49
physical addresses 49
target devices 50
Physical Device Object (PDO)
see PDO 68, 72
Physical MAC address 168
Plug and play 75, 168, 183, 186
Pnp 76, 77, 78, 80, 186
policies 79
Policy 81, 83, 84, 85, 87, 88, 89, 90, 92
Priority levels, scheduler process 48
Product description 2

Programming models 183, 184
Protocol driver 163
    introduction 163
ProtocolCoCreateVc 153
ProtocolCoDeleteVc 153
Protocols 179

Q

QoS 183, 186
Query objects 43
    class KConfigurationQuery 43
Queues
    callback 54
    class KDeviceQueue 45
    controller objects 47
    DPC 53
    driver managed 46
    I/O request 37, 59
    objects 45

R

ReadFile 39
Real
    adapter 177
    miniports 161
    NIC 161
    upper-layer protocols 161
Registry 43
    class KRegistryKey 43
    keys 43, 62
Reinitialization 35
Relationships and flow of control 135
Report descriptor 124
Request
    completion 179
    serialization 45
Reset 146, 150
Resource
    assignment object 44
    request 44
        objects 44
Retrieve attributes 38

S

Scatter/gather 54, 55
Sections
    paged 36
    unpaged 36
Semaphores 58
    class KSemaphore 58
    dispatcher object 58
Serializing I/O requests 45
Service Request Block (SRB) 136
Shared resources 47
Shortcuts, build environment 17
Signature 141
Slaved device 53, 54, 55
SMP environment 184
smpletcp.h 185
SOCK_RAW 183
Socket options 185
Spin locks 57
    and FIFO objects 61
    and list objects 61
    objects
        class KSpinLock 57
        lock 57
        unlock 57
Static library 183
Stream
    class driver 69
    control and data 137
    minidrivers 69, 73, 75
Streams 136, 137, 138
    classes for 132
Strings
    creating event log entries 62
    objects 62
        class KUstring 62
Structure for managing devices 34
Summary of initialization logic 138
Symbolic links 39, 62
Synchronization 53
    dispatcher objects 57
    semaphores 58
    spin locks 57
    using controller object 47
System
    DMA channel 54
    thread object 60
    threads 58, 60
        dispatcher object 58
szProvider 179

T

Target devices, peripheral address objects 50
Target OS 19
TCP 179
TCP/IP 142
tcpip.sys 165

TDI 141
client 141, 165, 173
connection endpoint object 171
IRPs 171
plug and play system 165
specification 165
TA_ADDRESS structure 171
transports 165
Templates 61
FIFO 61
list 61
Thin TDI layer 183
Threads 60
Timed callback object 60
Timer object 60
Timers 58, 60
dispatcher object 58
Transfer objects 55, 132
termination methods 56
Transport
(layer 4) protocol 166
address object 173
Driver Interface 142
drivers 165, 169
object pointer 173
Type 183

U

UDP 179
Understanding 143, 154, 174
notify object framework 174
Unicode
characters 62
strings 62
Unitized names 63
Universal Serial Bus (USB) 70, 72, 73, 75, 126
Unloading a driver 36
Unlock
dynamically, image section 36
spin lock method 57
Utility functions 185

V

Virtual
adapter 177
instance name 177
addresses 55, 56
memory 54, 55, 56
Visual Studio
IDE, setting up 18
integration 15
VxD 109

W

WDM 67, 68, 69, 70, 72, 74, 108
and lower device objects 41
and query objects 43
bus drivers 70
class drivers 69
classes 73
DriverEntry 34
DriverObject 34
environment 68
filter device object 134
kernel stream subsystem 135, 136
layered architecture 69
minidrivers 69
resource management 108
stream
class driver 69
minidriver 69
streaming
architecture 135
DirectShow filter 135
filter driver 135
kernel streaming classes 132
user-mode stream proxy 135
drivers 132
streaming architecture 135
Universal Serial Bus (USB) 70
WdmSniffer 7
Welcome 1
WHQL certifiable 12
Windows
2000 67
98 67
driver model 67
Me 67
XP 67
WinSock 166, 182, 183
winsock2.h 185
Wizards, code generation 5
WOSA component 183
Wrapper 141
template classes 142
WriteFile 39
WSAEnumProtocols() 186
WSAProviderConfigChange() 186