

[Table of Contents](#)
[About the Authors](#)

[The Windows 2000 Device Driver Book, A Guide for Programmers, Second Edition](#)

Art Baker Jerry Lozano Publisher: Prentice Hall PTR Second Edition November 20, 2000
ISBN: 0-13-020431-5, 480 pages

-
- The #1 Windows device driver book - fully updated for Windows 2000!
-
- Step-by-step planning, implementation, testing, debugging, installation, and distribution
-
- Complete coverage of the new Windows Driver Model (WDM)
-
- Practical debugging and interactive troubleshooting
-
- CD-ROM: Exclusive tools for streamlining driver development, plus extensive C/C++ sample driver library!
-
- Windows Driver Model (WDM) for Windows 2000 and 98 - in depth!
-
- Building drivers that support Plug-and-Play and Power Management
-
- Windows Management Instrumentation: logging device errors and events - and interpreting them
-
- Constructing safe reentrant driver code
-
- Handling time-out conditions safely and effectively
-
- Advanced techniques: kernel-mode threads, layered drivers, and more
-
- Start-to-finish debugging and troubleshooting techniques

Foreword by Andrew Scoppa, UCI Corporation

The #1 book on Windows driver development - totally updated for Windows 2000!

With *The Windows 2000 Device Driver Book*, any experienced Windows programmer can master driver development start to finish: planning, implementation, testing, debugging, installation, and distribution. Totally updated to reflect Microsoft's Windows Driver Model (WDM) for Windows 2000 and 98, this programming bestseller covers everything from architecture to tools, and includes a powerhouse library of exclusive tools and code for streamlining any driver development project.

You'll start with a high-level overview of WDM components and then move quickly into the details of the development environment and driver installation. Next, master the Windows 2000 I/O Manager, its data structures, and its interaction with drivers. Using extensive practical examples, you'll implement Plug-and-Play and Power Management; construct safe reentrant driver code; use Windows Management Instrumentation to log errors and events, and more.

The book covers today's most advanced Windows driver development techniques and

The Windows 2000 Device Driver Book, A Guide for Programmers, Second Edition

Foreword

Preface

- What You Should Already Know
- What's Covered
- What's Not
- About the Sample Code
- History of this Book
- Training and Consulting Services

Acknowledgments

1. Introduction to Windows 2000 Drivers

- Overall System Architecture
- Kernel-Mode I/O Components
- Special Driver Architectures
- Summary

2. The Hardware Environment

- Hardware Basics
- Buses and Windows 2000
- Hints for Working with Hardware
- Summary

3. Kernel-Mode I/O Processing

- How Kernel-Mode Code Executes
- Use of Interrupt Priorities by Windows 2000
- Deferred Procedure Calls (DPCs)
- Access to User Buffers
- Structure of a Kernel-Mode Driver
- I/O Processing Sequence
- Summary

4. Drivers and Kernel-Mode Objects

- Data Objects and Windows 2000
- I/O Request Packets (IRPs)
- Driver Objects
- Device Objects and Device Extensions
- Controller Objects and Controller Extensions
- Adapter Objects
- Interrupt Objects
- Summary

5. General Development Issues

- Driver Design Strategies
- Coding Conventions and Techniques
- Driver Memory Allocation
- Unicode Strings
- Interrupt Synchronization
- Synchronizing Multiple CPUs
- Linked Lists
- Summary

6. Initialization and Cleanup Routines

- Writing a DriverEntry Routine
- Code Example: Driver Initialization
- Writing Reinitialize Routines
- Writing an Unload Routine
- Code Example: Driver Unload
- Writing Shutdown Routines
- Testing the Driver
- Summary

7. Driver Dispatch Routines

- [Announcing Driver Dispatch Routines](#)
- [Writing Driver Dispatch Routines](#)
- [Processing Read and Write Requests](#)
- [Code Example: A Loopback Device](#)
- [Extending the Dispatch Interface](#)
- [Testing Driver Dispatch Routines](#)
- [Summary](#)

[8. Interrupt-Driven I/O](#)

- [How Programmed I/O Works](#)
- [Driver Initialization and Cleanup](#)
- [Writing a Start I/O Routine](#)
- [Writing an Interrupt Service Routine \(ISR\)](#)
- [Writing a DpcForIsr Routine](#)
- [Some Hardware: The Parallel Port](#)
- [Code Example: Parallel Port Loopback Driver](#)
- [Testing the Parallel Port Loopback Driver](#)
- [Summary](#)

[9. Hardware Initialization](#)

- [The Plug and Play Architecture: A Brief History](#)
- [The Role of the Registry for Legacy Drivers](#)
- [Detecting Devices with Plug and Play](#)
- [The Role of Driver Layers in Plug and Play](#)
- [The New WDM IRP Dispatch Functions](#)
- [Device Enumeration](#)
- [Device Interfaces](#)
- [Code Example: A Simple Plug and Play Driver](#)
- [Summary](#)

[10. Power Management](#)

- [Hot Plug Devices](#)
- [OnNow Initiative](#)
- [Wake Requests](#)
- [Power Management Issues](#)
- [Summary](#)

[11. Timers](#)

- [Handling Device Timeouts](#)
- [Code Example: Catching Device Timeouts](#)
- [Managing Devices without Interrupts](#)
- [Code Example: A Timer-Based Driver](#)
- [Summary](#)

[12. DMA Drivers](#)

- [How DMA Works under Windows 2000](#)
- [Working with Adapter Objects](#)
- [Writing a Packet-Based Slave DMA Driver](#)
- [Code Example: A Packet-Based Slave DMA Driver](#)
- [Writing a Packet-Based Bus Master DMA Driver](#)
- [Writing a Common Buffer Slave DMA Driver](#)
- [Writing a Common Buffer Bus Master DMA Driver](#)
- [Summary](#)

[13. Windows Management and Instrumentation](#)

- [WMI: The Industry Picture](#)
- [The WMI Architecture](#)
- [WMI Summary](#)
- [Conventional Driver Event Logging](#)
- [Summary](#)

[14. System Threads](#)

- [Definition and Use of System Threads](#)
- [Thread Synchronization](#)
- [Using Dispatcher Objects](#)
- [Code Example: A Thread-Based Driver](#)
- [Summary](#)

[15. Layered Drivers](#)

- [An Overview of Intermediate Drivers](#)
- [Writing Layered Drivers](#)
- [Writing I/O Completion Routines](#)
- [Allocating Additional IRPs](#)
- [Writing Filter Drivers](#)
- [Code Example: A Filter Driver](#)
- [Writing Tightly Coupled Drivers](#)
- [Summary](#)

- [16. Driver Installation](#)
 - [Installation of a Driver](#)
 - [Auto-Install Using INF Files](#)
 - [Using a Driver INF File](#)
 - [Controlling Driver Load Sequence](#)
 - [Digital Signing of a Driver](#)
 - [Summary](#)

- [17. Testing and Debugging Drivers](#)
 - [Guidelines for Driver Testing](#)
 - [Why Drivers Fail](#)
 - [Reading Crash Screens](#)
 - [An Overview of WinDbg](#)
 - [Analyzing a Crash Dump](#)
 - [Interactive Debugging](#)
 - [Writing WinDbg Extensions](#)
 - [Code Example: A WinDbg Extension](#)
 - [Miscellaneous Debugging Techniques](#)
 - [Summary](#)

- [A. The Driver Debug Environment](#)
 - [Hardware and Software Requirements](#)
 - [Debug Symbol Files](#)
 - [Enabling Crash Dumps on the Target System](#)
 - [Enabling the Target System's Debug Client](#)

- [B. Bugcheck Codes](#)

- [C. Building Drivers](#)
 - [The Build Utility](#)
 - [Using Visual Studio to Build Drivers](#)

- [Bibliography](#)

Foreword

Drivers are the most fundamental and technically difficult part of operating system development. As a reader of this book, you are probably well aware of the complexities involved. Even for the most seasoned software engineer the task can be daunting. Writing device drivers under Windows 2000 is a big challenge to learn. The most comprehensive, authoritative guide to Windows NT driver development, *The Windows NT Device Driver Book* by Art Baker is now a classic. I can not think of anyone better qualified to write the second edition of Art's outstanding book than Jerry Lozano. Jerry combines the qualities of strong technologist, excellent writer, and gifted educator. These qualities have translated into book form very well. Reading this book I felt I was taking one of Jerry's classes.

There are two kinds of books. Some books provide reference information that very much read like an encyclopedia. Such books are picked up occasionally to answer a specific question. Other books are tutorial in nature. They are designed to be read from front to back in order to transfer the knowledge and skill necessary to perform a task.

The Windows 2000 Device Driver Book, like its predecessor, falls clearly into the latter category. It is intended to be used as an instructional guide for device driver authors. Unlike other books on the subject, this book does not attempt to reproduce the DDK. The DDK stands as the definitive reference on the Windows 2000 device driver technology. Instead, *The Windows 2000 Device Driver Book* provides the guiding information needed to successfully master W2K driver development. This book gives developers the knowledge to design, write, and debug Windows 2000 devices, and is based on a course Jerry created and teaches for UCI. Based on feedback from the course, Jerry found that one of the biggest problems device driver and kernel-mode code developers face is the lack of clear, concise technical information on driver models, kernel mode programming, and hardware interfaces. In this book Jerry has succeeded in solving this problem with detailed examples and informative coverage in all areas, and presenting it with exceptional clarity.

As the book went to press, it was clear that another chapter was highly desirable. The chapter concerns USB and IEEE 1394 driver specifics. The revision author has generously agreed to include this chapter on the book's companion web site: <http://www.W2KDriverBook.com>. Readers that need this information should visit this informative site.

Andrew Scoppa

President

UCI Software Technical Training

Preface

This book explains how to write, install, and debug device drivers for Windows 2000. It is intended to be a companion to the Microsoft DDK documentation and software.

Windows 2000 represents a major improvement to previous versions of Windows NT. Device drivers for Windows 2000 may be designed for the new Windows Driver Model (WDM) architecture. If so, the driver will be *source* compatible with Windows 98. This book covers the new WDM specification.

This book will also prove useful to those studying the internals of Windows 2000, particularly the I/O subsystem and related components.

What You Should Already Know

All instruction assumes a base knowledge level. First, the reader should be familiar with Windows 2000 administration—security and setup, for example. Since experimentation with kernel-mode code can (and will) cause system problems, the reader should be prepared and able to restore a chaotic OS.

Second, the reader should be competent in the C programming language and somewhat familiar with C++. Only a little C++ is used in this book, and then only for the purpose of simplifying tedious code.

Third, experience with Win32 user-mode programming is useful. Knowing how user-mode code *drives* I/O devices is useful in designing and testing device driver code. The test code for the examples in this book rely on the console subsystem model for Windows. To review this topic, the reader is referred to the *Win32 Programmers Reference*, particularly the chapters on I/O primitives (CreateFile, ReadFile, WriteFile, and DeviceIoControl). The bibliography lists other references for this topic.

Finally, while no specific prior knowledge of hardware or device driver software design is assumed, it would be useful if the reader had experience with some aspect of low-level device interfacing. For example, knowledge of writing device drivers for a Unix system will prove quite useful when reading this book.

What's Covered

The focus of this book is to first explain the *architecture* of the hardware, environment, and device driver, and then to explain the *details* of writing code.

Chapters are grouped within this book as follows:

[Chapter 1](#), [Chapter 2](#), [Chapter 3](#), [Chapter 4](#), [Chapter 5](#):

The first five chapters of this book cover the foundation of what's needed to write a device driver. This includes coverage of the Windows 2000 architecture, hardware terminology and bus basics, and an in-depth view of the Windows 2000 I/O Manager and related services.

[Chapter 6](#), [Chapter 7](#), [Chapter 8](#), [Chapter 9](#), [Chapter 10](#), [Chapter 11](#), [Chapter 12](#), [Chapter 13](#):

The next eight chapters form the nucleus of this book. The chapters cover everything from the mechanics of building a driver to the specifics of instrumenting a driver to log errors and other events.

[Chapter 14](#), [Chapter 15](#):

These two chapters deal with somewhat more advanced topics within device driver construction. This includes the use of system threads, layering, filtering, and utilizing driver classes.

[Chapter 16](#), [Chapter 17](#):

The final chapters deal with the practical but necessary details of driver installation and debugging. The use of Windows 2000 INF files for "automatic" installation of a plug and play device driver is covered (as well as manual installation for legacy devices). The use of WinDbg is covered in sufficient detail so that the programmer can actually perform interactive debugging.

Appendices:

The appendices cover reference information needed for driver development. The mechanics of Windows 2000 symbol file installation, bugcheck codes, and so on are listed.

What's Not

Since the purpose of this book is to cover driver development from "the ground up," some specific topics fall outside its scope. Specifically, the list of topics not covered includes

File system drivers: Currently, the construction of a full Windows 2000 Installable File System requires the acquisition of the Microsoft IFS kit. The bibliography of this book points to one source for more information on this topic. Potential users of the IFS kit will benefit greatly from this book, as the material covered is essential prerequisite knowledge.

Device-specific driver information: The construction of NIC (Network Interface Card), SCSI, video (including capture devices), printers, and multimedia drivers is not specifically covered in this book. [Chapter 1](#) discusses the architectural implications of such drivers, but even individual chapters on each of these driver types would seriously shortchange the requisite knowledge.

Virtual DOS device drivers: The current wave of driver development is toward the WDM 32-bit model. Legacy 16-bit VDDs are no longer of interest.

About the Sample Code

Most chapters in this book include one or more sample drivers. All code is included on the accompanying CD. Samples for each chapter are in separate subdirectories on the CD, so installation of individual projects is straightforward.

The CD also includes a device driver application wizard for Microsoft Visual C++ version 6. This wizard configures the build environment so that code can be written, compiled, and linked within Visual Studio.

Platform dependencies:

The sample code included with this book has been targeted and tested on Intel platforms only. Since it appears that the last non-Intel platform (Alpha) was dropped from the final release of Windows 2000, this should come as no surprise. Be advised, however, that Windows 2000 is intrinsically a platform-independent OS. It is a straightforward process to port the OS to many modern hardware sets. Driver writers should consider designs that take advantage of the Windows 2000 abstractions that permit source compatibility with non-Intel platforms.

To build and run the examples:

Besides the Microsoft DDK (Device Driver Kit) (which is available on an MSDN subscription or, at present, free for download from the Microsoft web site at <http://www.microsoft.com/DDK>), the sample code assumes that Microsoft Visual C++ is installed. The device driver application wizard was built for Visual Studio version 6. Obviously, with some effort the sample code can be built using other vendors' compilers.

Of course, an installed version of Windows 2000 (Professional, Server, or Enterprise) is required. For interactive debugging using WinDbg, a second host platform is required.

History of this Book

The first version of this book was written by Art Baker, entitled *The Windows NT Device Driver Book*. By any account, the book was required reading for any NT driver author. The Microsoft driver model is a continuously moving target. As such, recently introduced books on this subject provided more and up-to-date information. The goal of this revision of the book is to carry forward the goals, style, and clarity of Art's original work while updating the material with the very latest information available from Microsoft.

If you are a previous reader of the original version of this book, I hope you will find this version just as useful. I have attempted to provide accurate, concise, and clear information on the subject of Windows 2000 device drivers. While I have relied heavily on Art's original work, any errors present in this book are entirely mine.

Training and Consulting Services

The material in this book is based on training and consulting performed for various companies within the industry.

The subject matter of this book is presented exclusively by UCI in the format of a five-day instructor-lead lecture/lab course. The course is available as public or on site classes. UCI provides comprehensive training in high-end programming, web development and administration, databases, and system technologies.

For more information please visit the UCI web site at <http://www.ucitraining.com> or use the address information below:

UCI Corporation

4 Constitution Way

Suite G

Woburn, MA 01801

1-800-884-1772

The revision author, Jerry Lozano, provides seminars and workshops on the topic of device drivers and other related subjects. For more information visit the web site: <http://www.StarJourney.com>

Acknowledgments

I am grateful to many people who helped me with this 2nd edition. First and foremost, I want to thank Art Baker for his original work. The structure and content of this revision is based on his initial efforts.

To my partner in life, Carol, who makes everything possible. I thank Carol for both her personal and professional support. Without your encouragement, I would never have started this project. Without your help, I would never have finished.

Thanks to Russ Hall, my development editor and friend, for making the book sound good.

Thanks to Patty Donovan and her staff at Pine Tree Composition for making the book look good.

The staff of Prentice Hall PTR, especially Mike Meehan and Anne Trowbridge, deserve considerable credit for their patience and encouragement in leading me through the entire process.

I wish to thank Bryce Leach of Texas Instruments who tried to correct my misunderstandings of IEEE 1394. Your comments and suggestions for [Chapter 2](#) were invaluable.

Thanks to Ron Reeves, for several great technical comments on several chapters.

And finally, thanks to the many people who attend my seminars, workshops, and classes for asking all those wonderful, thought-provoking questions.

Chapter 1. Introduction to Windows 2000 Drivers

CHAPTER OBJECTIVES

- [Overall System Architecture](#)
- [Kernel-Mode I/O Components](#)
- [Special Driver Architectures](#)
- [Summary](#)

Device drivers on any operating system necessarily interact intimately with the underlying system code. This is especially true for Windows 2000. Before jumping into the world of Windows 2000 device drivers, this chapter presents the design philosophy and overall architecture of Windows 2000.

Overall System Architecture

Windows 2000 presents arguably the most aggressive attempt at operating system control in the history of computers. This section tours the Windows 2000 architecture, highlighting the features of significant interest to a device driver author.

Design Goals for Windows 2000

The original goals for Microsoft's NT ("New Technology") operating system took form in early 1989. Interestingly, the original concept for NT did not even include a Windows operating environment. While the NT OS has indeed come a long way since 1989, the five fundamental goals remain intact.

-

Compatibility.

The OS should support as much existing software and hardware as possible.

-

Robustness and reliability.

The OS should be resilient to inadvertent or intentional misuse. A user's application should not be able to crash the system.

-

Portability.

The OS should run on as many present and future platforms as possible.

-

Extendibility.

Since the market requirements will change (grow) over time, the OS must make it easy to add new features and support new hardware with minimal impact on existing code.

-

Performance.

The OS should provide good performance for a given capability of the hardware platform which hosts it.

Of course, goals are not reality, and over time, serious compromise of one goal may be necessary to achieve another. NT is an operating system and, as such, is subject to the same sorts of compromises that affect all systems. The remainder of this section describes the delicate balance of solutions that Microsoft OS designers chose to implement their goals.

Hardware Privilege Levels in Windows 2000

To achieve the robustness and reliability goal, the designers of NT chose a *client-server architecture* for its core implementation. A user application runs as a client of OS services.

The user application runs in a special mode of the hardware known generically as *user mode*. Within this mode, code is restricted to nonharmful operations. For example, through the magic of virtual memory mapping, code cannot touch the memory of other applications (except by mutual agreement with another application). Hardware I/O instructions cannot be executed. Indeed, an entire class of CPU instructions (designated *privileged*), such as a CPU Halt, cannot be executed. Should the application require the use of any of these prohibited operations, it must make a request of the operating system kernel. A hardware-provided *trap* mechanism is used to make these requests.

Operating system code runs in a mode of the hardware known as *kernel mode*. Kernel-mode code can perform any valid CPU instruction, notably including I/O operations. Memory from any application is exposed to kernel-mode code, providing, of course, that the application memory has not been *paged out* to disk.

All modern processors implement some form of *privileged vs. nonprivileged* modes. Kernel-mode code executes in this privileged context, while user-mode code executes in the nonprivileged environment. Since different processors and platforms implement privileged modes differently, and to help achieve the goal of portability, the OS designers provided an abstraction for user and kernel modes. OS code always uses the abstraction to perform privileged context switches, and thus only the abstraction code itself need be ported to a new platform. On an Intel platform, user mode is implemented using Ring 3 of the instruction set, while kernel mode is implemented using Ring 0.

This discussion is relevant to device driver writers in that kernel-mode drivers execute in a privileged context. As such, poorly

Kernel-Mode I/O Components

The purpose of this section is to describe the goals and architecture of the Windows 2000 I/O subsystem. Since different kinds of drivers perform wildly different kinds of service, the I/O Manager's categorization of drivers is also discussed.

Design Goals for the I/O Subsystem

The I/O subsystem of Windows 2000 added to the overall design goals of the operating system by including

- Portability, platform to platform.
- Configurability in terms of both hardware and software. For Windows 2000 drivers, this would include full support for Plug and Play buses and devices.
- Preemptable and interruptable. I/O code should never block and should always be written thread-safe.
- Multiprocessor-safe. The same I/O code should run on both uniprocessor and multiprocessor configurations.
- Object-based. The services provided by I/O code should be offered in encapsulated data structures with well-defined allowable operations.
- Packet-driven. Requests made of the I/O subsystem should be submitted and tracked using a distinct "work order" format, known as an *I/O Request Packet* (IRP).
- Asynchronous I/O support. Requests made of the I/O subsystem should be allowed to complete in parallel with the requestor's execution. When the request ultimately completes, a mechanism must exist to notify the caller of completion.

Besides these published goals, there is also strong emphasis placed on code reusability. This translates to heavy structuring of I/O code (including drivers) into logical layers. For example, bus-driving code should be layered separately from specific device code to allow for reuse of the bus code across multiple devices. In many cases, different vendors supply code for different layers. Only through careful *modularization* can this goal be achieved.

Kinds of Drivers in Windows 2000

There once was a time when a device driver author could understand the intricacies of the new hardware, learn the OS device driver interface, scope the work, and "just write the code." For better or worse, the days of monolithic device driver code have passed. Today, an author must understand the *architectures* of both complex hardware buses and heavily layered I/O subsystems just to scope the work statement. Deciding what *kind* of driver to write for Windows 2000 is itself an interesting challenge. Deciding whether to implement or to reuse a layer is yet another challenge. The purpose of this section is to describe where different kinds of drivers fit within the hardware world and the OS.

At the highest level, Windows 2000 supports two kinds of drivers, user-mode and kernel-mode. User-mode drivers, as the name implies, is system-level code running in user mode. Examples include a simulated, or *virtualized*, driver for imaginary hardware or perhaps a new environmental subsystem. Since Windows 2000 user mode does not allow direct access to hardware, a virtualized driver necessarily relies upon real driver code running in kernel mode. This book does not describe user-mode drivers. The purpose of this book is to describe *real* drivers, which in Windows 2000 are known as *kernel-mode drivers*.

Kernel-mode drivers consist of system-level code running in kernel mode. Since kernel mode allows direct hardware access, such drivers are used to control hardware directly. Of course, nothing prevents a kernel-mode driver from virtualizing real hardware—the choice between user and kernel mode is largely an implementer's choice. Again, however, the purpose of this book is to present the strategies for implementing true kernel-mode drivers for real hardware.

Moving down a level, kernel-mode drivers can be further decomposed into two general categories, legacy and Windows Driver Model (WDM). Legacy drivers were fully described in the first edition of this book. The techniques needed to discover hardware and interface with the I/O subsystem are well documented. Thankfully, most of the knowledge gained by understanding legacy Windows NT drivers is transportable to the Windows 2000 (and Windows 98) WDM world.

WDM drivers are Plug and Play compliant. They support power management, autoconfiguration, and hot plugability. A correctly written WDM driver is usable on both Windows 2000 and Windows 98, though at present, Microsoft does not guarantee binary

Special Driver Architectures

Building upon the intermediate driver strategy described in the last section, Microsoft supplies driver architectures for several types or classes of devices.

- Video drivers
- Printer drivers
- Multimedia drivers
- Network drivers

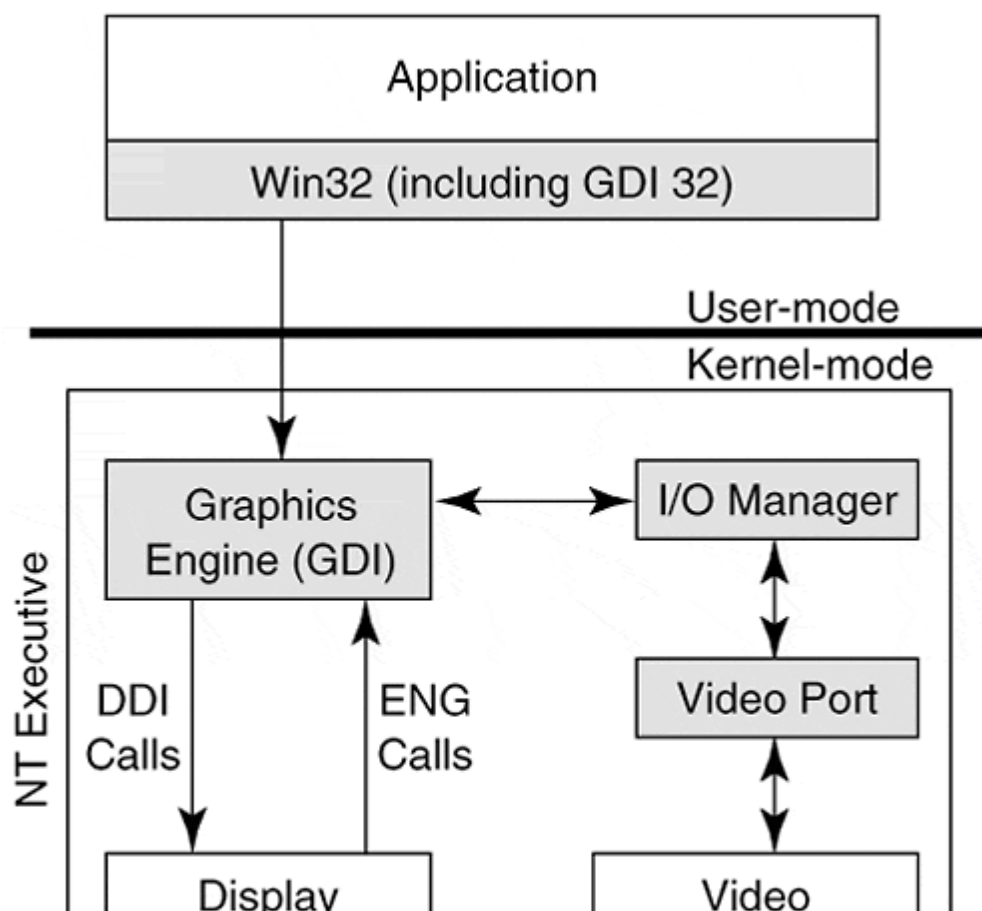
These architectures conform to the spirit, if not to the letter, of the classifications of the last section. Each architecture is described in more detail in the following sections.

Video Drivers

Video drivers in Windows 2000 present special requirements to the I/O subsystem. Because the graphical interface is constantly exposed to users, the apparent overall speed of the system is judged (often incorrectly) by the performance of this component. Competition among video adaptor hardware vendors has forced aggressive graphic accelerators to be included on high-end boards. The video driver architecture must exploit such hardware when it exists, yet provide full compatibility and capability when it does not. Finally, video drivers have evolved since the 16-bit world of Windows. There is a need to provide as much compatibility as possible with legacy drivers.

The video driver architecture of Windows 2000 is shown in [Figure 1.4](#). The shaded components are provided with Windows 2000. Vendors of specific display adaptors supply the display driver. Since many display adaptors are designed using common chip sets, the chip set manufacturer supplies the video miniport *class driver* for its adaptor-manufacturing customers. For example, an ET4000 Miniport driver exists for all adaptors that utilize the ET4000 chip set. The extra hardware surrounding the chip set is driven with adaptor-specific display driver code.

Figure 1.4. Video driver architecture.



Summary

Windows 2000 provides a rich architecture for applications to exploit. Of course, this richness comes at a price that device driver authors (among others) will have to pay. The I/O processing scheme of Windows 2000 is complex and the need to keep a view of the "big picture" is difficult, but necessary.

At this point, there should be a good understanding of what kinds of drivers exist (and where they fit) in Windows 2000. The next chapter begins a discussion of the hardware that drivers must control. An overview of the various buses that are supported by Windows 2000 is presented.

Chapter 2. The Hardware Environment

CHAPTER OBJECTIVES

-

- [Hardware Basic](#)

-

- [Buses and Windows 2000](#)

-

- [Hints for Working with Hardware](#)

-

- [Summary](#)

It is a fact of life that device drivers must interact with "real" hardware. Traditionally, first-generation device drivers were written as unit test code by the hardware designers. These early drivers were then passed to hardware-savvy software engineers to produce a fully-integrated driver.

As discussed in the previous chapter, the term *device driver* in Windows 2000 can describe a block of code miles away from any real hardware. Sooner or later, however, a device driver author must understand hardware and the terms sufficient to drive it. This chapter gives a gentle overview of hardware found on most systems.

Hardware Basics

Regardless of the type of device being driven, there are several basic items that must be known.

-
- How to use the device's control and status registers
-
- What causes the device to generate an interrupt
-
- How the device transfers data
-
- Whether the device uses any dedicated memory
-
- How the device announces its presence when attached
-
- How the device is configured, preferably through software

The following sections discuss each of these topics in a general way.

Device Registers

Drivers communicate with a peripheral by reading and writing registers associated with the device. Each device register generally performs one of the following functions:

-
- **Command.**
Bits in these registers *control* the device in some way—perhaps starting or aborting a data transfer or configuring the device—and are typically *written* by the driver.
-
- **Status.**
These registers are typically *read* by the driver to discover the current state of the device.
-
- **Data.**
These registers are used to transfer data between device and driver. Output data registers are *written* by the driver, while input data registers are *read* by the driver.

Simple devices (like the parallel port interface in [Table 2.1](#)) have only a few associated registers, while complex hardware (like graphics adapters) have many registers. The number and purpose of the registers is ultimately defined by the hardware designer and *should* be well documented in the Hardware Design Specification. Often, however, trial and error is required on the part of the driver author to determine the *real* behavior of bits in the various device registers. Further, experience shows that "reserved" bits in a register do not necessarily mean "don't care" bits. It is usually best to mask out these bits when reading, and to force them to zero when writing registers with reserved bits.

Accessing Device Registers

Once the hardware functions are known, there is still the small matter of knowing how to programmatically *reference* the device's registers. To do this, two additional pieces of information are required.

-
- The address of the device's first register
-
- The address *space* where the registers live

Usually, device registers occupy consecutive locations. Therefore, the address of the first register is a necessary clue in gaining

Buses and Windows 2000

A *bus* is a collection of data, address, and control signal lines that allow devices to communicate. Some buses are wide, allowing simultaneous transmission of many bits of data and control. Others are nothing more than a single wire, allowing devices to transmit data and control in a serial fashion. Some buses allow any device to communicate with any other device on the bus. Others require the presence of a master controller (e.g., a CPU or I/O controller) that is the sole recipient or sender of data.

Buses gain and lose popularity over time. Trade-offs between speed, cost, extensibility, and ease of use change as bus technology advances. Device requirements change as new technologies are introduced into the PC world. For example, the common use of digital video on home PCs has produced a need for a simple, ultra-high speed bus.

The device driver architecture of Windows 2000 supports new buses easily and efficiently. Many popular buses are supported "out of the box." The remainder of this section describes the currently supported buses on Windows 2000. The descriptions are meant to provide an overview of the bus technology. For more detailed information, please refer to the bibliography.

ISA: The Industry Standard Architecture

This is the bus that IBM specified for the PC/AT in the early 1980s. It supports 8-bit and 16-bit devices. Because its origins are two decades old, ISA is neither fast nor simple. The bus clock rate is 8.33 MHz. Since even 16-bit transfers take at least two clock cycles, the maximum transfer rate is only about 8 MB/second. Since current CPU speeds are two orders of magnitude faster, ISA is no longer a relevant bus. Its inclusion in Windows 2000 and this book are for backward compatibility reasons only.

REGISTER ACCESS

ISA is the bus by which the need for autorecognition and autoconfiguration were defined. That is, ISA provides the definition for bus chaos. No standards exist for register usage, so devices may grab any I/O address. Generally, I/O addresses between 0x0000 and 0x00FF belong only to devices on the mainboard. Territory between 0x0100 and 0x03FF is available for plug-in cards. Each card is typically assigned a 32-byte contiguous range of register addresses.

Sadly, many legacy ISA cards do not decode all 16 I/O address lines. Instead, they decode only the first 10 bits of address. Such cards respond to alias addresses in multiples of 0x400. A device at address 0x300 also responds at address 0x700. When such devices are present in a system, the 64 KB I/O address range diminishes quickly.

INTERRUPT MECHANISMS

Interrupts on an ISA bus are traditionally handled with two Intel 8259A programmable interrupt controller (PIC) chips (or an emulation thereof.) Each PIC provides eight levels of interrupt priority which, in essence, multiplex eight lines into one. One of the eight input lines (line 2, to be precise) from the first (master) chip routes to the output of the other (slave) PIC, which cascades the sixteen inputs from both chips into a single output. Of course, since one input is lost to the cascade configuration, only 15 lines are left for devices to utilize. [Table 2.4](#) lists the ISA priority levels and their typical assignments.

The 8259A chip can be programmed to respond to either edge-triggered or level-sensitive interrupts. This choice must be made for the entire chip, not on a line-by-line basis. Traditional BIOS code initializes both PICs to use edge-triggered interrupts. Therefore, on typical systems, ISA cards may not share IRQ lines. Some main boards that perform custom emulation of the 8259A do allow IRQ-by-IRQ programming for edge vs. level triggering.

DMA CAPABILITIES

The standard implementation of ISA DMA uses a pair of Intel 8237 DMAC chips (or an emulation thereof). Each of these chips provides four independent DMA channels. As with the PICs, the standard configuration cascades the two chips, routing the output of the master DMAC through the first channel of the slave DMAC. As before, the cascading technique results in the loss of one channel, leaving seven free for DMA devices. [Table 2.5](#) describes the channel configurations for the two DMACs.

When more than one DMA channel is in simultaneous use, the DMAC chips follow a software-selected scheme for prioritization. Typically, highest priority is given to channel 0 and lowest to channel 7. Also note that only the slave DMAC can be used to transfer words (16 bits). These upper channels can transfer data at twice the rate of the lower channels, since they move two bytes per cycle.

Table 2.4. Interrupt Priorities on ISA Systems

ISA Interrupt Priority Sequence			
Priority	IRQ Line	Controller	Use for...
Highest	0	Master	System timer
	1	Master	Keyboard
	2	Master	(Unavailable—pass through from slave)

Hints for Working with Hardware

Working with new hardware can be a real challenge. Hardware engineers follow a different design methodology than do software engineers. (Consider the user interface on a VCR.) The following hints may help make it easier to work with a new piece of hardware.

Learn about the Hardware

Before starting a new driver, learn as much as possible about the hardware itself. Most of the information needed should be in the hardware documentation. At the very least, ensure that information is available for

- - bus architecture
- - control registers
- - error and status reporting
- - interrupt behavior
- - data transfer mechanisms
- - device memory

BUS ARCHITECTURE

The hardware's bus architecture will have a big impact on the design of the driver. Auto recognition and configuration information must be clear. New drivers will be expected to participate in the Plug and Play mechanisms offered by Windows 2000.

CONTROL REGISTERS

The size and addressing scheme of the device's registers must be known. The purpose of each register and contents of any control, status, and data registers must be fully described. Odd behavior must be identified. For example:

- - Some device registers may be read-only or write-only.
- - A single register address may perform different functions on a read than it does on a write.
- - Data or status registers may not contain valid information until some fixed time interval after a command is issued.
- - Register access in a specific order may be required.

ERROR AND STATUS REPORTING

Determine any protocols used by the device for reporting hardware failures and device status.

INTERRUPT BEHAVIOR

Find out exactly what device conditions cause the hardware to generate an interrupt, and whether the device uses more than one interrupt vector. If working with a multidevice controller, interrupts may come from the controller itself, and there must be a mechanism to identify the actual device that wants attention.

DATA TRANSFER MECHANISMS

Drivers for programmed I/O devices are very different from DMA drivers. Some devices are capable of doing both kinds of I/O. In

Summary

This chapter presented a cursory look at hardware issues. A driver needs to find the devices it will be controlling and determine the system resources required. Some bus architectures make this easy, some hard. Later chapters describe the services provided by Windows 2000 to make this job easier.

The next chapter provides an overview of the Windows 2000 I/O process.

Chapter 3. Kernel-Mode I/O Processing

CHAPTER OBJECTIVES

- [How Kernel-Mode Code Executes](#)
- [Use of Interrupt Priorities by Windows 2000](#)
- [Deferred Procedure Calls \(DPCs\)](#)
- [Access to User Buffers](#)
- [Structure of a Kernel-Mode Driver](#)
- [I/O Processing Sequence](#)
- [Summary](#)

With the hardware issues covered in the previous chapter, this chapter introduces the role of the Windows 2000 Executive components in processing I/O requests.

This chapter covers three areas. First, in sections 3.1 through 3.4, it introduces key concepts and techniques important to I/O processing.

Next, the chapter describes the purpose of the various routines that make up a driver. The details of each routine are covered in subsequent chapters.

Finally, the entire life cycle of an I/O request is examined. A good understanding of the I/O flow of control is probably the most important piece of knowledge a driver author can have.

How Kernel-Mode Code Executes

The kernel-mode parts of Windows 2000 (including a device driver) consist of massive amounts of code. But just what causes this code to execute? All code executes within a hardware and software *context*. A context, as used here, describes the state of the system while a CPU instruction executes. It includes the state of all CPU registers (including the stack), the processor mode (user or kernel), and significantly, the state of the hardware page tables. This last item describes what memory can be seen by executing code, and where within the address space that memory is located.

Clearly, code must make assumptions about the context in which it executes. Windows 2000 defines three execution contexts for kernel-mode execution. In other words, kernel-mode driver code executes in one of three contexts.

Trap or Exception Context

[Chapter 1](#) described how user-mode code can request an OS service by trapping into kernel mode. When a kernel-mode routine executes, it may be because a user-mode application or service caused a hardware or software exception, or trap, to occur. In this case, the context of the kernel-mode code is largely that of the user code that caused the exception. The memory seen by kernel-mode code includes the same view as seen by the requesting user-mode thread.

When a user-mode thread makes a direct request of the I/O Manager, the I/O Manager executes within the context of the requester. In turn, the I/O Manager may call a dispatch routine within a device driver. Dispatch routines of a driver therefore execute within this exception context.

Interrupt Context

When the hardware (or software) generates an acknowledged interrupt, whatever code is executing within the system is stopped dead in its tracks. The executing context is saved and control is promptly handed over to a service routine appropriate for the kind of interrupt that occurred.

Clearly, the context of the executing code at the time of the interrupt is irrelevant and arbitrary. Kernel-mode code servicing the interrupt cannot make any assumptions about the state of the page tables. User-mode buffers must be considered unavailable in this context. Code running in interrupt context (which includes the bulk of driver routines) can make no assumptions about the current process or thread.

Kernel-Mode Thread Context

The final possibility is that a piece of code runs in the context of a separate kernel thread. Some drivers spawn separate threads to deal with devices that require polling or to deal with specialized timeout conditions. These kernel-mode threads are not significantly different from user-mode threads described in Win32 programming books. They execute when scheduled by the kernel's scheduler, in accordance with the assigned thread priority.

Like the interrupt context, kernel-mode thread context can make no assumption about the current process or thread. The state of the page tables is largely arbitrary as seen by the thread. [Chapter 14](#) discusses the use of kernel-mode threads.

Use of Interrupt Priorities by Windows 2000

The last chapter introduced the concept of interrupt priorities as a means of arbitrating among different I/O devices competing for CPU service. This section presents a scheme implemented by Windows 2000 that not only takes into account hardware interrupt prioritization, but extends the concept to include prioritization of execution context.

CPU Priority Levels

Since different CPU architectures have different ways of handling hardware interrupt priorities, Windows 2000 presents an idealized, abstract scheme to deal with all platforms. The actual implementation of the abstraction utilizes HAL routines that are platform-specific.

The basis for this abstract priority scheme is the *interrupt request level* (IRQL). The IRQL (pronounced *irk-al*) is a number that defines a simple priority. Code executing at a given IRQL cannot be interrupted by code at a lower or equal IRQL. [Table 3.1](#) lists the IRQL levels used in the Windows 2000 priority scheme. Regardless of the underlying CPU or bus architectures, this is how IRQL levels appear to a driver. It is important to understand that at any given time, instructions execute at one specific IRQL value. The IRQL level is maintained as part of the execution context of a given thread, and thus, at any given time, the current IRQL value is known to the operating system.

Table 3.1. IRQL Level Usage

IRQL Levels		
Generated By	IRQL Name	Purpose
Hardware	HIGHEST_LEVEL	Machine checks and bus errors
	POWER_LEVEL	Power-fail interrupts
	IPI_LEVEL	Interprocessor doorbell for MP systems
	CLOCK2_LEVEL	Interval clock 2
	CLOCK1_LEVEL	Interval clock 1
	PROFILE_LEVEL	Profiling timer
	DIRQLs	Platform-dependent number of levels for I/O device interrupts
Software	DISPATCH_LEVEL	Thread schedule and deferred procedure call execution
	APC_LEVEL	Asynchronous procedure call execution
	PASSIVE_LEVEL	Normal thread execution level

The actual hardware interrupt levels fall between DISPATCH_LEVEL and PROFILE-LEVEL of the IRQL abstraction. These hardware interrupt levels are defined as the device IRQLs (DIRQLs).

Interrupt Processing Sequence

When an interrupt reaches the CPU, the processor compares the IRQL value of the requested interrupt with the CPU's current IRQL value. If the IRQL of the request is less than or equal to the current IRQL, the request is temporarily ignored. The request remains pending until a later time when the IRQL level drops to a lower value.

On the other hand, if the IRQL of the request is higher than the CPU's current IRQL, the processor performs the following tasks:

1. Suspends instruction execution.
2. Saves just enough state information on the stack to resume the interrupted code at a later time.
3. Raises the IRQL value of the CPU to match the IRQL of the request, thus preventing lower priority interrupts from occurring.
4. Transfers control to the appropriate interrupt service routine for the requested interrupt.

When finished, the service routine executes a special instruction that dismisses the interrupt. This instruction restores the CPU state information from the stack (which includes the previous IRQL value) and control is returned to the interrupted code.

Notice that this scheme allows higher-IRQL requests to interrupt the service routines of lower-IRQL interrupts (an interrupt of an

Deferred Procedure Calls (DPCs)

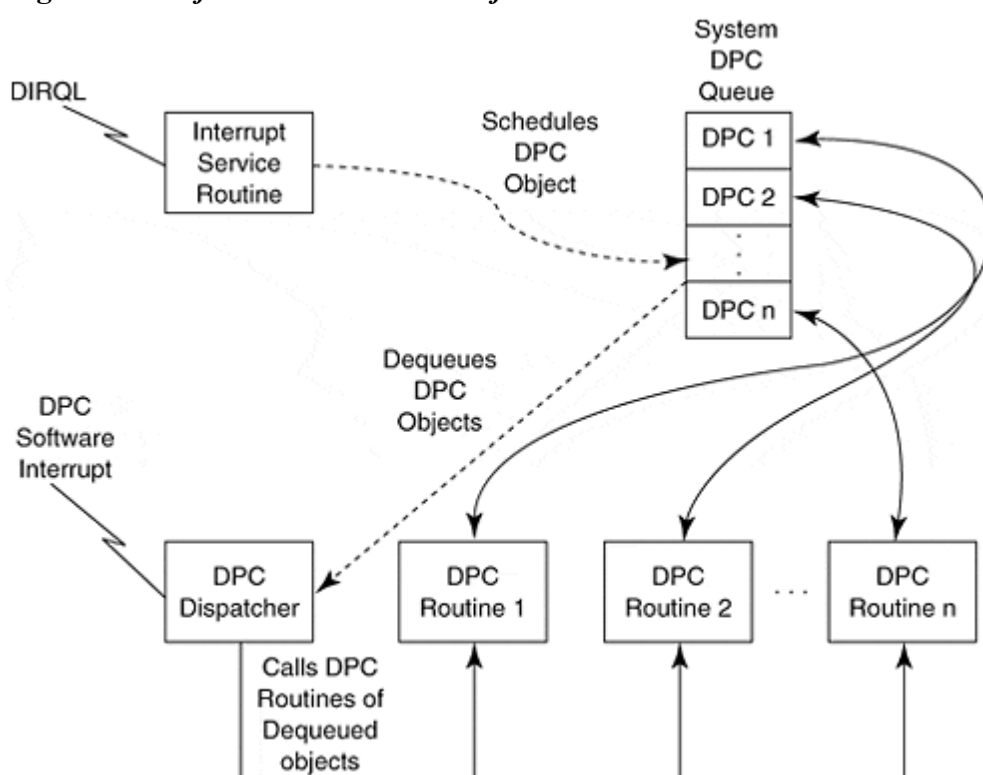
While a piece of kernel-mode code is running at an elevated IRQL, nothing executes (on the same CPU) at that or any lower IRQL. Of course, if too much code executes at too high an IRQL, overall system performance will degrade. Time-critical event handling could be deferred and cause more disastrous results.

To avoid these problems, kernel-mode code must be designed to execute as much code as possible at the lowest possible IRQL. One important part of this strategy is the *Deferred Procedure Call* (DPC).

Operation of a DPC

The DPC architecture allows a task to be *triggered*, but not *executed*, from a high-level IRQL. This *deferral of execution* is critical when servicing hardware interrupts in a driver because there is no reason to block lower-level IRQL code from executing if a given task can be deferred. [Figure 3.1](#) illustrates the operation of a DPC. Subsequent chapters present more specific information about the use of a DPC in a driver, but an overview is presented below.

Figure 3.1. Deferred Procedure Call flow.



1.

When some piece of code running at a high (e.g., hardware) IRQL wants to schedule some of its work at a lower IRQL, it adds a DPC object to the end of the system's DPC dispatching queue and requests a DPC software interrupt. Since the current IRQL is above `DISPATCH_LEVEL`, the interrupt won't be acknowledged immediately, but instead remains pending.

2.

Eventually, the processor's IRQL falls below `DISPATCH_LEVEL` and the previously pended interrupt is serviced by the DPC dispatcher.

3.

The DPC dispatcher dequeues each DPC object from the system queue and calls the function whose pointer is stored in the object. This function is called while the CPU is at `DISPATCH_LEVEL`.

4.

When the DPC queue is empty, the DPC dispatcher dismisses the `DISPATCH_LEVEL` software interrupt.

Device drivers typically schedule cleanup work with a DPC. This has the effect of reducing the amount of time the driver spends at its DIRQL and improves overall system throughput.

Behavior of DPCs

Access to User Buffers

When a user-mode thread makes an I/O request, it usually passes the address of a data buffer located in user space. Since user-mode addresses are referenced through the lower half (< 2 GB) of the page tables, a driver must cope with the possibility that the page tables will change before the request can be completed. This would occur if the driver code executed at Interrupt Context or Kernel-Mode Thread context. As previously discussed, the lower half of the page tables are changed with each process switch. Thus, code executing with an arbitrary page table state cannot assume that any user-mode address is valid.

Worse, the user buffer may be paged out of RAM and exist only on the system's swap file on disk. User memory is always subject to swap-out if the system needs RAM for other processes.

In general, user memory is pagable, and by rule, pagable memory may not be accessed at `DISPATCH_LEVEL_IRQL` or higher. This rule is necessary since the tasks providing page-in service of the requested memory might need a device at a lower `DIRQL` than that which needed the page.

Buffer Access Mechanisms

Now that the problem is defined, a solution is needed. Fortunately, the I/O Manager provides drivers with two different methods for accessing user buffers. When a driver initializes, it tells the I/O Manager which strategy it plans to use. The choice depends on the nature and speed of the device.

The first strategy is to ask the I/O Manager to copy the entire user buffer into dedicated system RAM, which remains fixed (i.e., not paged) in both the page tables and physical memory. The device uses the copy of the buffer to perform the requested I/O operation. Upon completion, the I/O Manager conveniently copies the system buffer back to the user buffer. Actually, only one copy operation is performed. On I/O write requests, the user buffer is copied before presentation to the driver. On I/O read requests, the system buffer is copied after the driver marks the request as completed. Standard read or write requests do not require bidirectional copying.

The first technique is known as *buffered I/O* (BIO). It is used by slower devices that do not generally handle large data transfers. The technique is simple for driver logic, but requires the somewhat time-consuming job of buffer copying.

The second technique avoids the buffer copy by providing the driver with direct access to the user's buffer in physical memory. At the beginning of the I/O operation, the I/O Manager "locks down" the entire user buffer into memory, thus preventing the block from swap-out and causing a deadly page fault. It then constructs a list of page table entries that are mapped into slots above 2 GB and are thus not subject to process context switches. With the memory and page table entries locked for the duration of the I/O request, driver code may safely reference the user buffer. (Note, however, that the original user address is translated into another logical address, valid only from kernel-mode code. The driver must use the translated address.)

This second technique is well-suited for fast devices that transfer large blocks of data. It is known as *direct I/O* (DIO). DMA devices almost always use this technique and it is further discussed in [chapter 12](#).

Structure of a Kernel-Mode Driver

A kernel-mode driver looks very different from conventional applications. In general, a driver is just a collection of routines that are called by operating system software (usually the I/O Manager). Flow charts don't provide much benefit when diagramming control paths of a device driver. The routines of the driver sit passively until they are invoked by routines of the I/O Manager.

Depending on the driver, the I/O Manager might call a driver routine in any of the following situations:

-
- When a driver is loaded
-
- When a driver is unloaded or the system is shutting down
-
- When a device is inserted or removed
-
- When a user-mode program issues an I/O system service call
-
- When a shared hardware resource becomes available for driver use
-
- At various points during an actual device operation

The remainder of this section briefly describes the major categories of routines that make up a kernel-mode driver. The details of each of these kinds of driver routines are discussed in subsequent chapters.

Driver Initialization and Cleanup Routines

When a driver loads into the system, various tasks must be performed. Similarly, drivers need to clean up before they are unloaded. There are several routines that a driver may supply to perform these operations.

DriverEntry ROUTINE

The I/O Manager calls this routine when a driver is first loaded, perhaps as early as boot time, but drivers may be dynamically loaded at any time. The DriverEntry routine performs all first-time initialization tasks, such as announcing the addresses of all other driver routines. It locates hardware that it will control, allocates or confirms hardware resource usage (ports, interrupts, DMA), and provides a device name visible to the rest of the system for each hardware device discovered. For WDM drivers participating in Plug and Play, this hardware allocation step is deferred to a later time and routine, the AddDevice function.

REINITIALIZE ROUTINE

Some drivers may not be able to complete their initialization during DriverEntry. This could occur if the driver was dependent on another driver that had yet to load, or if the driver needed more support from the operating system than had booted at the time DriverEntry was initially called. These kinds of drivers can ask that their initialization be deferred by supplying a Reinitialize routine during DriverEntry.

UNLOAD ROUTINE

The I/O Manager calls the Unload routine of a driver when a driver is unloaded dynamically. It must reverse every action that DriverEntry performed, leaving behind no allocated resource. This includes removing the system-wide device name supplied during initialization for all controlled devices. For WDM drivers, device resources (and names) are removed as each device is removed, during RemoveDevice.

SHUTDOWN ROUTINE

Surprisingly, the Unload routine is not invoked during a system shutdown. Since the system is "going away" anyway, a perfect cleanup is not required. Instead, the I/O Manager invokes a driver's Shutdown routine to provide an opportunity to place hardware into a quiescent state.

BUGCHECK CALLBACK ROUTINE

If a driver needs to gain control in the event of a system crash, it can provide a Bugcheck routine. This routine, when properly

I/O Processing Sequence

It is important to understand the complete life cycle of an I/O request. This section describes the flow of a request—from user-mode code to I/O Manager to device driver.

An I/O request goes through several stages during its life.

-
- Preprocessing by the I/O Manager
-
- Preprocessing by the device driver
-
- Device start and interrupt service
-
- Postprocessing by the driver
-
- Postprocessing by the I/O Manager

The following sections describe the stages in more detail.

Preprocessing by the I/O Manager

This phase performs device-independent preparation and verification of an I/O request.

1.
 - The Win32 subsystem converts the request into a native system service call. The system service dispatcher traps into kernel mode and into the I/O Manager.
2.
 - The I/O Manager allocates a data structure known as an *I/O Request Packet* (IRP). The next chapter describes an IRP in more detail, but an IRP can be thought of as a work order presented to a driver and its device. The IRP is filled with necessary information, including a code which identifies the type of I/O request.
3.
 - The I/O Manager performs some validation of the arguments passed with the request. This includes verifying the file handle, checking access rights to the file object, ensuring that the device supports the requested function, and validating the user buffer addresses.
4.
 - If the device requests a buffered I/O (BIO) operation, the I/O Manager allocates a nonpaged pool buffer, and for a write request, copies data from user space into the system buffer. If the device requests direct I/O (DIO), the user's buffer is locked down and a list of page descriptors is built for use by the driver.
5.
 - The I/O Manager invokes the appropriate driver dispatch routine.

Preprocessing by the Device Driver

Each driver builds a dispatch table of entry points for each supported I/O function request. The I/O Manager uses the function code of the I/O request to index into this table and invoke the appropriate driver Dispatch routine. The routine performs the following tasks:

1.
 - It performs additional parameter validation. The device driver can detect device-specific limitations unknown to the I/O Manager. For example, a user might request a data transfer size in violation of the capabilities of a specific device (or driver).
2.
 - If the request can be handled without device activity (e.g., reading zero bytes), the Dispatch routine simply completes the request and sends the IRP back to the I/O Manager, marked as complete.

Summary

The quick tour of the Windows 2000 I/O subsystem is now complete. The structure of a Windows 2000 basic device driver should be clear—only the details remain.

The next chapter deals with some of these details: the data structures created and used by the I/O Manager and the device driver.

[< BACK NEXT >](#)

Chapter 4. Drivers and Kernel-Mode Objects

CHAPTER OBJECTIVES

- [Data Objects and Windows 2000](#)
- [I/O Request Packets \(IRPs\)](#)
- [Driver Objects](#)
- [Device Objects and Device Extensions](#)
- [Controller Objects and Controller Extensions](#)
- [Adapter Objects](#)
- [Interrupt Objects](#)
- [Summary](#)

Data structures are the lifeblood of most operating systems as well as most complicated applications, and Windows 2000 is no exception. Windows 2000 is different from most operating systems because it uses a taste of object technology to manage its data structures. This chapter examines the Windows 2000 approach to data structures and objects, then introduces the major structures involved in processing I/O requests.

Data Objects and Windows 2000

Object-oriented programming (OOP) is a proven software design technique to provide overall organization of code and high-level encapsulation of major design concepts. In this scheme, a data structure is grouped with the code that modifies it. Together, the code and its associated data are called *objects*. The data of the object remains opaque to the users of the object. Using code must invoke methods of the object to manipulate its state. The methods of an object form a strict interface for its use.

The overall goal of the object-orientation technique is to improve the reliability, robustness, and reusability of software by hiding implementation details from the users of an object.

Windows 2000 and OOP

Using a strict definition of OOP, the design of Windows 2000 isn't truly object-oriented. Rather, it can be thought of as object-based because it manages its internal data structures in an object-like way. In particular, Windows 2000 defines not only necessary data structures, but also groups of access functions that are allowed to manipulate those data structures. All other modules are expected to access functions to manipulate the contents of the structures.

In practice, not all elements of all data structures are completely opaque. In fact, since kernel-mode driver code is trusted code, there is no real way to enforce the object boundary. Additionally, the I/O Manager treats device drivers as trusted components of itself. A driver is required to read and write some offsets within some I/O Manager data structures directly.

Windows 2000 Objects and Win32 Objects

The Win32 API also defines kernel objects. These user-mode objects differ from internal OS objects in two important ways. First, in almost all cases, internal objects have no external name. This is because these objects aren't exported to user mode and, therefore, don't need to be managed by the Object Manager.

Second, user-mode code uses *handles* to reference the objects it creates or opens. Internal objects are referenced with a direct memory pointer. This pointer is routinely passed between I/O Manager and device driver. In some cases, the driver itself will allocate and initialize memory for the object.

I/O Request Packets (IRPs)

Almost all I/O under Windows 2000 is packet-driven. Each separate I/O transaction is described by a work order that tells the driver what to do and tracks the progress of the request through the I/O subsystem. These work orders take the form of a data structure called an *I/O Request Packet* (IRP), and this section describes their use and purpose.

1.

With each user-mode request for I/O, the I/O Manager allocates an IRP from nonpaged system memory. Based on the file handle and I/O function requested by the user, the I/O Manager passes the IRP to the appropriate driver dispatch routine.

2.

The dispatch routine checks the parameters of the request, and if valid, passes the IRP to the driver's Start I/O routine.

3.

The Start I/O routine uses the contents of the IRP to begin a device operation.

4.

When the operation is complete, the driver's DpcForIsr routine stores a final status code in the IRP and returns it to the I/O Manager.

5.

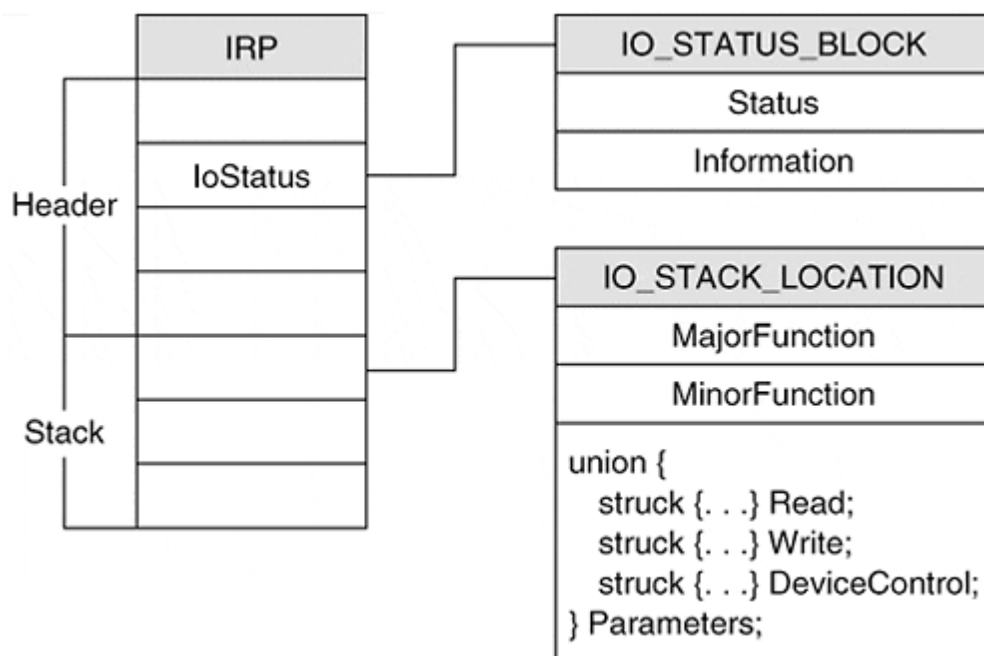
The I/O Manager uses the information in the IRP to complete the request and send the user the final status.

The procedure just described is, in fact, a simplified model of IRP processing. It only applies in a flat driver model. When drivers are layered, one upon the other, the procedure is more complex. A single IRP may travel through several layers of drivers before the request is completed. Additionally, higher-level drivers can create new IRPs and disperse them to other drivers.

Layout of an IRP

An IRP is a variable-sized structure allocated from nonpaged pool. [Figure 4.1](#) shows that an IRP has two sections.

Figure 4.1. Structure of an IRP.



•

A header area containing general bookkeeping information

•

One or more parameter blocks called *I/O stack locations*

IRP HEADER

This area of the IRP holds various pieces of information about the overall I/O request. Some parts of the header are directly accessible to the driver, while other pieces are the exclusive property of the I/O Manager. [Table 4.1](#) lists the fields in the header that

Driver Objects

DriverEntry is the only driver routine with an exported name. When the I/O Manager needs to locate other driver functions, it uses the Driver object associated with a specific device. This object is basically a catalog that contains pointers to various driver functions. The life of a driver object is explained below.

1.

The I/O Manager creates a driver object whenever it loads a driver. If the driver fails during initialization, the I/O Manager deletes the object.

2.

During initialization, the DriverEntry routine loads pointers to other driver functions into the driver object.

3.

When an IRP is sent to a specific device, the I/O Manager uses the associated driver object to find the right Dispatch routine.

4.

If a request involves an actual device operation, the I/O Manager uses the driver object to locate the driver's Start I/O routine.

5.

If the driver is unloaded, the I/O Manager uses the driver object to find an Unload routine. When the Unload routine returns, the I/O Manager deletes the Driver object.

Layout of a Driver Object

There is a unique driver object for each driver currently loaded in the system. [Figure 4.2](#) illustrates the structure of the driver object. As you can see, the driver object also contains a pointer to a linked list of devices serviced by this driver. A driver's Unload routine can use this list to locate any devices it needs to delete.

Figure 4.2. The driver object.

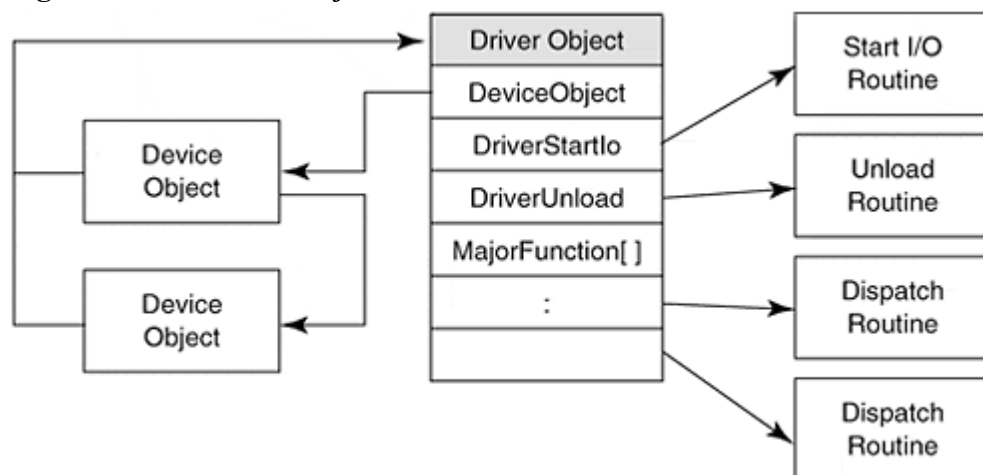


Table 4.5. Externally Visible Fields of a Driver Object

Driver Object Fields	
Field	Description
PDRIVER_STARTIO DriverStartIo	Address of driver's Start I/O routine
PDRIVER_UNLOAD DriverUnload	Address of driver's Unload routine
PDRIVER_DISPATCH MajorFunction[]	Table of driver's Dispatch routines, indexed by I/O operation code
PDEVICE_OBJECT DeviceObject	Linked list of device objects created by this driver

Unlike other objects, there are no access functions for modifying driver objects. Instead, the DriverEntry routine sets various fields directly. [Table 4.5](#) lists the fields a driver is allowed to touch.

Device Objects and Device Extensions

Both the I/O Manager and a driver need to know what's going on with an I/O device at all times. *Device objects* make this possible by keeping information about the device's characteristics and state. There is one device object for each virtual, logical, and physical device on the system. The life cycle of a device object is shown below.

1.

The DriverEntry routine creates a device object for each of its devices. For WDM drivers, the Device object is created by the AddDevice Plug and Play routine.

2.

The I/O Manager uses a back-pointer in the device object to locate the corresponding driver object. There it can find driver routines to operate on I/O requests. It also maintains a queue of current and pending IRPs attached to the device object.

3.

Various driver routines use the device object to locate the corresponding device extension. As an I/O request is processed, the driver uses the extension to store any device-specific state information.

4.

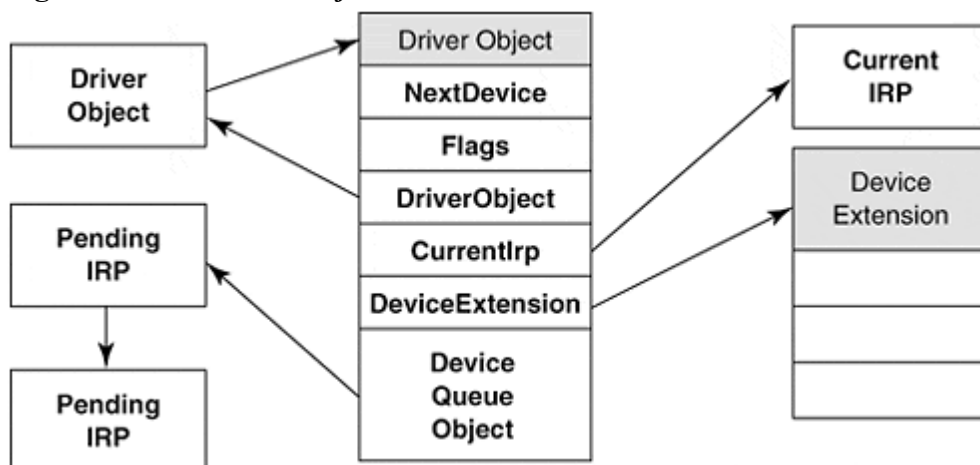
The driver's Unload routine deletes the device object when the driver is unloaded. The act of deleting the device object also deletes the associated device extension. For WDM drivers, RemoveDevice performs the task of deleting the Device object.

Physical device drivers are not alone in their use of device objects. [Chapter 15](#) describes the way higher-level drivers use device objects.

Layout of the Device Object

[Figure 4.3](#) illustrates the structure of the device object and its relationship to other structures. Although the device object contains a lot of data, much of it is the exclusive property of the I/O Manager. A driver should limit its access to only those fields listed in [Table 4.6](#).

Figure 4.3. The device object.



Manipulating Device Objects

[Table 4.7](#) lists many of the I/O Manager functions that operate on device objects. The I/O Manager also passes a device object pointer as an argument to most of the routines in a driver.

Table 4.6. Externally Visible Fields of a Device Object

Device Object Fields	
Field	Description
PVOID DeviceExtension	Points to device extension structure
PDRIVER_OBJECT DriverObject	Points to driver object for this device
ULONG Flags	Specifies buffering strategy for device
	DO_BUFFERED_IO
	DO_DIRECT_IO

Controller Objects and Controller Extensions

Some peripheral adapters manage more than one physical device using the same set of control registers. The floppy disk controller is one example of this architecture. This kind of hardware poses a synchronization dilemma. If the driver tries to perform simultaneous operations on more than one of the connected devices without first synchronizing its access to the shared register space, the control registers receive confusing values. To help with this problem, the I/O Manager provides *controller objects*.

The controller object is a kind of mutex that can be owned by only one device at a time. Before accessing any device registers, the driver asks that ownership of the controller object be given to a specific device. If the hardware is free, ownership is granted. If not, the device's request is put on hold until the current owner releases the hardware. By managing the controller object, the I/O Manager guarantees that multiple devices will access the hardware in a serial fashion. The life cycle of a typical controller object is described below.

1.

The DriverEntry (or AddDevice) routine creates the Controller object and usually stores its address in a field of each device's Device Extension.

2.

Before it starts a device operation, the Start I/O routine asks for exclusive ownership of the controller object on behalf of a specific device.

3.

When the controller object becomes available, the I/O Manager grants ownership and calls the driver's ControllerControl routine. This routine sets up the device's registers and starts the I/O operation. As long as this device owns the controller object, any further requests for ownership block at step 2 until the object is released.

4.

When the device operation is finished, the driver's DpcForIsr routine releases the Controller object, making it available for use by other pending requests.

5.

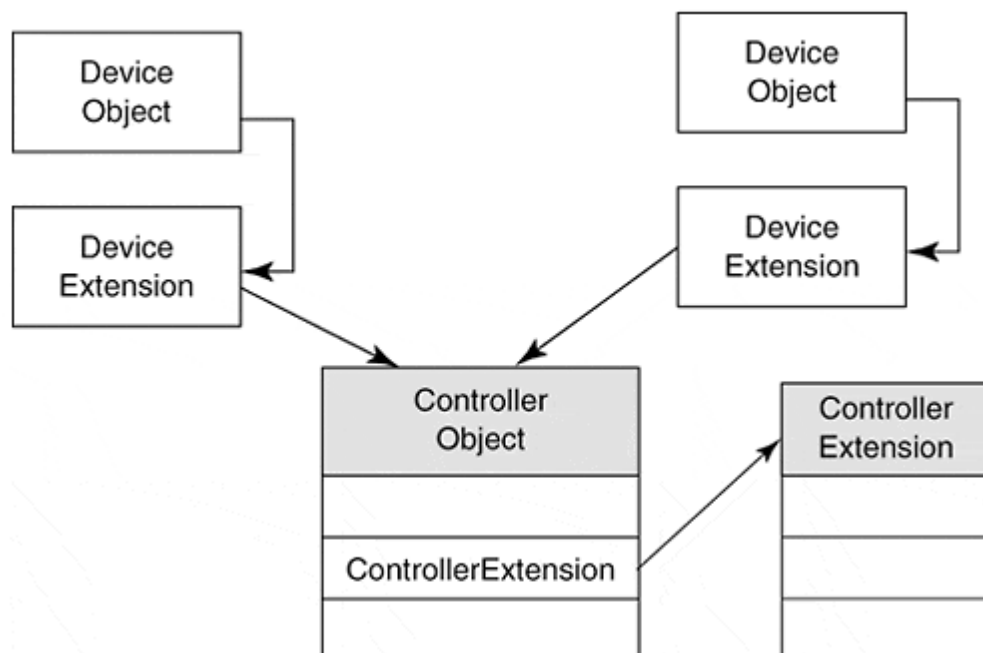
The driver's Unload routine deletes the controller object when the driver is unloaded.

Obviously, not all drivers need a controller object. If an interface card supports only one physical or virtual device, or if multiple devices on the same card don't share any control registers, then there is no need to create a controller object.

Layout of the Controller Object

[Figure 4.4](#) shows the relationship of a Controller object to other system data structures. The only externally visible field in a Controller object is the **PVOID ControllerExtension** field, which contains a pointer to the extension block.

Figure 4.4. The controller object.



Manipulating Controller Objects

Adapter Objects

Just as multiple devices on the same controller need to coordinate their hardware access, so it is that devices that perform DMA need an orderly way to share system DMA resources. The I/O Manager uses adapter objects to prevent arguments over DMA hardware. There is one adapter object for each DMA data transfer channel on the system.

Like a controller object, an adapter object can be owned by only one device at a time. Before starting a DMA transfer, the Start I/O routine asks for ownership of the adapter object. If the hardware is free, ownership is granted. If not, the device's request is put on hold until the current owner releases the hardware. Obviously, if the device supports only programmed I/O, it has no need for an adapter object. The life cycle of the adapter object is described below.

1.

The HAL creates Adapter objects for any DMA data channels detected at boot time.

2.

The DriverEntry or AddDevice routine locates the adapter object for its device and stores that pointer in the device or controller extension. Adapter objects for nonsystem (i.e., bus master) DMA hardware may be created on the fly.

3.

The Start I/O routine requests ownership of the adapter object on behalf of a specific device.

4.

When ownership is granted, the I/O Manager calls the driver's adapter Control routine. This routine then uses the adapter object to set up a DMA transfer.

5.

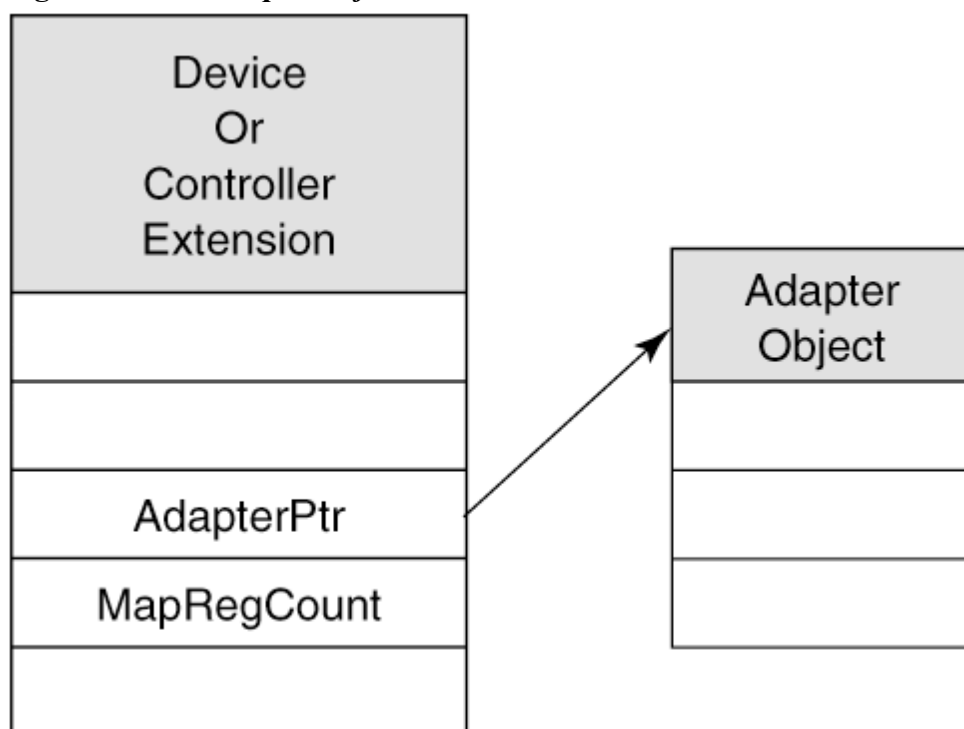
The driver's DpcForIsr routine may use the adapter object to perform additional operations in the case of a split transfer. When a transfer is finished, DpcForIsr releases the adapter object.

Another important function of the adapter object is to manage *mapping registers*. The HAL uses these registers to map the standard physical pages of a user's buffer onto the contiguous range of addresses required by most DMA hardware. The complete mechanics of DMA transfers are covered in detail in [chapter 12](#).

Layout of an Adapter Object

[Figure 4.5](#) illustrates the relationship of adapter objects to other structures. As the diagram shows, the adapter object is completely opaque and has no externally visible fields. When working with DMA devices, the pointer to the adapter object, as well as the number of mapping registers it supports, should be stored in the device extension or controller extension structure.

Figure 4.5. The adapter object



Interrupt Objects

The last kernel object described in this chapter is the interrupt object. Interrupt objects simply give the kernel's interrupt dispatcher a way to find the right service routine when an interrupt occurs. The life cycle of an interrupt object is described below.

1.

The DriverEntry or AddDevice routine creates an interrupt object for each interrupt vector supported by the device or the controller.

2.

When an interrupt occurs, the kernel's interrupt dispatcher uses the Interrupt object to locate the Interrupt Service routine.

3.

The Unload or RemoveDevice routine deletes the interrupt object after disabling interrupts from the device.

A driver does not interact with interrupt objects other than to create and delete them. A pointer to the interrupt object is typically stored in the device extension or controller extension.

Layout of an Interrupt Object

[Figure 4.6](#) illustrates the structure of an interrupt object. Like adapter objects, they are completely opaque and have no externally visible fields.

Figure 4.6. The interrupt object.

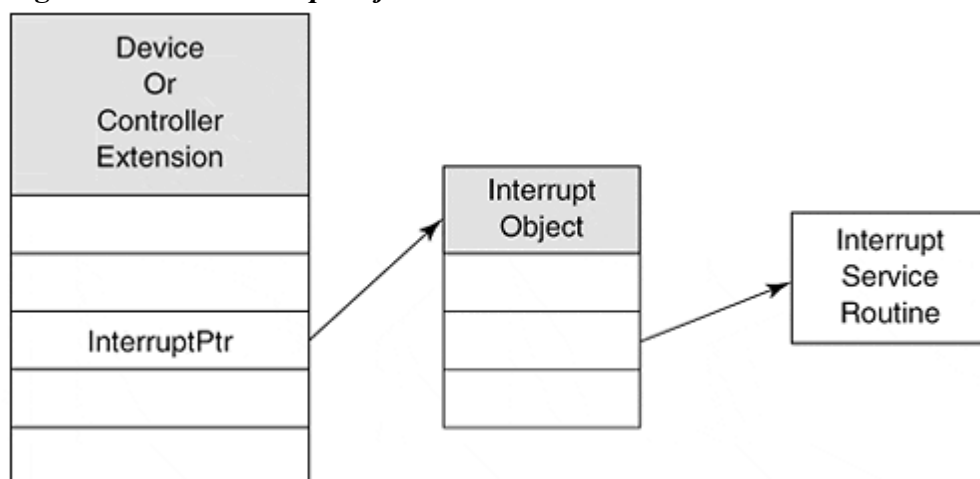


Table 4.10. Access Functions for an Interrupt Object

Interrupt Object Access Functions		
Function	Description	Called by...
HalGetInterruptVector	Converts bus-relative interrupt vector to systemwide value	DriverEntry
IoConnectInterrupt	Associates Interrupt Service routine with an interrupt vector	DriverEntry or AddDevice
KeSynchronizeExecution	Synchronizes driver routines that run at different IRQLs	(Various)
IoDisconnectInterrupt	Removes interrupt object	Unload or RemoveDevice

Manipulating Interrupt Objects

Several system components export functions that work with interrupt objects. [Table 4.10](#) lists the most common ones.

Summary

Although it may seem as if there are a lot of objects involved in I/O processing, all are necessary and important. With this chapter, the background material explanation is complete. The next chapter begins the fun work of writing some actual driver code.

Chapter 5. General Development Issues

CHAPTER OBJECTIVES

- - [Driver Design Strategies](#)
- - [Coding Conventions and Techniques](#)
- - [Driver Memory Allocation](#)
- - [Unicode Strings](#)
- - [Interrupt Synchronization](#)
- - [Synchronizing Multiple CPUs](#)
- - [Linked Lists](#)
- - [Summary](#)

Writing kernel-mode code is not the same as writing an application program because the driver is a trusted component of the system. Code must be carefully written and follow special rules. This chapter is a short manual of good and sometimes required etiquette for driver writers.

Driver Design Strategies

Like most software, drivers benefit from an organized approach to development. This section gives some guidelines that may help shorten development time.

Use Formal Design Methods

Traditionally, writing a device driver implied a license to code without a design strategy. In the haste to test brand new hardware, the unit test code became the basis for the ultimate driver. Perhaps, too, many device drivers were written by programmers unfamiliar with the operating system. The first driver written is really a journey of discovery as the driver author learns the overall I/O architecture of the system.

Fortunately (or perhaps sadly, depending on one's perspective), the days of software cowboys are over. Between complicated hardware and operating systems, there is no longer room for design-as-you-go coding. Writing Windows 2000 drivers certainly entails a steep learning curve. Good software engineering and practices are required for success.

A short list of design techniques, some of which are borrowed from real-time design methodology, are listed below.

- Data flow diagrams can help break a driver into discrete functional units. These diagrams make it easier to visualize how the functional units in a driver relate to each other, and how they transform input data into output data.
- State-machine models are another good way to describe the flow of control in a driver—especially one that manages an elaborate hardware or software protocol. In the process of verifying the state machine, potential synchronization issues can be identified.
- An analysis of expected data arrival rates or mandatory input-to-output response will give a set of quantitative timing requirements. These are important to establish the overall performance goals of the driver.
- Another useful tool is an explicit list of external events and the driver actions that these events should trigger. This list should include both hardware events from the device and software I/O requests from users.

Using these techniques helps decompose a driver into well-defined functional units. In some cases, this might mean breaking a single, monolithic driver into a pair of port and class drivers that handle hardware-dependent and hardware-independent functions. In any event, the time spent analyzing the design of a driver at the start of the project more than pays for itself in reduced debugging and maintenance.

Use Incremental Development

Once an initial analysis and design is completed, it's time to start the actual development. Following the steps below can reduce debugging time by helping to detect problems while they're still easy to find.

1.
Decide which kinds of kernel-mode objects a driver needs.
2.
Decide what context or state information a driver needs, and decide where it will be stored.
3.
Write the DriverEntry and Unload routines first. Do not initially add Plug and Play support. This allows for testing the driver manually using the Computer Management Console.
4.
Add driver dispatch routines that process IRP_MJ_CREATE and IRP_MJ_CLOSE operations. These routines typically do not require device access. The driver can then be tested with a simple Win32 program that calls **CreateFile** and **CloseHandle**.
5.
Add code that finds and allocates the driver's hardware, as well as code to deallocate the hardware when the driver unloads. If the hardware supports Plug and Play detection, this step tests the hardware and the driver's ability to automatically load.
- 6.

Coding Conventions and Techniques

Writing a trusted kernel-mode component is not the same as writing an application program. This section presents some basic conventions and techniques to make it easier to code in this environment.

General Recommendations

First of all, here are some general guidelines to follow when writing a driver.

Avoid the use of assembly language in a driver. It makes the code hard to read, nonportable, and difficult to maintain. The C programming language is only a small step away from assembly language, anyway. Further, the HAL macros provide the only safe mechanism to access I/O device registers. Therefore, the use of assembly language in a driver should be extremely rare. Be sure to isolate such code into its own module.

For platform-specific code, provide a separate module, or at the very least, bracket it with **#ifdef/#endif** directives.

A driver should not be linked with the standard C runtime library. Besides being wasteful of memory space (each of 20 drivers should not include the same C runtime library support), some library routines are stateful or hold context information that is not thread-safe or driver-safe.

This particular guideline is perhaps the most uncomfortable aspect of writing device drivers. C programmers who live with their runtime environment day in and day out, often do not make a clear distinction between the C language and the C runtime library. The C runtime library requires initialization. It attempts to initialize a heap area and, in the case of C++, invoke constructors of global objects. All of these tasks interfere with proper driver operation.

Windows 2000 provides its own environmental support for kernel-mode code. This support includes **RtlXxx** functions (RunTime Library) to provide many of the common C runtime library services. Many of these routines are described later in this chapter.

Driver projects should be managed with source-code control. Microsoft Visual Source Safe® is a popular choice. For large projects that span multiple platforms, ClearCase® from Rational Software should also be considered.

Naming Conventions

All large software projects should adopt some standard naming convention for routines and data defined throughout the code. Device driver projects are no exception. Naming conventions improve the efficiency of development, debugging, testing, and maintenance of the driver.

Microsoft provides a naming convention for use with the DDK. A header file, **NTDDK.h**, defines all the data types, structures, constants, and macros used by base-level kernel-mode drivers. By DDK convention, all of these types of names are capitalized. Even native C-language data types are provided a corresponding DDK name. For example, the C data type **void*** is given the name **PVOID** by NTDDK.h. These definitions make future ports to 64-bit platforms easier.

Microsoft recommends that a driver-specific prefix be added to each of the standard driver routines. For example, if writing a mouse class driver, the Start I/O routine might be named **MouseClassStartIo**. Similarly a shorter two or three character prefix should be applied to internal names. This yields a name such as **MouConfiguration**. This recommendation is often, but not always, followed by driver authors.

Regardless of what convention is chosen for a driver project, it is important to establish a consistent way of naming entities within the entire project. It pays to spend a few hours making this decision early in the development life cycle.

Header Files

Besides including NTDDK.h or WDM.h, a driver should use private header files to hide various hardware and platform dependencies. For example, register access macros should be provided in a private header file. These macros should be surrounded by **#ifdef** compiler directives that allow for simple platform-to-platform porting. This technique, of course, solves the issue of register access differences between I/O space and memory space.

Even if portability were not a concern, register access macros make the driver easier to read and maintain. The following code fragment is an example of some hardware beautification macros for a parallel port device. The example assumes that some initialization code in the driver has put the address of the first device register in the **PortBase** field of the device extension.

```
//
// Define device registers as relative offsets
//
#define PAR_DATA 0
```

Driver Memory Allocation

An important aspect of programming concerns the need to allocate storage. Unfortunately, drivers don't have the luxury of making simple calls to **malloc** and **free**, or **new** and **delete**. Instead, care must be taken to ensure that memory of the right type is allocated. Drivers must also be sure to release any memory they allocate, since there is no automatic cleanup mechanism for kernel-mode code. This section describes techniques a driver can use to work with temporary storage.

Memory Available to Drivers

There are three options for allocating temporary storage in a driver. The criteria used to select the right storage type depends on factors such as duration, size, and from what IRQL level the code which accesses it will be running. The available options are

-

Kernel stack.

The *kernel stack* provides limited amounts of nonpaged storage for local variables during the execution of driver routines.

-

Paged pool.

Driver routines running below DISPATCH_LEVEL IRQL can use a heap area called *paged pool*. As the name implies, memory in this area is pagable, and a page fault can occur when it is accessed.

-

Nonpaged pool.

Driver routines running at elevated IRQLs need to allocate temporary storage from another heap area called *nonpaged pool*. The system guarantees that the virtual memory in nonpaged pool is always physically resident. The device and controller extensions created by the I/O Manager come from this pool area.

Because a driver must be reentrant, global variables are almost never declared. The only exception occurs for read-only data. Otherwise, the attempt by one thread of execution to store into the global variable interferes with another thread's attempt to read or write the same data.

Of course, local static variables in a driver function are just as bad. State for a driver must be kept elsewhere, such as in a device extension as just described.

Working with the Kernel Stack

On the x86 platform, the kernel stack is only 12 KB in size. On other platforms, the stack size is 16 KB. Therefore, the kernel stack must be considered a precious resource. Overflowing the kernel stack will cause an exception—something to be avoided at all costs in kernel mode. To avoid kernel stack overflowing, follow these guidelines.

-

Don't design a driver in such a way that internal routines are deeply nested. Keep the call tree as flat as possible.

-

Avoid recursion, but where required, limit the depth of recursion. Drivers are not the place to be calculating Fibonacci series using a recursive algorithm.

-

Do not use the kernel stack to build large data structures. Use one of the pool areas instead.

Another characteristic of the kernel stack is that it lives in cached memory. Therefore, it should not be used for DMA operations. DMA buffers should be allocated from nonpaged pool. [Chapter 12](#) describes DMA caching issues in more detail.

Working with the Pool Areas

To allocate memory in the pool area, drivers use the kernel routines **ExAllocatePool** and **ExFreePool**.

These functions allow the following kinds of memory to be allocated:

-

NonPagedPool

Unicode Strings

All character strings in the Windows 2000 operating system are stored internally as Unicode. The Unicode scheme uses 16 bits to represent each character and makes it easier to port applications and the OS itself to most languages of the world. Unicode is an industry standard (incidentally, the character coding standard for Java). More information can be found at the Web site <http://www.Unicode.org>. Unless otherwise noted, any character strings a driver sends to or receives from Windows 2000 will be Unicode. Note, however, that data transfer between a user's buffer and the device is not necessarily Unicode. Data transfers are considered to be binary and transparent to the I/O subsystem of Windows 2000.

Unicode String Data Types

The Unicode data type is now part of the C-language specification. To use "wide" characters in a program, perform the following:

Table 5.2. The UNICODE_STRING Data Structure

UNICODE_STRING, *PUNICODE_STRING	
Field	Contents
USHORT Length	Current string length in bytes
USHORT MaximumLength	Maximum string length in bytes
PWSTR Buffer	Pointer to driver-allocated buffer holding the real string data

- Prefix Unicode string constants with the letter L. For example, "some text" generates Unicode text, while "some text" produces 8-bit ANSI.
- Use the **wchar_t** data type for Unicode characters. The DDK header files provide a typedef, **WCHAR**, for the standard wchar_t, and **PWSTR**, for wchar_t*.
- Use the constant UNICODE_NULL to terminate a Unicode string. A UNICODE_NULL is defined to be 16 bits of zero.

Windows 2000 system routines work with a Unicode structure, UNICODE_STRING, described in [Table 5.2](#). The purpose of this structure is to make it easier to pass around Unicode strings and to help manage them. Although the standard C library provides Unicode string functions to perform common operations (e.g., wcsncpy is equivalent to strcpy for wchar_t* data), this environment is not available to kernel-mode driver code.

Incidentally, the DDK also defines an ANSI_STRING structure. It is identical to the UNICODE_STRING structure except that the buffer offset is of type char*. Several Rtl conversion routines require the data type of ANSI_STRING.

Working with Unicode

The kernel provides a number of functions for working with the ANSI and Unicode strings. These functions replace (albeit clumsily) the standard C library routines that work with Unicode. [Table 5.3](#) presents several of these functions. The Windows 2000 DDK provides the authoritative list and usage of the functions and should be reviewed. Some of these functions have restrictions on the IRQL levels from which they can be called, so care must be taken when using them. To be safe, it is best to restrict the use of all Rtl Unicode functions to PASSIVE_LEVEL IRQL.

Working with Unicode can be frustrating primarily because the length in bytes of a Unicode string is twice the content length. C programmers are ingrained with the belief that one character equals one byte, but with Unicode the rule is changed. When working with Unicode, consider the following:

Table 5.3. Common Unicode Manipulation Functions

Unicode String Manipulation Functions	
Function	Description
RtlInitUnicodeString	Initializes a UNICODE_STRING from a NULL-terminated Unicode string
RtlAnsiStringToUnicodeSize	Calculates number of bytes required to hold a converted ANSI string
RtlAnsiStringToUnicodeString	Converts an ANSI string to Unicode
RtlIntegerToUnicodeString	Converts an integer to Unicode text
RtlAppendUnicodeStringToString	Concatenates two Unicode strings
RtlCopyUnicodeString	Copies a source string to a destination
RtlUnicodeStringToAnsiString	Converts a Unicode string to ANSI

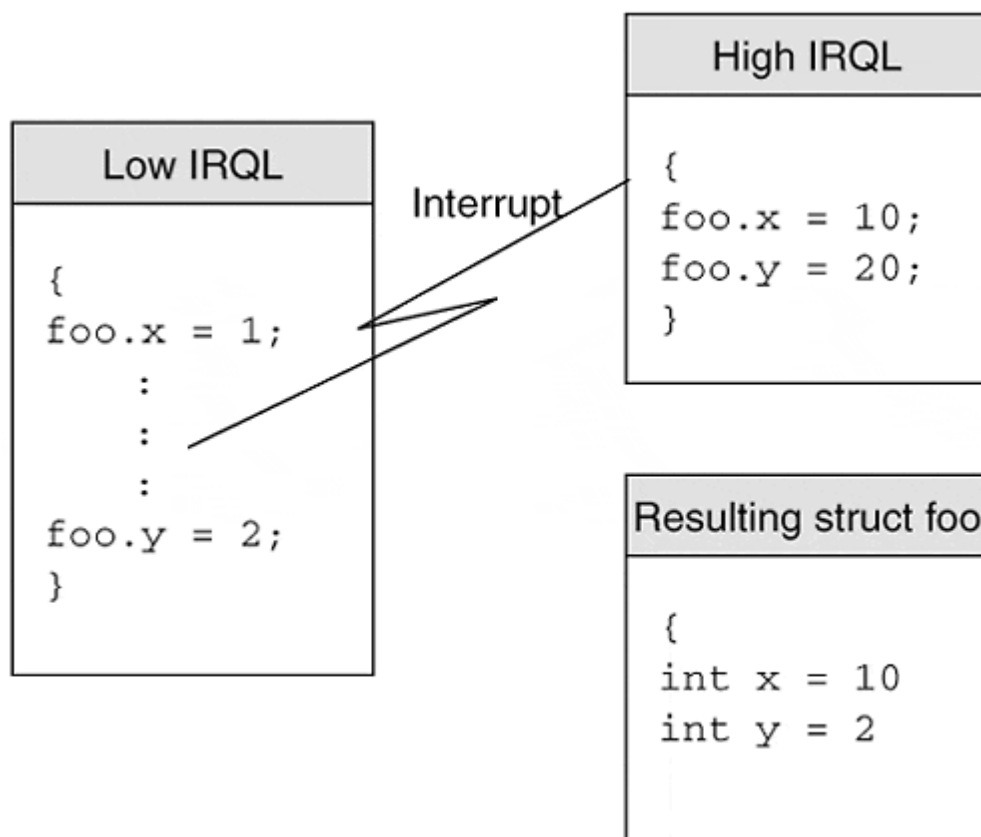
Interrupt Synchronization

Writing reentrant code that executes at multiple IRQ levels requires attention to proper synchronization. This section examines the issues that arise in this kind of environment.

The Problem

If code executing at two different IRQs attempts to access the same data structure simultaneously, the structure can become corrupted. [Figure 5.1](#) illustrates the details of this synchronization problem.

Figure 5.1. Synchronization problem with interrupting code.



To understand the exact problem, consider this sequence of events.

1.

Suppose code executing at a low IRQ decides to modify several fields in the **foo** data structure. For this example, the code executes to the point where the field **foo.x** is set to 1.

2.

An interrupt occurs and a higher-IRQ piece of code gets control of the processor. This code also modifies **foo**, setting **foo.x** to 10 and **foo.y** to 20.

3.

When the higher-IRQ code dismisses its interrupt and control returns to the lower-IRQ routine, the original modification of **foo** completes by setting **foo.y** to 2. The lower-IRQ code is completely unaware that it was interrupted.

4.

The **foo** structure is now inconsistent, with 10 in the **x** field and 2 in the **y** field. Neither section of code obtains the desired result.

Of course, similar results occur even when only a single shared value is used with different code threads. When one code path attempts to increment the variable, there may be a brief moment in time when the modified value is held in a CPU register. If this thread is interrupted prior to storing the register back into the variable, the same contention problem arises.

In the following sections, techniques that a driver can use to synchronize the work of multiple code paths is described.

Interrupt Blocking

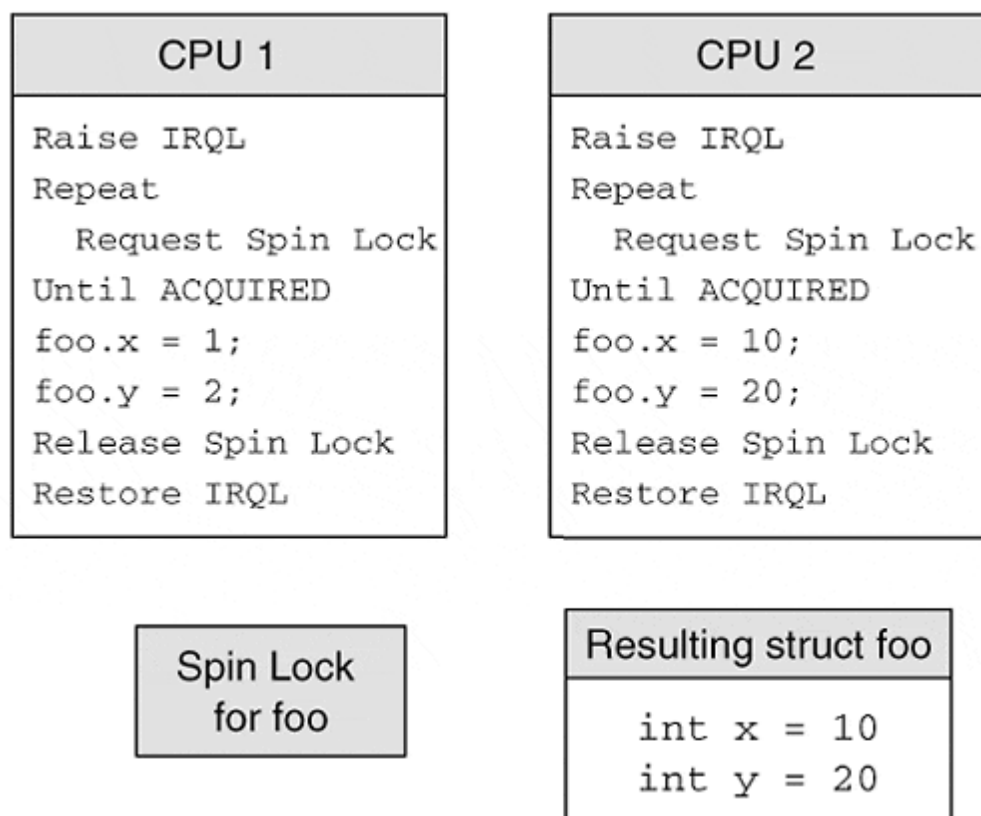
Synchronizing Multiple CPUs

Modifying the IRQL of one CPU has no effect on other CPUs and multiprocessor systems. Consequently, IRQLs provide only local CPU protection to share data. To prevent corruption of data structures in a multiprocessor environment, Windows 2000 uses synchronization objects called *spin locks*.

How Spin Locks Work

A spin lock is simply a mutual-exclusion object that is associated with a specific group of data structures. When a piece of kernel-mode code wants to touch any of the guarded data structures, it must first request ownership of the associated spin lock. Since only one CPU at a time can own the spin lock, the data structure is safe from contamination. Any CPU requesting an already-owned spin lock will busy-wait until the spin lock becomes available. [Figure 5.2](#) illustrates the process.

Figure 5.2. Spin locks.



A given spin lock is always acquired and released at a specific IRQL level. This has the effect of blocking dangerous interrupts on the local CPU and preventing the synchronization problems described earlier. While a CPU is waiting for a spin lock, all activity at or below the IRQL of the spin lock is blocked on that CPU. Both steps are essential—elevate the IRQL level *and* acquire the spin lock.

Using Spin Locks

There are two major kinds of spin locks provided by the kernel. They are distinguished by the IRQL level at which they are used.

-

Interrupt spin locks.

These synchronize and provide access to driver data structures shared by multiple-driver routines. Interrupt spin locks are acquired at the DIRQL associated with the device.

-

Executive spin locks.

These guard various operating system data structures, and their associated IRQL is DISPATCH_LEVEL.

When a driver uses Interrupt spin locks, operation is straightforward. The function **KeSynchronizeExecution** is described in [chapter 8](#).

Executive spin locks are more complicated to work with. The following steps must be followed when using Executive spin locks:

Linked Lists

Drivers sometimes need to maintain linked list data structures. This section describes the support available for managing singly and doubly linked lists.

Singly Linked Lists

To use singly linked lists, begin by declaring a list head of type **SINGLE_LIST_ENTRY**. This simplistic structure is also the data type of the linked pointer itself. Indeed, the **SINGLE_LIST_ENTRY** structure has but one offset: **Next**. The list head must be manually initialized, as demonstrated in the following code fragment.

```
typedef struct _DEVICE_EXTENSION {
    :
    SINGLE_LIST_ENTRY listHead;    // Declare head ptr
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

:
pDevExt->listHead.Next = NULL;    // init the list
```

To add or remove entries from the front of the list, call **PushEntryList** and **PopEntryList**. Depending on how the list is used, the actual entries can be in either page or nonpaged memory. Remember that the use of these functions may require synchronization.

The Windows 2000 kernel also provides convenient support for singly linked lists guarded by an Executive spin lock. This kind of protection is important when sharing a linked list among driver routines running at or below DISPATCH_LEVEL IRQL. To use one of these lists, set up the list head in the usual way, and then initialize an Executive spin lock that guards the list.

```
typedef struct _DEVICE_EXTENSION {
    :
    SINGLE_LIST_ENTRY listHead;    // head pointer
    KSPIN_LOCK listLock;          // declare list lock
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

:
KeInitializeSpinLock(&pDevExt->listLock);
pDevExt->listHead.Next = NULL;
```

Once the list head and spin lock are declared and initialized, the functions **ExInterlockedPushEntryList** and **ExInterlockedPopEntryList** provide convenient, protected access to the list. Code must be running at or below DISPATCH_LEVEL IRQL to use either function. The list entries themselves must reside in nonpaged memory, since the system will be linking and unlinking them from DISPATCH_LEVEL IRQL.

Doubly Linked Lists

To use doubly linked lists, declare a list head of type **LIST_ENTRY**. This is also the data type of the linked pointer itself. The **LIST_ENTRY** structure declares two offsets, **Flink** and **Blink**, for the forward and back pointers. The list head is initialized with a helper routine, **InitializeListHead**, as demonstrated in the following code fragment.

```
typedef struct _DEVICE_EXTENSION {
    :
    LIST_ENTRY listHead;    // head pointer
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

:
InitializeListHead( &pDevExt->listHead );
```

To add entries to the list, call **InsertHeadList** or **InsertTailList**, and to pull entries out, call **RemoveHeadList** or **RemoveTailList**. To determine if the list is empty, use **IsListEmpty**. Again, the entries can be paged or nonpaged, but these functions do not perform synchronization.

The Windows 2000 kernel also provides support for doubly linked lists. To use these lists, the list head must be manually initialized, as demonstrated in the following code fragment.

Summary

In this chapter some general guidelines for designing and coding a Windows 2000 device driver were covered. The prerequisites to coding the initial Windows 2000 device driver are now in place. In the next chapter, an actual device driver is implemented.

Chapter 6. Initialization and Cleanup Routines

CHAPTER OBJECTIVES

- [Writing a DriverEntry Routine](#)
- [Code Example: Driver Initialization](#)
- [Writing Reinitialize Routines](#)
- [Writing an Unload Routine](#)
- [Code Example: Driver Unload](#)
- [Writing Shutdown Routines](#)
- [Testing the Driver](#)
- [Summary](#)

*Everything has to start somewhere. In the case of a Windows 2000 kernel-mode driver, the starting point is a function called **DriverEntry**. This chapter shows how to write a **DriverEntry** routine along with various other pieces of initialization and cleanup code. At the conclusion of this chapter, a minimal driver is produced that can actually load into the system.*

Writing a DriverEntry Routine

Every Windows 2000 kernel-mode or WDM driver, regardless of its purpose, has to expose a routine whose name is **DriverEntry**. This routine initializes various driver data structures and prepares the environment for all the other driver components.

Execution Context

The I/O Manager calls a DriverEntry routine once it loads the driver. As [Table 6.1](#) shows, the DriverEntry routine runs at PASSIVE_LEVEL IRQL, which means it has access to page system resources.

The DriverEntry routine receives a pointer to its own driver object, which it must initialize. It also receives a UNICODE_STRING containing the path to the driver's service key in the Registry. WDM drivers have little use for this registry name. Kernel-mode drivers rely on the string to extract any driver-specific parameters stored in the system registry. The Registry String takes the form HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ DriverName.

What a DriverEntry Routine Does

Although the exact details vary based on whether a driver is WDM or kernel-mode, in general the following steps are taken in a DriverEntry routine.

1. DriverEntry locates hardware that it will be controlling. That hardware is *allocated*—it is marked as under the control of this driver.
2. The driver object is initialized by *announcing* other driver entry points. The announcements are accomplished by storing function pointers directly into the driver object.
3. If the driver manages a multiunit or multifunction controller, **IoCreateController** is used to create a controller object. A controller extension is then initialized.
4. **IoCreateDevice** is used to create a device object for each physical or logical device under the control of this driver. A device extension is then initialized.
5. The created device is made visible to the Win32 subsystem by calling **IoCreateSymbolicLink**.
6. The device is connected to an interrupt object. If the ISR requires the use of a DPC object, it is created and initialized in this step.
7. Steps 4 to 6 are repeated for each physical or logical device controlled by this driver.
8. If successful, DriverEntry should return STATUS_SUCCESS to the I/O Manager.

Table 6.1. Function Prototype for DriverEntry

NTSTATUS DriverEntry	IRQL = PASSIVE_LEVEL
Parameter	Description
IN PDRIVER_OBJECT pDriverObject	Address of driver object for this driver
IN PUNICODE_STRING pRegistryPath	Registry path string for this driver's key
Return value	• STATUS_SUCCESS
	• STATUS_XXX - some error code

It is important to understand that steps 1 and 3 to 6 are not performed by a WDM driver's DriverEntry routine. They are deferred to another routine, AddDevice.

If DriverEntry should fail during initialization for any reason, it should release any system resources it may have allocated up to the failure, and return an appropriate NTSTATUS failure code to the I/O Manager.

Code Example: Driver Initialization

The following example shows how a basic kernel-mode device driver initializes itself. The code for this example is in [chapter 6](#) directory on the disk that accompanies this book.

This first minimal driver must be manually loaded. It does not touch any hardware, but instead creates an internal device name (MINIMAL0) and a symbolic link name (MIN1). It consists of a single source module, Driver.cpp. A header file, Driver.h, declares driver-specific information about our nonhardware device, such as the DEVICE_EXTENSION.

DRIVERENTRY

In our first non-WDM driver example, the DriverEntry routine is small and straightforward. The responsibilities include

1. Announcing other DriverEntry points. For the Minimal driver, the only other routine to announce is the driver's Unload function.
2. Creating the logical devices that will be managed by the Minimal driver. For initial testing purposes, only a single device will be created.

```
//++
// Function:  DriverEntry
//
// Description:
//     Initializes the driver, locating and claiming
//     hardware resources. Creates the kernel objects
//     needed to process I/O requests.
//
// Arguments:
//     pDriverObject - Passed from I/O Manager
//     pRegistryPath - UNICODE_STRING pointer to
//                   registry info (service key)
//                   for this driver
//
// Return value:
//     NTSTATUS signaling success or failure
//--
NTSTATUS DriverEntry (
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING pRegistryPath ) {
    ULONG ulDeviceNumber = 0;
    NTSTATUS status;

    // If this driver controlled real hardware,
    // code would be placed here to locate it.
    // Using IoReportResourceUsage, the ports,
    // IRQs, and DMA channels would be "marked"
    // as "in use" and under the control of this driver.
    // This minimal driver has no HW, so...

    // Announce other driver entry points
    pDriverObject->DriverUnload = DriverUnload;
    // Over time, the MajorFunction array will be filled

    // For each physical or logical device detected
    // that will be under this Driver's control,
    // a new Device object must be created.
    status =
        CreateDevice(pDriverObject, ulDeviceNumber);
    // This call would be repeated until
    // all devices are created. E.g.,
    // ulDeviceNumber++;
    // status =
```

Writing Reinitialize Routines

Intermediate-level drivers loading at system boot time may need to delay their initialization until one or more lower-level drivers have finished loading. If the loading of the drivers is under central control, the load sequence can be fixed by setting various Registry entries at installation. But if the drivers are supplied by different vendors, the load order may be indeterminate. In the latter case, the intermediate driver must register and implement a Reinitialize routine.

Execution Context

If the DriverEntry routine discovers that it can't finish its initialization because system bootstrapping hasn't yet gone far enough, it can announce a Reinitialize routine by calling **IoRegisterDriverReinitialization**. The I/O Manager calls the Reinitialize routine at some later point during the bootstrap. As described in [Table 6.3](#), the Reinitialize routine runs at PASSIVE_LEVEL IRQL, which means it has access to paged system resources. Reinitialize routines are useful only for drivers that load automatically at system boot.

What a Reinitialize Routine Does

The Reinitialize routine can perform any driver initialization that the DriverEntry routine was unable to complete. If the Reinitialize routine discovers that the environment still isn't suitable, it can call **IoRegisterDriverReinitialization** again to reregister itself.

Table 6.3. Function Prototype for Reinitialize Routine

VOID Reinitialize	IRQL = PASSIVE_LEVEL
Parameter	Description
IN PDRIVER_OBJECT pDriverObject	Pointer to driver object
IN PVOID Context	Context block specified at registration
IN ULONG Count	Zero-based count of reinitialization calls
Return value	(void)

Writing an Unload Routine

By default, once the driver is loaded, it remains in the system until a reboot occurs. To make a driver unloadable, an Unload routine is necessary. The Unload routine is announced during DriverEntry. The I/O Manager then calls this routine whenever the driver is manually or automatically unloaded.

Execution Context

The I/O Manager calls a driver's Unload routine just before removing the driver from memory. [Table 6.4](#) shows that the Unload routine runs at PASSIVE_LEVEL IRQL, which means it has access to paged system resources.

What an Unload Routine Does

Although the exact details vary from driver to driver, in general the following steps are performed in an Unload routine.

Table 6.4. Function Prototype for Unload Routine

VOID Unload	IRQL = PASSIVE_LEVEL
Parameter	Description
IN PDRIVER_OBJECT pDriverObject	Pointer to driver object for this driver
Return value	(void)

1.

For some kinds of hardware, the state of the device should be saved in the Registry. That way, the device can be restored to its last known state the next time DriverEntry executes. For example, an audio card driver might save the current volume setting of the card.

2.

If interrupts have been enabled for the device, the Unload routine must disable them and disconnect from its interrupt object. It is crucial that the device not generate any interrupt requests once the interrupt object is deleted.

3.

Hardware belonging to the driver must be deallocated.

4.

The symbolic link name must be removed from the Win32 namespace. This is accomplished using **IoDeleteSymbolicLink**.

5.

The device object itself must be removed using **IoDeleteDevice**.

6.

If managing multiunit controllers, repeat steps 4 and 5 for each device attached to the controller. Then remove the controller object itself, using **IoDeleteController**.

7.

Repeat steps 4 through 6 for all controllers and devices that belong to the driver.

8.

Deallocate any pool memory held by the driver.

It is important to note that for WDM drivers, the responsibilities of Unload are performed in the RemoveDevice routine.

One final note: A driver's Unload routine is not called at system shutdown time. Any special work required during system shutdown must be performed inside of a separate shutdown routine.

Code Example: Driver Unload

In the Minimal driver, an Unload routine is supplied that undoes the work of DriverEntry. Its work is straightforward, as it must delete each symbolic link and device object that has been created. To perform this work, the Unload routine relies on the fact that the driver object points to a linked list of device objects controlled by the driver.

The first device controlled by a driver is pointed to by the field **DeviceObject** within the driver object. Each device points to the next via the field **NextDevice**. When examining the Unload routine, remember that the Minimal **DEVICE_EXTENSION** structure maintains a back pointer to the parent device object.

```
//++
// Function:  DriverUnload
//
// Description:
//      Stops & Deletes devices controlled by this driver.
//      Stops interrupt processing (if any)
//      Releases kernel resources consumed by driver
//
// Arguments:
//      pDriverObject - Passed from I/O Manager
//
// Return value:
//      None
//--

VOID DriverUnload (
    IN PDRIVER_OBJECT pDriverObject ) {

    PDEVICE_OBJECT pNextObj;

    // Loop through each device controlled by Driver
    pNextObj = pDriverObject->DeviceObject;
    while (pNextObj != NULL) {
        // Dig out the Device Extension from the
        // Device Object
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
            pNextObj->DeviceExtension;
        // This will yield the symbolic link name
        UNICODE_STRING pLinkName =
            pDevExt->ustrSymLinkName;
        // ... which can now be deleted
        IoDeleteSymbolicLink(&pLinkName);
        // a little trickery...
        // we need to delete the device object, BUT
        // the Device object is pointed to by pNextObj
        // If we delete the device object first,
        // we can't traverse to the next Device in the list
        // Rather than create another pointer, we can
        // use the DeviceExtension's back pointer
        // to the device.
        // So, first update the next pointer...
        pNextObj = pNextObj->NextDevice;
        // then delete the device using the Extension
        IoDeleteDevice( pDevExt->pDevice );
    }
    // Finally, hardware that was allocated in DriverEntry
    // would be released here using
    // IoReportResourceUsage
}
```

Writing Shutdown Routines

If a driver has special processing to do before the operating system disappears, a driver should supply a Shutdown routine.

Execution Context

The I/O Manager calls a Shutdown routine during a system shutdown. As described in [Table 6.5](#), the Shutdown routine runs at PASSIVE_LEVEL IRQL, which means it has access to paged system resources.

Table 6.5. Function Prototype for Shutdown Routine

NTSTATUS Shutdown	IRQL = PASSIVE_LEVEL
Parameter	Description
IN PRDRIVE_OBJECT pDriverObject	Pointer to driver object for this driver
IN PIRP pIrp	Pointer to shutdown IRP
Return value	• STATUS_SUCCESS -or- • STATUS_XXX - some error code

What a Shutdown Routine Does

The main purpose of the Shutdown routine is to put the device into a quiescent state and perhaps store some device information into the system Registry. Again, saving the current volume settings from a sound card is a good example of something a Shutdown routine might do.

Unlike the driver's Unload routine, Shutdown routines don't have to worry about releasing driver resources because the operating system is about to disappear anyway.

Enabling Shutdown Notification

There is no direct field in the Driver object for announcing the Shutdown routine. Instead, the event of system shutdown is sent as a separate I/O request to a driver. It is handled with an entry inside of the driver's **MajorFunction** code array.

Additionally, the I/O Manager must be notified that a driver is interested in receiving shutdown notifications. This is done by making a call to **IoRegisterShutdownNotification**. The following code fragment shows how to enable shutdown notifications in a driver.

```
NTSTATUS DriverEntry (
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING pRegistryPath ) {
    :
    pDriverObject->MajorFunction[ IRP_MJ_SHUTDOWN ] =
        Shutdown;
    IoRegisterShutdownNotification( pDriverObject );
    :
}
```

Testing the Driver

Even though the Minimal driver is far from being useful, its operation can be verified in several ways. In particular, it can be tested to ensure that it

- - Compiles and links successfully
- - Loads and unloads without crashing the system
- - Creates device objects and Win32 symbolic links
- - Releases any resources when it unloads

While these goals may not seem very ambitious, they are an important milestone in beginning the construction of the rest of a truly useful driver.

Testing Procedure

While a description of the production-level installation of the device driver must be left until [chapter 16](#), a manual installation of the Minimal driver can be performed now. The disk accompanying this book includes a useful tool that makes this job much easier.

Visual C++ Device Driver AppWizard

In the Tools folder of the accompanying disk, there is a file called DDApWiz.awx. This file is the result of building a custom AppWizard project for Visual C++. When properly installed, it allows the creation of the new project that sets up the Visual C++ environment for Windows 2000 device driver development. For example, it modifies the linker settings to specify the -driver switch. It supplies the correct list of DDK libraries to the linker. It specifies the required load address of 0x10000. In short, it makes the construction of the Windows 2000 device driver project about as simple as starting an MFC application.

To install this AppWizard, it is necessary to copy DDApWiz.awx into the appropriate Visual Studio directory. The exact location varies based on the installation directory chosen for Visual Studio. Typically, the correct target directory for the AppWizard file is \Program Files\Microsoft Visual Studio\Common\MSDev98\Template. The correct directory can be confirmed by examining a file called Readme.txt in the target. Once the file is correctly copied, selecting File, then New, and then Project should reveal a new option for Windows 2000 device driver development.

Indeed, a Developer Studio Project and Workspace environment is supplied in the [chapter 6](#) folder of the disk. The Minimal.dsw workspace can be opened directly and the build attempted. The example code listed in this chapter is already included in this project.

The Windows 2000 DDK

The Windows 2000 Device Driver Kit must be installed on a development system in order to build device drivers. As mentioned in [chapter 1](#), the DDK is available for download from the *Microsoft.com* site. By default, it installs into a folder named NTDDK. If another folder name is chosen during installation of the DDK, the Project settings supplied by the device driver AppWizard must be modified.

The DDK allows for two different build environments, checked and free. The checked build environment is analogous to the Debug build environment of Visual C++, while the free build environment is analogous to the Release setting. As such, two different sets of libraries are supplied with the DDK, Libfre and Libchk. For the initial build of the Minimal project, it is suggested that a checked build be performed. This is easily done by selecting the Debug project target.

Results of the Driver Build

Once a driver is successfully built, a file ending with a .SYS extension is created. In the case of the Minimal driver, the file is called Minimal.SYS. This file must be copied to the appropriate Windows directory before it can be loaded. The target directory is typically \WINNT\System32\Drivers. The WINNT portion of the name varies based on the name selected during the Windows 2000 installation.

The project settings supplied by the AppWizard include a post-build step, which copies the driver to the appropriate directory. It is important to keep the results of each build in sync with the Drivers directory.

Summary

This completes the implementation of an initial Windows 2000 device driver. Granted, this Minimal driver is kernel-mode only. Converting it to a WDM driver must wait until [chapter 9](#). Nevertheless, tools to build and install a driver are now ready to go. Additionally, the beginning structure of the kernel-mode device driver is forming.

The next chapter discusses the Dispatch routines necessary to make the driver useful. Read, Write, and I/O Control functions are implemented. Coding shown in the next chapter builds upon the minimal driver built in this chapter.

Chapter 7. Driver Dispatch Routines

CHAPTER OBJECTIVES

- [Announcing Driver Dispatch Routines](#)
- [Writing Driver Dispatch Routines](#)
- [Processing Read and Write Requests](#)
- [Code Example: A Loopback Device](#)
- [Extending the Dispatch Interface](#)
- [Testing Driver Dispatch Routines](#)
- [Summary](#)

Getting a driver to load is the first step, but ultimately the job of a driver is to respond to I/O requests—from user-mode applications or from other parts of the system. Windows 2000 drivers process these requests by implementing Dispatch routines. The I/O Manager in response to a request calls these routines. This chapter describes the basic Dispatch routines and explains the overall process of handling I/O requests.

Announcing Driver Dispatch Routines

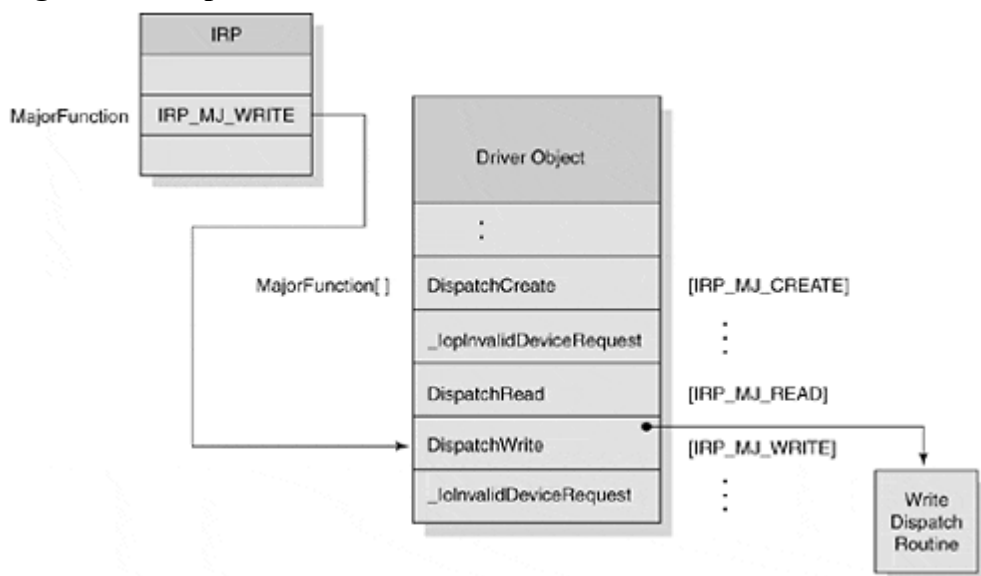
Before a driver can process I/O requests, it must announce what kinds of operations it supports. This section describes the I/O Manager's dispatching mechanism and explains how to enable receipt of appropriate I/O function codes. It also presents guidelines for deciding which function codes should be supported for different device types.

I/O Request Dispatching Mechanism

Windows 2000 I/O operations are packet-driven. When an I/O request is initiated, the I/O Manager first builds an IRP work-order to keep track of the request. Among other things, it stores a function code in the **MajorField** field of the IRP's I/O stack location to uniquely identify the type of request.

The **MajorField** code is used by the I/O Manager to index the Driver object's **MajorFunction** table. The table contains a function pointer to a Dispatch routine specific to the I/O request. If a driver does not support the requested operation, the **MajorFunction** table entry points to an entry within the I/O Manager, **_IoInvalidDeviceRequest**, which returns an error to the original caller. Thus, it is the responsibility of the driver author to provide Dispatch routines for each I/O function code that it supports. [Figure 7.1](#) illustrates this process.

Figure 7.1. Dispatch routine selection.



Enabling Specific Function Codes

To enable specific I/O function codes, a driver must first "announce" the Dispatch routine that responds to such a request. The announcement mechanism is simply the work performed by DriverEntry that stores the Dispatch routine function address into the appropriate slot of the **MajorFunction** table of the driver object. The I/O function code is the index used for the table. The following code fragment illustrates the process of announcement.

```

NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDO,
                    IN PUNICODE_STRING pRegPath ) {
    :
    pDO->MajorFunction[ IRP_MJ_CREATE ] = DispCreate;
    pDO->MajorFunction[ IRP_MJ_CLOSE ] = DispClose;
    pDO->MajorFunction[ IRP_MJ_CLEANUP ] = DispCleanup;
    pDO->MajorFunction[ IRP_MJ_READ ] = DispRead;
    pDO->MajorFunction[ IRP_MJ_WRITE ] = DispWrite;
    :
    return STATUS_SUCCESS;
}
  
```

Notice that each I/O function code (table index) is identified by a unique symbol of the form **IRP_MJ_XXX**, defined by the NTDDK.h (and WDM.h) include file. Of course, these symbolic constants should always be used in lieu of hard-coded constants.

The announcement technique also allows for a single routine to be used to handle multiple request types. DriverEntry would place the same Dispatch routine address in multiple slots. Since the IRP's MajorField contains the requested code, the

Writing Driver Dispatch Routines

Depending on the complexity of the device operation and the kind of I/O request, driver Dispatch routines range from trivial to quite difficult to implement. This section explains how to code these routines.

Execution Context

All Dispatch routines share the same function signature. (A function signature includes the number and type of parameters, and its calling convention.) [Table 7.2](#) shows the prototype for all Dispatch routines. Like the driver's initialization and unload routines, Dispatch routines run at `PASSIVE_LEVEL` IRQL, which means they can access paged system resources.

The I/O Manager invokes Dispatch routines in response to user-mode or kernel-mode requests. Before calling, the I/O Manager builds and fills the IRP with valid data. This includes the pointer to the user's buffer. The user buffer is validated by the I/O Manager to ensure that each page address spanned by the buffer is readable or writeable within the context of the requestor. If the request is for Buffered I/O, the I/O Manager first allocates a nonpaged pool buffer and, if a write request, copies data from the user buffer into the pool. If the request is for Direct I/O, the I/O Manager faults the entire user buffer into physical memory and locks it down.

A Dispatch routine can usually track the state of an I/O request using only the IRP. If a Dispatch routine uses any data structures outside the IRP, the driver must ensure that proper synchronization steps are taken. This would mean using a spin lock to coordinate with other driver routines running at `DISPATCH_LEVEL` or below IRQL, and **KeSynchronizeExecution** to synchronize with Interrupt Service code.

The IRP is shared data, albeit serially, with the I/O Manager. In particular, the I/O Manager uses fields of the **Parameters** union to complete the I/O request. For example, after a Buffered I/O request, it needs to copy data from the nonpaged pool into the user buffer. It must then deallocate the pool buffer. A field within the **Parameters** union points to this buffer. Therefore, changing the value of this buffer pointer would lead to disastrous results.

In general, if a Dispatch routine needs to modify an IRP field, it should make working copies on the stack or in the device extension.

Table 7.2. Function Prototype for Dispatch Routines

NTSTATUS Dispatch	IRQL==PASSIVE_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pDevObject	Pointer to target device for this request
IN PRIP pIrp	Pointer to IRP describing this request
Return value	•STATUS_SUCCESS - request complete
	•STATUS_PENDING - request pending
	•STATUS_XXX - appropriate error code

What Dispatch Routines Do

The exact behavior of a Dispatch routine will depend on the function it supports. However, the general responsibilities of these routines include the following.

1. Call **IoGetCurrentIrpStackLocation** to get a pointer to the IRP stack location belonging to this driver.
2. Perform any additional parameter validation specific to this request and device.
3. For an intermediate-level driver, consideration must be given to the limitations of the underlying physical device (for example, its maximum transfer size). The Dispatch routine may need to split the caller's request into multiple requests for the lower-level driver.
4. Continue processing the IRP until the request is complete or an error condition prevents further processing.

Exiting the Dispatch Routine

When a Dispatch routine processes an IRP, there are only three possible outcomes.

-

Processing Read and Write Requests

The most basic of I/O requests is to exchange data between a user buffer and a device. The I/O Manager presents a traditional read/write abstraction to requestors for such data transfers. The requests are presented to a driver in the form of an IRP with major function code of either IRP_MJ_READ or IRP_MJ_WRITE. Another field within the IRP specifies the address of the requestor's buffer. Whether the buffer address is a direct virtual address or an intermediate nonpaged pool buffer allocated and maintained by the I/O Manager is determined by the device object's **Flags** field. Regardless, it is the responsibility of the read and write Dispatch routines to transfer data between the buffer and the actual device for the requested number of bytes.

User Buffer Access

As discussed in chapter 6, a driver can specify DO_DIRECT_IO or DO_BUFFERED_IO on a per-device basis (via the **Flags** field of the device object.) The exact behavior of the I/O Manager as well as the subsequent responsibilities of the Dispatch routine is discussed in the following sections.

BUFFERED I/O

At the start of either a read or write operation, the I/O Manager validates that all virtual memory pages spanned by the user's buffer are valid. For buffered I/O, it then allocates a nonpaged pool buffer of a size sufficient to hold the user's request. The address of this temporary buffer is placed in the IRP field **AssociatedIrp.SystemBuffer**. This address remains valid for the duration of the transfer (i.e., until the IRP is marked as complete).

For read operations, the I/O Manager remembers the address of the original user buffer in the **UserBuffer** field of the IRP. It then uses this retained address upon completion of the request to copy data from the nonpaged pool into user memory.

For write operations, the I/O Manager copies the user buffer into the nonpaged buffer before invoking the write Dispatch routine. It then sets the **UserBuffer** field of the IRP to NULL, since there is no additional need to retain this state.

DIRECT I/O

At the start of the operation, the I/O Manager validates the page table entries spanned by the user's buffer. It then builds a data structure known as a *Memory Descriptor List* (MDL) and stores the address of the MDL in the IRP's **MdlAddress** field. Both the **AssociatedIrp.SystemBuffer** and **UserBuffer** fields are set to NULL.

For DMA operations, the MDL structure is used directly with an adapter object to perform the data transfer. [Chapter 12](#) will discuss this process in detail. For programmed I/O devices, the MDL can be used with the function **MmGetSystemAddressForMdl** to get a system-address view of the user buffer. Using this technique, the user's buffer is locked down into physical memory (i.e., forced to be nonpagable) and is made accessible to driver code via an address above 0x80000000. (User-mode code must still access the same physical memory with its original address below 0x7FFFFFFF.) When the I/O request is ultimately completed, the user buffer is unlocked and unmapped from system address space.

NEITHER METHOD

There are two bits within the **Flags** field of the device object which specify either DO_BUFFERED_IO or DO_DIRECT_IO. If neither bit is set for a device, the I/O Manager performs neither action specified above. Instead, it simply places the user-space address of the requestor's buffer into the IRP's **UserBuffer** field. The **AssociatedIrp.SystemBuffer** and **MdlAddress** fields are set to NULL.

A simple user-mode address is not terribly useful. Most routines within a driver cannot be assured that at the time of their execution the original requestor's page tables are mapped. Thus, the user buffer address is usually worthless. There is one exception: At the time a Dispatch routine of a highest-level driver is invoked, execution occurs using the original requestor's thread. As such, the user-space address is mapped and valid. An intermediate driver or any DPC or Interrupt Service Routine (ISR) can never rely upon a user-space buffer being valid.

Code Example: A Loopback Device

An interesting (albeit simplistic) example to consider is a loopback device. The driver processes write requests by reserving a paged pool buffer and copying user data into this temporary buffer. The buffer is retained until such time as a read request is issued. The data returned by the read request is the contents of the temporary buffer, which is then released. The example demonstrates the implementation of read and write Dispatch routines as well as user buffer access.

```

NTSTATUS DispatchWrite(  IN PDEVICE_OBJECT pDO
                        IN PIRP pIrp) {
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pDE;
    PVOID userBuffer;
    ULONG xferSize;
    // The stack location contains the user buffer info
    PIO_STACK_LOCATION pIrpStack;
    pIrpStack = IoGetCurrentIrpStackLocation( pIrp );
    // The example assumes the device is using BUFFERED_IO
    userBuffer = pIrp->AssociatedIrp.SystemBuffer;
    xferSize = pIrpStack->Parameters.Write.Length;
    // The temporary buffer pointer is kept
    // in the DEVICE_EXTENSION (obtained from Device obj)
    pDE = (PDEVICE_EXTENSION) pDO -> DeviceExtension;
    // If there is already a buffer, free it...
    if (pDE->deviceBuffer != NULL) {
        ExFreePool( pDE->deviceBuffer );
        pDE->deviceBuffer = NULL;
        xferSize = 0;
    }
    pDE->deviceBuffer =
        ExAllocatePool(PagedPool, xferSize);
    if (pDE->deviceBuffer == NULL) {
        // buffer didn't allocate???
        status = STATUS_INSUFFICIENT_RESOURCES;
        xferSize = 0;
    } else {
        // copy the buffer
        pDE->deviceBufferSize = xferSize;
        RtlCopyMemory( pDE->deviceBuffer, userBuffer,
                       xferSize );
    }
    // Now complete the IRP no device operation needed
    pIrp->IoStatus.Status = status;
    pIrp->IoStatus.Information = xferSize;
    IoCompleteRequest( pIrp, IO_NO_INCREMENT );
    return status;
}

NTSTATUS DispatchRead(  IN PDEVICE_OBJECT pDO,
                      IN PIRP pIrp ) {
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pDE;
    PVOID userBuffer;
    ULONG xferSize;
    // The stack location contains the user buffer info
    PIO_STACK_LOCATION pIrpStack;
    pIrpStack = IoGetCurrentIrpStackLocation( pIrp );
    userBuffer = pIrp->AssociatedIrp.SystemBuffer;
    xferSize = pIrpStack->Parameters.Read.Length;
    // The temporary buffer pointer is kept
    // in the DEVICE_EXTENSION (obtained from Device obj)
    pDE = (PDEVICE_EXTENSION) pDO -> DeviceExtension;
    // Don't transfer more than the user's request
    xferSize = (xferSize < pDE->deviceBufferSize) ?
        xferSize : pDE->deviceBufferSize;
    // Now copy the temporary buffer into user space

```

Extending the Dispatch Interface

Much of the I/O Manager operation supports a standard Read/Write abstraction. The requestor supplies a buffer and transfer length, and data is transferred from or to the device. Not all devices or their operations always fit this abstraction. For example, a disk format or repartition are requests that are not well-suited to a normal Read or Write operation. Such kinds of requests are handled with one of two extensible I/O function request codes. These codes allow any number of driver-specific operations, without the restrictions of the Read/Write abstraction.

- **IRP_MJ_DEVICE_CONTROL** allows for extended requests from user-mode clients through the Win32 **DeviceIoControl** call. The I/O Manager constructs an IRP with this **MajorFunction** code and an **IoControlCode** value (subcode) as specified by the caller of **DeviceIoControl**.

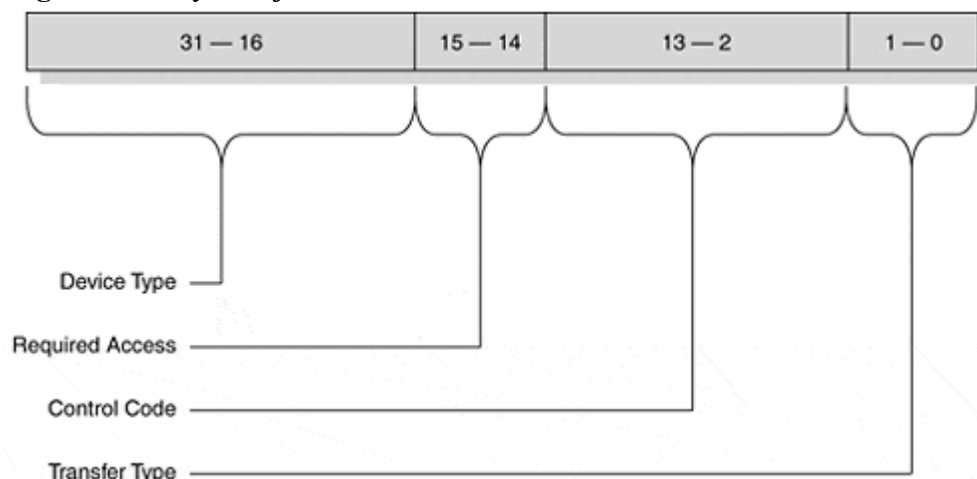
- **IRP_MJ_INTERNAL_DEVICE_CONTROL** allows for extended requests from kernel-mode code. No access to these operations is provided for user-mode requests. This facility is primarily used by other drivers in a layered stack to communicate special requests. Otherwise, the internal version of the device control operation is identical to the standard version. An **IoControlCode** value is placed into the IRP by the requestor.

As should be apparent, the implementation of either of these Dispatch routines requires a secondary dispatch based on the value of **IoControlCode** in the IRP. This value is also known as the **IOCTL** device control code. Since the secondary dispatch mechanism is completely contained within the driver's private routine(s), the interpretation of the **IOCTL** value is driver-specific. The remainder of this section describes the details of the device control interface.

Defining Private IOCTL Values

The IOCTL values passed to a driver follow a specific structure. [Figure 7.2](#) illustrates the fields of this 32-bit structure. The DDK includes a macro, **CTL_CODE**, that offers a convenient mechanism to generate IOCTL values. [Table 7.3](#) describes the arguments to this macro.

Figure 7.2. Layout of the IOCTL code structure.



IOCTL Argument-Passing Method

The extended functions defined with an IOCTL value within a driver often require an input or output buffer. For example, a driver might report performance data using an IOCTL value. The data reported would be transferred through a buffer supplied by the user. Indeed, the Win32 **DeviceIoControl** function defines parameters for two buffers, one for input, one for output. The buffer transfer mechanism provided by the I/O Manager is defined within the IOCTL value itself. It can be either buffered or direct I/O. As described previously, with buffered I/O, the I/O Manager copies the user buffer (into or out of) nonpaged system memory, where driver code can then conveniently operate. With direct I/O, the driver is given direct access to the user buffer.

Interestingly, the driver's overall strategy for buffer handling (defined during **DriverEntry**) is not enforced for IOCTL transfers. Instead, the buffer transfer mechanism is defined with each IOCTL value specification and is a field within the IOCTL structure. This provides maximum flexibility when performing **DeviceIoControl** operations.

Table 7.3. The CTL_CODE Macro Arguments

CTL_CODE Macro	
Parameter	Description
DeviceType	FILE_DEVICE_XXX value supplied to IoCreateDevice

Testing Driver Dispatch Routines

Even though a driver may be far from complete when only the DriverEntry, Unload, and Dispatch routines are present, significant code paths can be tested at this point. In particular, all of the following can be verified with a simple Win32 console program:

- - Opens and closes a handle to the device.
- - Supports Win32 I/O function calls that return successfully, even if zero byte transfers are specified.
- - Manages requests from multiple callers.

While the successful completion of these tests is hardly earth-shattering, it does form a tried-and-true recipe for driver authoring: Build a driver framework that is proven *before* adding hardware interaction.

Testing Procedure

The following procedure checks all the code paths through a driver's Dispatch routines:

1.
 - Write IRP_MJ_CREATE and IRP_MJ_CLOSE Dispatch routines for the driver.
2.
 - Test the driver with a simple Win32 console program that gets a handle to the device under test and then closes the handle.
3.
 - Write other Dispatch routines but ensure that all call **IoCompleteRequest** rather than starting any device operations.
4.
 - Enhance the Win32 test program to make **ReadFile**, **WriteFile**, and **DeviceIoControl** calls that exercise each driver's Dispatch routine. Alternatively, the test program can request zero-byte transfers.
5.
 - If a device is sharable, run several copies of the test program at once to be sure the driver works with multiple open handles.
6.
 - If a driver supports multiple physical devices, repeat the test with each device unit.

Sample Test Program

This is an example of a Win32 console test program that can be used to verify code paths through a driver's Dispatch routines.

```
#include <windows.h>
#include <stdio.h>

void main() {
    HANDLE hDevice;
    BOOL status;
    hDevice = CreateFile( "\\.\LbK1" ... );
    :
    status = ReadFile( hDevice, ... );
    :
    status = WriteFile( hDevice, ... );
    :
    status = DeviceIoControl( hDevice, ... );
    :
    status = CloseHandle( hDevice );
}
```

Summary

Dispatch routines form the basic interface between a requestor and a driver. This chapter presented the framework for these functions and discussed the details for accessing the user's buffers and other parameters. Read, Write, and DeviceIoControl requests can now be presented to the driver.

The next chapter begins the interface with the real device. The path from starting the device to register data transfer to interrupt handling is discussed.

Chapter 8. Interrupt-Driven I/O

CHAPTER OBJECTIVES

- [How Programmed I/O Works](#)
- [Driver Initialization and Cleanup](#)
- [Writing a Start I/O Routine](#)
- [Writing an Interrupt Service Routine \(ISR\)](#)
- [Writing a DpcForIsr Routine](#)
- [The Parallel Port](#)
- [Parallel Port Loopback Driver](#)
- [Testing the Parallel Port Loopback Driver](#)
- [Summary](#)

Some devices transfer data through registers, requiring continual CPU interaction. Such devices usually move small amounts of data at relatively infrequent intervals. For example, the mouse and keyboard transfer just a few bytes sporadically. Such devices remain idle for minutes (or hours) at a time and so are given the ability to interrupt the processor whenever data is available. This chapter explains how to write the data transfer routines for drivers of this kind of hardware.

How Programmed I/O Works

This section describes the events that occur during a programmed I/O operation as well as the actions that a driver must take in response to these events.

What Happens During Programmed I/O

In a programmed I/O operation, the CPU transfers each unit of data to or from the device in response to an interrupt. The following sequence of events occurs.

1.

An IRP request (typically an IRP_MJ_READ or IRP_MJ_WRITE) determines that device interaction is required to complete the request. The Dispatch routine queues the IRP for eventual delivery to the driver's Start I/O routine.

2.

The Start I/O routine performs any necessary preprocessing and setup based on the IRP's function code. It then starts the device, typically by writing or reading the first word of device data.

3.

Eventually, the device generates an interrupt, which the kernel passes to the driver's Interrupt Service Routine (ISR).

4.

If there is additional data for the driver to download, the ISR starts the next transfer. Steps 3 and 4 repeat until the entire data transfer specified in the request is complete.

5.

When the entire transfer is complete, the ISR queues a request to fire the driver's DpcForIsr routine. As described in chapter 3, DPC routines run at a lower IRQL level than the ISR.

6.

The I/O Manager's DPC dispatcher eventually runs the DpcForIsr routine scheduled by the ISR. The DpcForIsr routine marks the IRP as complete and informs the I/O Manager that another waiting IRP can now be presented to the Start I/O routine, repeating the entire cycle.

Synchronizing Driver Routines

As is apparent from the programmed I/O process just described, there are at least four separate code paths executing at three different IRQL levels.

•

The original I/O request is handled by a Dispatch routine running at PASSIVE_LEVEL IRQL.

•

The driver's Start I/O routine runs at DISPATCH_LEVEL IRQL.

•

The driver's ISR runs at the device's DIRQL level.

•

The driver's DpcForIsr routine runs at DISPATCH_LEVEL IRQL.

If any of these code paths share registers or memory areas, synchronization is required. Without such protection, an interrupt might arrive while a lower-level IRQL routine is using the shared resource, and an inconsistent state could result. The solution to this problem is to place code that touches shared resources in a SynchCriticalSection routine. [Table 8.1](#) shows the prototype for one of these routines.

This technique for synchronization requires that *the* code which touches the shared resource be placed in an appropriate SynchCriticalSection routine. When the shared resource is required, a code path invokes **KeSynchronizeExecution** (see [Table 8.2](#)), which takes the function address of the SynchCriticalSection as an argument. This kernel function raises the IRQL level of the calling code path to the DIRQL level of the device's interrupt object, acquires the object's spin lock, and then calls the supplied SynchCriticalSection routine.

Table 8.1. Function Prototype for a SynchCriticalSection Routine

BOOLEAN SynchCriticalSection	IRQL = DIRQL
------------------------------	--------------

Driver Initialization and Cleanup

Programmed I/O device drivers must perform extra initialization during DriverEntry (or AddDevice for WDM drivers). Similarly, the driver's Unload routine must be extended to remove the additional resources allocated.

Initializing the Start I/O Entry Point

If a driver has a Start I/O routine, it must be announced during DriverEntry. This is done by storing the address of the Start I/O routine into the **DriverStartIo** field of the driver object, as in the following code fragment.

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject,
                   IN PUNICODE_STRING pRegistryPath ) {
    :
    // Export other driver entry points
    //
    pDriverObject->DriverStartIo = StartIo;
    pDriverObject->DriverUnload = DriverUnload;

    pDriverObject->MajorFunction[ IRP_MJ_CREATE ] =
        DispatchOpenClose;
    :
}
```

An uninitialized **DriverStartIo** field within the driver object results in an access violation (and blue screen crash) when the Dispatch routines invoke **IoStartPacket**.

Initializing a DpcForIsr Routine

The I/O Manager provides a simplified version of the DPC mechanism for use with standard interrupt processing. One special DPC object may be associated with each device object, DpcForIsr. To utilize this mechanism, a driver must call **IoInitializeDpcRequest** to associate the DpcForIsr routine with the Device object (typically from DriverEntry or AddDevice). During the actual interrupt service (ISR), the driver schedules the DPC by invoking **IoRequestDpc**.

Of course, this simplified mechanism is quite restrictive. Some drivers require multiple DPCs for different circumstances. A subsequent chapter explains the process of creating custom DPCs for multiple purposes.

Connecting to an Interrupt Source

All interrupts are initially handled by the kernel of Windows 2000. As explained in chapter 1, this is done so that portability to multiple platforms can be easily achieved. The kernel dispatches interrupts to a driver's ISR by creation of and then connection to an interrupt object. These steps are accomplished by the I/O Manager's **IoConnectInterrupt** function, described in [Table 8.3](#). The driver's ISR address is passed (along with nine other input parameters) to this function so that the kernel associates a specific hardware interrupt with it.

IoConnectInterrupt returns a pointer to an interrupt object (via the first argument). This pointer should be saved in the Device or Controller Extension since it will be needed to ultimately disconnect from the interrupt source or to execute any SynchCriticalSection routines.

Table 8.3. Function Prototype for IoConnectInterrupt

NTSTATUS IoConnectInterrupt	IRQL = PASSIVE_LEVEL
Parameter	Description
OUT PKINTERRUPT *pInterruptObject	Address of pointer to receive pointer to interrupt object
IN PKSERVICE_ROUTINE ServiceRoutine	ISR that handles the interrupt
IN PVOID pServiceContext	Context argument passed to ISR; usually the device extension
IN PKSPIN_LOCK pSpinLock	Initialized spin lock (see below)
IN ULONG Vector	Translated interrupt vector value
IN KIRQL Irql	DIRQL value for device
IN KIRQL SynchronizeIrql	Usually same as DIRQL (see below)
IN KINTERRUPT_MODE InterruptMode	• LevelSensitive
	• Latched

Writing a Start I/O Routine

The remainder of this chapter discusses the development of a programmed I/O driver for a parallel port. For simplicity, this driver ignores many of the details that would have to be considered if writing a commercial driver.

Execution Context

The I/O manager calls the driver's Start I/O routine (described in Table 8.4) either when a Dispatch routine calls **IoStartPacket** (if the device was idle), or when some other part of the driver calls **IoStartNextPacket**. In either case, Start I/O runs at DISPATCH_LEVEL IRQL, so it must not do anything that causes a page fault.

What the Start I/O Routine Does

A driver's Start I/O routine is responsible for doing any function code-specific processing needed by the current IRP and then starting the actual device operation. In general terms, a Start I/O routine will do the following:

Table 8.4. Function Prototype for a Start I/O Routine

VOID StartIo	IRQL = DISPATCH_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pDevObj	Target device for this request
IN PIRP pIrq	IRP describing this request
Return value	Void

1. Call **IoGetCurrentStackLocation** to get a pointer to the IRP's stack location.
2. If a device supports more than one IRP_MJ_XXX function code, examine the I/O stack location's **MajorFunction** field to determine the operation.
3. Make working copies of the system buffer pointer and byte counts stored in the IRP. The Device Extension is the best place to keep these items.
4. Set a flag in the Device Extension, indicating that an interrupt is expected.
5. Begin the actual device operation.

To guarantee proper synchronization, any of these steps that access data shared with the ISR should be performed inside the SynchCriticalSection routine rather than in Start I/O itself.

Writing an Interrupt Service Routine (ISR)

Once the device operation begins, the actual data transfer is driven by the arrival of hardware interrupts. When an interrupt arrives, the driver's Interrupt Service routine acknowledges the request and either transfers the next piece of data or invokes a DPC routine.

Execution Context

When the kernel receives a device interrupt, it uses its collection of interrupt objects to locate an ISR willing to service the event. It does this by running through all the interrupt objects attached to the DIRQL of the interrupt and calling ISRs until one of them claims the interrupt.

The kernel interrupt dispatcher calls an ISR at the synchronization IRQL specified in the call to **IoConnectInterrupt**. Usually this is the DIRQL level of the device. The kernel dispatcher also acquires and releases the device spin lock.

Running at such a high IRQL, there are several things an ISR isn't allowed to do. In addition to the usual warning about page faults, an ISR shouldn't try to allocate or free various system resources (like memory). If system support routines must be called from an ISR, check for restrictions on the level of which they can be run. Such calls might require delegation to a DPC routine.

As shown in [Table 8.5](#), the kernel passes a pointer to whatever context information was identified in the original call to **IoConnectInterrupt**. Most often, this is a pointer to the Device or Controller Extension.

What the Interrupt Service Routine Does

The Interrupt Service routine is the real workhorse in a programmed I/O driver. In general, one of these routines does the following:

1. Determine if the interrupt belongs to this driver. If not, immediately return a value of FALSE.
2. Perform any operations needed by the device to acknowledge the interrupt.
3. Determine if any more data remains to be transferred. If so, start the next device operation. This eventually results in another interrupt.
4. If all the data has been transferred (or if a device error occurred), queue up a DPC request by calling **IoRequestDpc**.
5. Return a value of TRUE.

Always code an ISR for speed. Any work that isn't absolutely essential should go in a DPC routine. It is especially important that an ISR determine whether or not it will handle an interrupt immediately. There may be a number of other ISRs waiting in line for a given interrupt, and nonessential preprocessing blocks their proper operation.

Table 8.5. Function Prototype for an Interrupt Service Routine

BOOLEAN ISR	IRQL = DIRQL
Parameter	Description
IN PKINTERRUPT pInterruptObj	Interrupt object generating interrupt
IN PVOID pServiceContext	Context area passed to IoConnectInterrupt
Return value	•TRUE-interrupt was serviced by ISR
	•FALSE-interrupt not serviced

Writing a DpcForIsr Routine

A driver's DpcForIsr routine is responsible for determining a final status for the current request, completing the IRP, and starting the next one.

Execution Context

In response to the ISR's call to **IoRequestDpc**, a driver's DpcForIsr routine (described in [Table 8.6](#)) is added to the DPC dispatch queue. When the CPU's IRQL value drops below DISPATCH_LEVEL, the DPC dispatcher calls the DpcForIsr routine. A DpcForIsr routine runs at DISPATCH_LEVEL IRQL, which means it has no access to pageable addresses.

The I/O Manager ignores multiple calls to **IoRequestDpc** for a given device until the DpcForIsr routine executes. This is normal behavior for all DPC objects. If a driver design is such that it might issue overlapping DPC requests for the same device, then the driver must perform custom queuing of the DPC requests.

What the DpcForIsr Routine Does

Since most of the work happens during interrupt processing, the DpcForIsr routine in a programmed I/O driver doesn't have a lot to do. In particular, this routine should

1. Set the IRP's I/O status block. Put an appropriate STATUS_XXX code in the **Status** field and the actual number of bytes transferred in the **Information** field.
2. Call **IoCompleteRequest** to complete the IRP with an appropriate priority boost. Once called, the IRP must not be touched again.
3. Call **IoStartNextPacket** to send the next IRP to Start I/O.

Table 8.6. Function Prototype for a DpcForIsr Routine

VOID DpcForIsr	IRQL = DISPATCH_LEVEL
Parameter	Description
IN PKDPC pDpc	DPC object for this call
IN PDEVICE_OBJECT pDevObj	Target device for I/O request
IN PIRP pIrp	IRP describing this request
IN PVOID pContext	Context passed by IoRequestDpc
Return value	Void

Priority Increments

The Windows 2000 thread scheduler uses a priority boosting strategy to keep the CPU and I/O devices as busy as possible. As can be seen from the boost values listed in [Table 8.7](#), priority increments are weighted in favor of threads working with interactive devices like the mouse and keyboard.

As part of this strategy, a driver should compensate any thread that waits for an actual device operation by giving a priority boost. Choose an appropriate increment from the table and specify it as an argument to **IoCompleteRequest**.

Table 8.7. Priority Increment Values

Priority Increment Values		
Symbol	Boost	Use when completing
IO_NO_INCREMENT	0	No device I/O
IO_CD_ROM_INCREMENT	1	CDROM I/O
IO_DISK_INCREMENT	1	Disk I/O
IO_PARALLEL_INCREMENT	1	Parallel port I/O
IO_VIDEO_INCREMENT	1	Video output
IO_MAILSLLOT_INCREMENT	2	Mailslot I/O
IO_NAMED_PIPE_INCREMENT	2	Named pipe I/O
IO_NETWORK_INCREMENT	2	Network I/O
IO_SERIAL_INCREMENT	2	Serial port I/O

Some Hardware: The Parallel Port

Before walking through an example of a programmed I/O driver, it is helpful to look at some actual hardware. This serves the dual purpose of showing what kinds of devices tend to perform programmed I/O and providing a hardware example for the minimal driver to control.

How the Parallel Port Works

The parallel interface found on most PCs is based on an ancient standard from the Centronics Company. Although its original purpose was to communicate with printers, clever people have found many ways of attaching everything from disks to optical scanners to the parallel port. The DB-25 connector on this port carries a number of signals, the most important ones being

-

RESET.

The CPU sends a pulse down this line when it wants to initialize the printer.

-

DATA.

The CPU uses these eight lines to send one byte of data to the printer. On systems with extended parallel interfaces, these lines can also be used for input.

-

STROBE#.

The CPU pulses this line to let the printer know that valid information is available on the data lines. (The sharp character signifies an inverted signal. A valid pulse is sent by transitioning the line from a logical 1 to 0.)

-

BUSY.

The printer uses this line to let the CPU know that it can't accept any data.

-

ACK#.

The printer sends a single pulse down this line when it is no longer busy.

-

Errors.

The printer can use several lines to indicate a variety of not-ready and error conditions to the CPU.

The following sequence of events occurs during a data transfer from the CPU to a printer attached to the parallel port:

- 1.

The CPU places a byte on the eight data lines and lets the data settle for at least half a microsecond.

- 2.

The CPU grounds the STROBE# line for at least half a microsecond and then raises it again. This is the signal to the printer that it should latch the byte on the data lines.

- 3.

In response to the new data, the printer raises the BUSY line and starts to process the byte. This usually means moving the byte to an internal buffer.

- 4.

After it processes the character (which may take microseconds or seconds, depending on how full the printer's buffer is), the printer lowers the BUSY line and pulses the ACK# line.

As is apparent from this description, the parallel port offers a very low-level interface to the outside world. Most of the signaling protocol involved in a data transfer has to be implemented by the CPU itself. A driver for a legacy parallel port must be designed for this primitive operation.

Code Example: Parallel Port Loopback Driver

This example shows how to write a basic programmed I/O driver for the parallel port. The code for this example is in the *Chap8* directory on the disk that accompanies this book. Several code fragments follow.

Purpose of Driver

The purpose of this driver is to allow a test program to output nibbles (4-bit quantities) to the parallel port with the loopback connector attached. The data returned by the connector is stored in a temporary pool buffer within the driver. Thus, subsequent reads to the device should return the same nibble data that was output. Just to keep things interesting, the driver returns the nibble data shifted left by four bits.

Since the loopback connector is wired in a non-straightforward manner, the driver code must assemble the nibble data from various data and status bits.

Driver.H

The main header file for this driver builds on the two seen in previous chapters. Changes were made to the `DEVICE_EXTENSION` structure to support the parallel port hardware and driver-specific functionality.

```
typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;
    ULONG DeviceNumber;
    CUStrng ustrDeviceName;    // internal name
    CUStrng ustrSymLinkName;   // external name
    PCHAR deviceBuffer;        // temporary pool buffer
    ULONG deviceBufferSize;
    ULONG xferCount;           // current transfer count
    ULONG maxXferCount;        // requested xfer count
    ULONG portBase;           // I/O register address
    ULONG Irq;                 // Irq for parallel port
    PKINTERRUPT pIntObj;       // the interrupt object
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Additionally, macros were added to the file for convenience in reading the parallel port device registers.

```
#define PPORT_REG_LENGTH 4
#define DATA_REG 0
#define STATUS_REG 1
#define CONTROL_REG 2

//
// Define access macros for registers. Each macro takes
// a pointer to a Device Extension as an argument
//
#define ReadStatus( pDevExt ) \
    (READ_PORT_UCHAR( (PCHAR) \
        pDevExt->portBase + STATUS_REG ))

#define ReadControl( pDevExt ) \
    (READ_PORT_UCHAR( (PCHAR) \
        pDevExt->PortBase + CONTROL_REG ))

#define WriteControl( pDevExt, bData ) \
    (WRITE_PORT_UCHAR( (PCHAR) \
        pDevExt->portBase + CONTROL_REG, bData ))

#define WriteData( pDevExt, bData ) \
    (WRITE_PORT_UCHAR( (PCHAR) \
        pDevExt->portBase + DATA_REG, bData ))
```

Testing the Parallel Port Loopback Driver

At this point, a real driver is working, touching real hardware. A tester routine is included with this driver to demonstrate correct functionality and use. It verifies that the driver

- - Sends IRPs from its Dispatch routine to its StartI/O routine.
- - Responds to device interrupts.
- - Transfers data successfully.
- - Completes requests.
- - Manages requests from multiple callers.

On the other hand, this driver is far from complete. It assumes that all data transfer requests complete without error. Simply removing the parallel port loopback connector before starting the test demonstrates this problem—a hung system. Subsequent chapters correct this problem as well as explain and demonstrate the correct way to locate hardware.

Testing Procedure

When testing programmed to I/O device drivers, it is advisable to follow a fixed procedure.

1.
 - Write a minimal Start I/O routine that simply completes each IRP as soon as it arrives. This allows the test of the linkage between the driver's Dispatch and Start I/O routines.
2.
 - Write the real Start I/O routine, the ISR, and the DpcForIsr routine. If the driver supports both read and write operations, implement and test each path separately.
3.
 - Exercise all the data transfer paths through the driver with a simple Win32 program that makes **ReadFile**, **WriteFile**, and **DeviceIoControl** calls.
4.
 - Stress test the driver with a program that generates large numbers of I/O requests as quickly as possible. Run this test on a busy system.
5.
 - If the device is sharable, run several copies of the test program at once to be sure the driver works with multiple open handles.
6.
 - If the driver supports multiple physical devices, repeat the test with each device unit.
7.
 - If possible, repeat steps 4 to 6 on a multiprocessor system to verify SMP synchronization.

Summary

At this point, the basic components of the working driver are in place. The Start I/O routine is initiating each request and the ISR is servicing interrupts. The DpcForIsr routine is correctly completing IRPs below the DIRQL level of the device.

In the next chapter, the detection of Plug and Play devices is covered.

Chapter 9. Hardware Initialization

CHAPTER OBJECTIVES

- - [The Plug & Play Architecture: A Brief History](#)
- - [The Role of the Registry for Legacy Drivers](#)
- - [Detecting Devices with Plug and Play](#)
- - [The Role of Driver Layers in Plug and Play](#)
- - [The New WDM IRP Dispatch Functions](#)
- - [Device Enumeration](#)
- - [Device Interfaces](#)
- - [Code Example: A Simple Plug and Play Driver](#)
- - [Summary](#)

At the end of the last chapter, the sample parallel loopback driver was unrealistic in one critical area: the configuration of the parallel hardware was presumed. Hard-coded driver logic assumed that port hardware could be found at a fixed I/O address with a fixed IRQ level. Clearly, real driver code cannot be presumptuous. This chapter covers the subject of driver and device initialization.

First, in sections 9.1 and 9.2 the challenges of initialization are discussed from a historical perspective. The role of the Windows 2000 Registry in tracking installed devices is covered.

Next, the current Plug and Play architecture of Windows 2000 is discussed, including the use of layered drivers used in this implementation. The specifics of new IRP Dispatch functions are shown.

Finally, the driver presented in the last chapter is "corrected" to use the Plug and Play architecture. The parallel port hardware is automatically detected and its real configuration is used.

The Plug and Play Architecture: A Brief History

It is apparent that the revolution of the PC within the industry occurred because everyone could afford and use computers. Yet more than a decade went by during which naive users were expected to understand the hardware details of each device inserted into their systems. Knowledge of I/O port, interrupt level, and DMA channel usage of every existing device within a system was prerequisite when installing a new device.

Further, the design of operating systems assumed a very static hardware configuration. The installation or removal of a device required, at a minimum, a full reboot of the OS.

With the advent of Windows 95 (and some hardware prerequisites), a concerted effort to automate the configuration of new and removed devices was implemented. This attempt greatly enhanced the acceptance of and migration to Windows 95, which in turn accelerated the migration to 32-bit OS PC environments such as NT. With Windows 2000, Microsoft has enhanced and implemented a complete Plug and Play architecture for the I/O subsystem.

Goals of Plug and Play

The overall goal of the Plug and Play (PnP) architecture is to provide automated support for the installation and removal of system devices. To support this overall goal, several features are necessary.

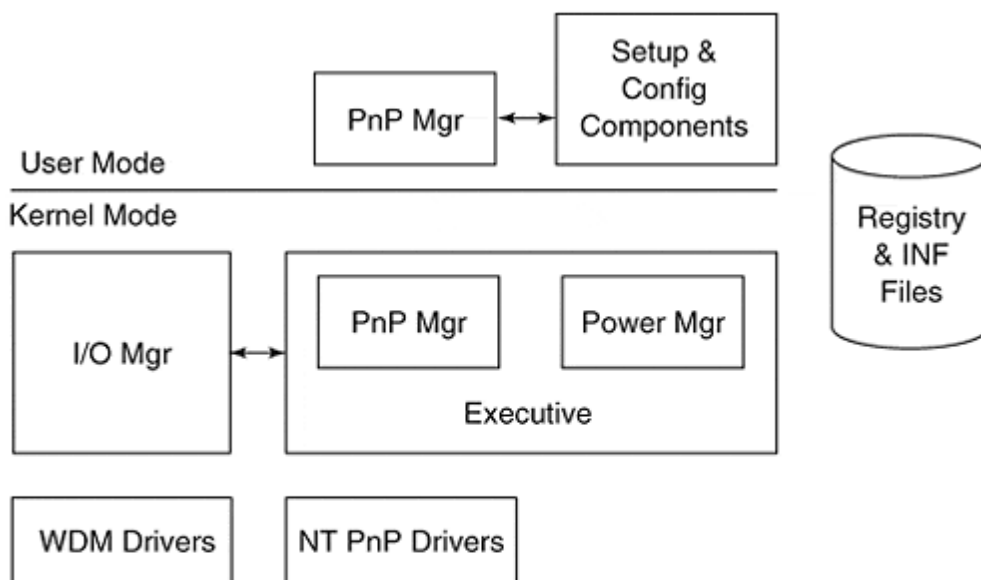
- Automatic detection of installed and removed hardware. The device and the bus in which it inserts must notify controlling software that a particular device configuration has changed.
- Devices must allow for software configuration. The port, IRQ, and DMA resources used by a device must be capable of assignment by controlling software. (In other words, configuration of a board can no longer come from DIP switches or jumpers.)
- Necessary drivers for new hardware must be automatically loaded as needed by the operating system.
- When devices and their interfacing buses permit, the system should support *hot plugging* of the device. That is, it should be possible to insert or remove the device into or from a "live" system without disturbing operation.

While necessary, these goals are aggressive and invasive, and they require support from all levels of hardware and software.

Components of Plug and Play

The Windows 2000 operating system implements PnP with numerous software components. These components are depicted in [Figure 9.1](#).

Figure 9.1. PnP components.



The Role of the Registry for Legacy Drivers

In recent history, it was common for hardware to remain dormant until software somehow magically became aware of its existence and began to stimulate it. The techniques used by drivers and the NT operating system fell into one of three categories:

- - A driver would maintain a list of potential hardware resources (i.e., port address, DMA channels, and interrupt levels) for each device that it might encounter. By probing each potential resource during DriverEntry, a driver would create an appropriate Device object (**IoCreateDevice**).
- - A driver would rely upon an installation program to specify (either by probing or by user specification) the resources and devices that the driver would manage. This list of devices and associated resources would then be conveniently stored in the system registry.
- - The NT operating system, as part of a preboot process, would probe for conventional devices and resources. For example, COM serial ports are routinely present at 0x3F8 or 0x2F8 (et. al.) and are therefore probed during the boot process. The discovery of devices was reported and maintained within the system Registry.

As system designers recognized the need for a more orderly hardware configuration process, new buses and protocols were specified that automatically reported the presence (or removal) of hardware. Generally, any bus type other than ISA supports some form of autodetection.

As an intermediate step to take advantage of autodetecting buses and hardware, earlier versions of NT augmented the preboot process to include autodetected hardware within the system Registry. Thus, an initializing driver (DriverEntry) would have the ability to see the list of autodetected hardware that was present during the boot process and create appropriate devices.

Regardless of which technique the OS or driver utilized, it was still imperative to force the loading of a driver *manually*. That is, appropriate entries within the system registry allowed a driver to load (either at system boot or later), which in turn would control discovered devices. Indeed, the examples given in previous chapters rely upon a manual installation of the driver. Because a driver would first load and then begin to control devices, the technique is sometimes referred to as *driver-centric*.

With the advent of Windows 95, and subsequently Windows 98 and Windows 2000, the model has been reversed. Devices that announce themselves, either during boot or by subsequent insertion (hot plug), force the loading of an appropriate registered driver. The new technique is said to be *device-centric*.

While the original Windows 95 driver model supported Plug and Play, it differed significantly from the NT driver model. Microsoft, recognizing the need for a compatible driver environment, extended the NT model to support Plug and Play and incorporated this technique in Windows 2000 and Windows 98. The new common model is designated the *Windows Driver Model* (WDM).

Detecting Devices with Plug and Play

The WDM model is an extension of the NT driver model. DriverEntry still serves as the driver's initial entry point, but its responsibilities are reduced. In particular, the role of DriverEntry is limited to the announcement of other driver functions. Specifically, DriverEntry does not create device objects for hardware it is capable of controlling.

A new driver function, AddDevice, is announced within (interestingly) the Driver Extension. The prototype for the AddDevice function is shown in [Table 9.1](#) and is announced with code typical of the following:

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject,
                   IN PUNICODE_STRING pRegistryPath ) {
    :
    // Export other driver entry points...
    // for example...
    pDriverObject->DriverUnload = DriverUnload;
    pDriverObject->DriverExtension->AddDevice =
        AddDevice;
    :
}
```

Table 9.1. Function Prototype for an AddDevice Routine

NTSTATUS AddDevice	IRQL == PASSIVE_LEVEL
Parameter	Description
IN PDRIVER_OBJECT pDriverObject	Pointer to driver object
IN PDEVICE_OBJECT pdo	Pointer to physical device object
Return value	Success or failure code

The prime responsibility of the AddDevice function is to create a device object using **IoCreateDevice** in the same manner described in [chapter 6](#).

The Role of Driver Layers in Plug and Play

The WDM model of drivers is built upon structured layers of Physical Device Objects (PDOs) and Functional Device Objects (FDOs). Generally, a PDO exists for each physical piece of hardware attached to a bus and assumes the responsibility for low-level device control common to multiple functions realized by the hardware. An FDO exists for each logical or abstract function presented to higher-level software.

As an example, consider a physical disk drive and driver. It can be represented by a PDO that implements bus adapter functions (e.g., adapting an IDE disk bus to a PCI bus). Once the PDO for the disk bus adapter is realized, an FDO assumes responsibility for the functional operations of the disk itself. The FDO may choose to handle a particular disk I/O request directly (e.g., reading a disk sector). It may choose to pass down other requests to its physical device partner (e.g., a power down request).

In reality, the role of a PDO can quickly become complicated and recursive. For example, a USB host adapter starts life as a physical device of the bus it adapts (e.g., PCI). The host adapter then takes on the role of a new bus driver, enumerating each USB device it discovers as its collection of PDOs. Each PDO then controls its FDO.

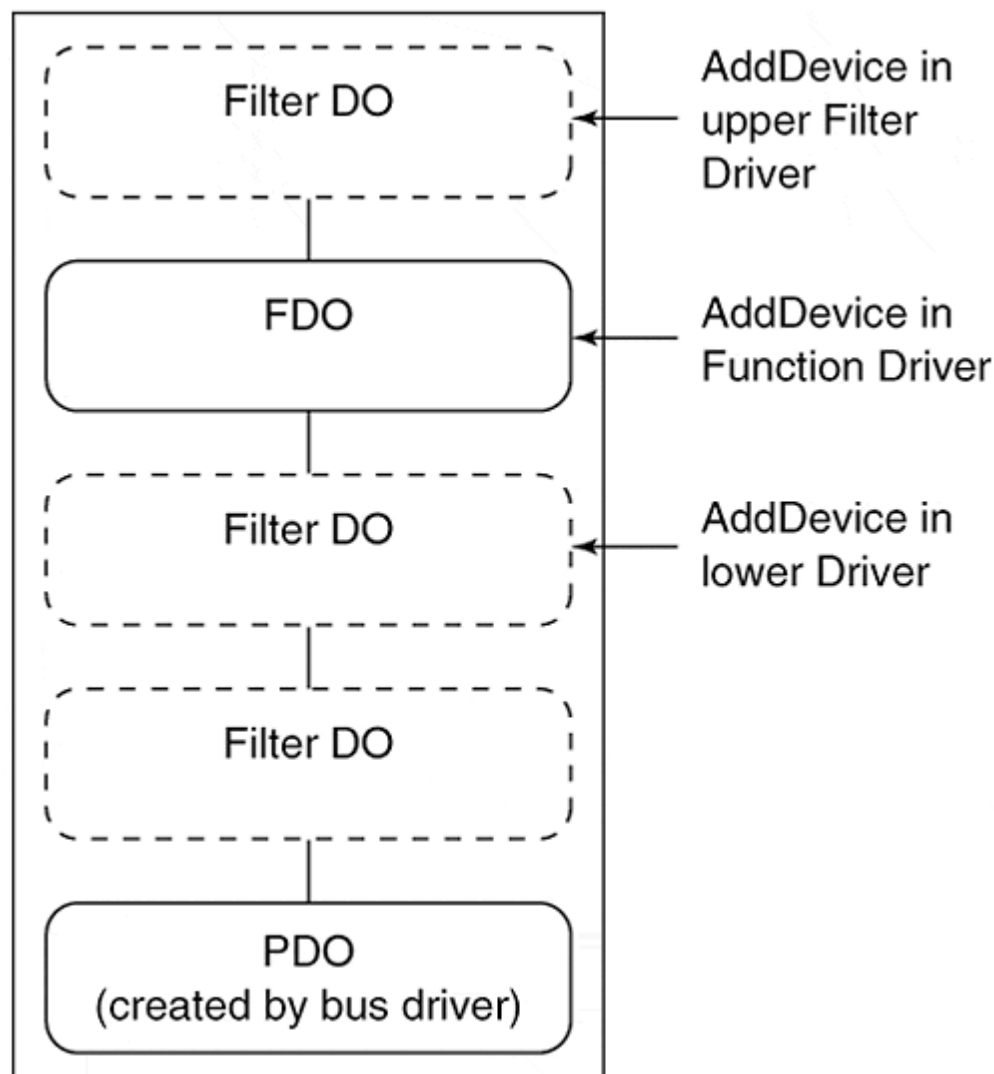
The technique is further complicated by allowing FDOs to be surrounded by *filter device objects*. These upper- and lower-level filter objects may exist in any number to modify or enhance the way that I/O Requests are handled by the resulting stack of device objects.

In order to distinguish between those FDOs which implement hardware buses and those that do not, the terms *bus FDO* and *nonbus FDO* are used within the DDK. A bus FDO implements the bus driver's responsibility of enumerating the devices attached to a bus. The bus FDO then creates a PDO for each attached device.

It is also worth noting that there is only a conceptual difference between a nonbus FDO and a filter device object. From the PnP Manager's perspective, all Device objects position themselves within the device stack, and the fact that some devices choose to consider themselves more than a filter is inconsequential.

The arrangement of a device stack is shown in [Figure 9.2](#). The distinction between a bus FDO and a nonbus FDO is depicted in [Figure 9.3](#).

Figure 9.2. The device stack.



The New WDM IRP Dispatch Functions

The AddDevice function, called by the PnP Manager, merely initializes the device (and its extension) object. It is apparent from the AddDevice code that hardware is not yet touched. In fact, two general responsibilities remain for a driver.

-
- Reserve and configure hardware resource requirements for the device
-
- Initialize the hardware to prepare it for use

Both tasks are performed by a driver upon receipt of a special IRP function (and subfunction) code that is new for WDM drivers: IRP_MJ_PNP. PnP IRP codes are sent by the PnP Manager when a variety of events occur, including

-
- Device initialization (perhaps due to insertion)
-
- Device shutdown (perhaps due to removal)
-
- Configuration queries

As described in [chapter 6](#), IRP major function codes, such as Read and Write requests, are handled by an indexed table of Dispatch routines. Since an entire category of IRP_MJ_PNP messages are routed through this single Dispatch routine, it is the responsibility of this handler to perform a secondary dispatch using the minor subcode contained within the IRP. For PnP, the minor subcodes take the symbolic form IRP_MN_XXX, where XXX is a specific Plug and Play action requested by the PnP Manager.

An example of the initialization of the major function dispatch is shown below.

```
...
pDriverObject->MajorFunction[IRP_MJ_PNP] =
    DispPnP;
```

The code to perform the secondary dispatch based upon the minor subcode of the IRP is shown below.

```
NTSTATUS DispPnp(          IN PDEVICE_OBJECT pDO,
                          IN PIRP pIrp ) {
    // Obtain current IRP stack location
    PIO_STACK_LOCATION pIrpStack;
    pIrpStack = IoGetCurrentIrpStackLocation( pIrp );
    switch (pIrpStack->MinorFunction) {
    case IRP_MN_START_DEVICE:
        ...
    case IRP_MN_STOP_DEVICE:
        ...
    default:
        // if not supported here, just pass it down
        return PassDownPnP(pDO, pIrp);
    }
    // all paths from the switch statement will "return"
    // the results of the handler invoked
}
```

Required Plug and Play IRPs

In order to be WDM compliant, drivers *must* support specific PnP IRPs, depending upon the type of device object—nonbus FDO, bus FDO and PDO. [Table 9.3](#) lists the subcodes that must be supported for all device object types.

Device Enumeration

As briefly described earlier in this chapter, the PnP Configuration Manager is responsible for *enumerating* the hardware discovered on a system. Starting at system boot, or as devices are inserted or removed, a bus driver is responsible for identifying and listing (i.e., *enumerating*) attached hardware. The hardware resources assigned to a device are supplied to the driver when the PnP Start Message subcode (IRP_MN_START_DEVICE) is sent.

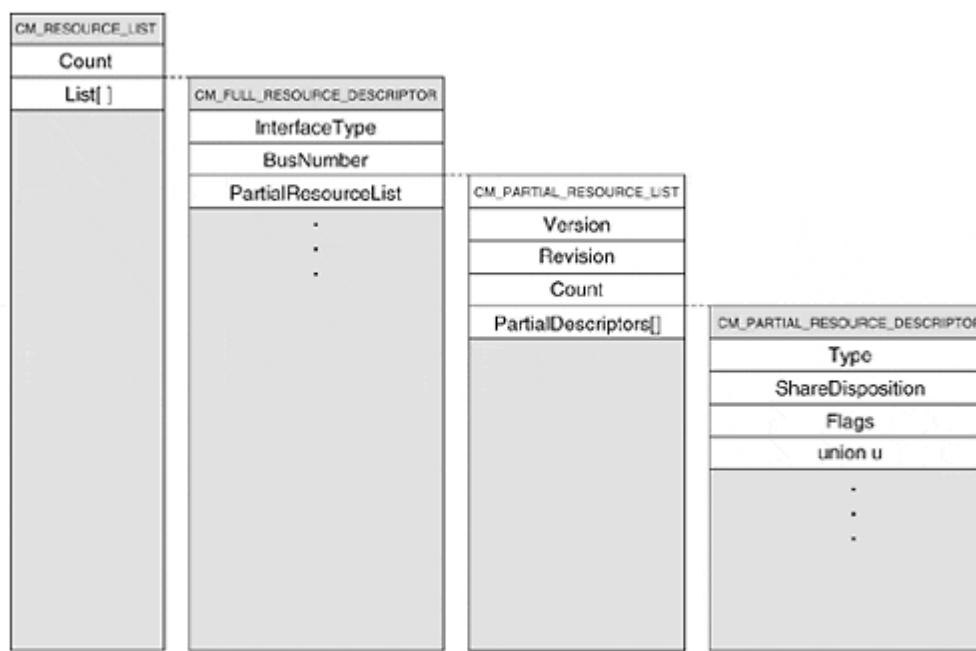
The technique used to assign unique hardware resources to a device is bus-and driver-dependent. For example, it is possible that two devices installed in a system have conflicting resource requirements at boot time. Both devices might default to the same I/O port address. The PnP Configuration Manager is responsible for sorting out any such conflicts. Further, an individual driver (FDO) might choose not to utilize a resource at all. For example, printer drivers do not always use an interrupt line even though the hardware allows it. An unused printer IRQ can be safely assigned to other devices. In other words, the assignment of hardware resources is a dynamic and iterative process involving the bus and device hardware, the PnP Manager, and the device driver.

Hardware Resource Descriptors

When a driver receives the PnP subcode IRP_MN_START_DEVICE, two fields within the IRP stack list the assigned hardware resources: **Parameters.StartDevice.AllocatedResourcesTranslated** and **Parameters.StartDevice.AllocatedResources**.

The structure used to describe these resources is of the type CM_RESOURCE_LIST, which is a *counted array*. The first offset within the structure, **Count**, signifies the number of array elements which follow. Each array element is of the type CM_FULL_RESOURCE_DESCRIPTOR. The array element contains a bus type and number (e.g., ISA bus 0) as well as a CM_PARTIAL_RESOURCE_LIST, which is another counted array. Each element of the inner array is of the type CM_PARTIAL_RESOURCE_DESCRIPTOR, which finally (and thankfully) describes the resources assigned to the device. [Figure 9.6](#) depicts this four-level structure.

Figure 9.6. Resource list data structures.



The structure of interest is within the fourth level and is a union, **u**. The four significant united types are **Port**, **Interrupt**, **Memory**, and **Dma**. The fields of each of these types are copied into the device extension and are used by an FDO driver for the duration. For example, when the time comes to perform an actual DMA operation, the **Channel** and **Port** offsets of the **Dma** structure are required.

Example code that would retrieve the *i*th element within the PARTIAL_RESOURCE_LIST follows.

```

PCM_RESOURCE_LIST pResourceList;
PCM_FULL_RESOURCE_DESCRIPTOR pFullDescriptor;
PCM_PARTIAL_RESOURCE_LIST pPartialList;
PCM_PARTIAL_RESOURCE_DESCRIPTOR pPartialDescriptor;
int i;
pResourceList =
    &Parameters.StartDevice.AllocatedResourcesTranslated;
pFullDescriptor =
    pResourceList->List:

```

Device Interfaces

In [chapter 7](#), a primitive mechanism for extending the Dispatch interface is described. It relies upon a single user-mode call, `DeviceIoControl`, which allows specification of a control code that is generally device-specific. The remaining arguments of the Win32 API call specify separate input and output buffers. Of course, the structure and meaning of the buffers is code-dependent and relies upon a specification outside the scope of the I/O Manager or any other system component.

Interface Definition

With the introduction of Plug and Play for Windows 2000, Microsoft allows device drivers to extend a generic *interface* to user-mode code. An interface is a *specification*, or contract, between caller and implementer that is nothing more than a group of (hopefully) related function calls (methods). A single driver can support many interfaces, and interfaces can be reimplemented by many drivers. Thus, the functionality of a driver (using interfaces) is merely the mix and match conglomeration of the interfaces it implements.

An interface should be *immutable*. Once formally published (or distributed), an interface should not be changed. Once multiple drivers implement the same interface, it realistically cannot be changed. This includes additions as well as deletions. The immutability requirement allows client code to be certain that if a driver supports an interface, its meaning is completely unambiguous.

Interfaces are identified by a unique number, or interface type (ID). To avoid inter-vendor collisions, the ID space is somewhat large, consisting of 128 bits. Microsoft has chosen to conform to the Open Software Foundation (OSF) Distributed Computing Environment (DCE) specification for the generation of these Universally Unique Identifiers (UUID). To generate a statistically unique UUID, Microsoft supplies two tools, GUIDGEN (Windows based) and UUIDGEN (console based).

Although a direct call interface is available only for kernel-mode code, user-mode code can also benefit. The interfaces supported by a driver can be enumerated by an application, and if recognized, offer a level of assurance as to driver functionality.

Interface Construction

Once a suitable ID is generated, the interface is constructed with a structure that includes a function pointer for each method of the specification. All device interfaces are based on the following structure:

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

If using C++, it is acceptable to define the new interface using inheritance (structs support inheritance as well as classes in C++). For example, to define an interface used to launch a missile,

```
typedef struct _COORDINATES {
    LONG latitude;
    LONG longitude;
} COORDINATES, *PCOORDINATES;
typedef BOOLEAN
    (*PLAUNCH_INTERFACE_LAUNCH)(COORDINATES coords);
typedef BOOLEAN
    (*PLAUNCH_INTERFACE_DESTROY)(VOID);
// Derive the new struct from the generic struct
typedef struct _LAUNCH_INTERFACE : INTERFACE {
    // only need to define interface-specific entries
    PLAUNCH_INTERFACE_LAUNCH Launch;
    // the next function cancels a missile launch
    // we wouldn't want this function to be optional
    PLAUNCH_INTERFACE_DESTROY Destroy;
} LAUNCH_INTERFACE, *PLAUNCH_INTERFACE;
```

Code Example: A Simple Plug and Play Driver

A complete, albeit simple, Plug and Play driver for the parallel port is included on the CD which accompanies this book and on the companion *Web site*, <http://www.W2KDriverBook.com>.

Summary

The conversion of a legacy driver into a simple WDM driver is relatively straightforward, as demonstrated with the previous example. Nevertheless, the number of details requiring mastery and understanding is formidable. It pays to have a base understanding of Windows 2000 drivers *before* attempting to add Plug and Play support.

The next chapter deals with the "full blown" WDM and PnP models, examining and demonstrating power management and hot plug issues.

Chapter 10. Power Management

CHAPTER OBJECTIVES

-

- [Hot Plug Devices](#)

-

- [OnNow Initiative](#)

-

- [Wake Requests](#)

-

- [Power Management Issues](#)

-

- [Summary](#)

The WDM model is ideal for enumerating and arbitrating discovered hardware. Additionally, the model supports the frugal management of power to those devices.

Devices that consume power only when used are hardly a new concept. After all, household appliances normally remain off until used. Yet within legacy computers, every device consumes power even during significant idle periods. The fact that a device might be burning just a few watts or less encourages the thought that the waste is unworthy of concern.

The need to manage power to installed devices is obvious in mobile and portable environments where the source of energy is limited. Even in desktop and server environments, however, the reduction of unnecessary power expenditure lowers heat production, increases component reliability, and lowers electric consumption and cost.

This chapter starts by describing the goals of and problems associated with power management. Then the mechanics of power management with a device driver is presented. This involves gracefully powering down a device through various levels of power consumption and reversing the process when new device requests arrive.

Finally, an example driver with power management handling incorporated is presented.

Hot Plug Devices

In order to manage power within a computer system, three major categories require cooperative design.

- - System and bus hardware
- - Devices
- - Operating system and device driver software

Windows 2000 and Windows 98 support the WDM driver model, which incorporates sophisticated power management for bus and device hardware. Since the remainder of this chapter discusses the software support for power management, it is appropriate to first review the hardware requirements for the feature.

Bus Considerations

First and foremost, before a device can be powered down, the bus into which it plugs must electrically provide for such a state. For example, many bus drivers require power to guarantee an on-bus or off-bus state. Without power to the electrical bus-driver chip itself, the driver and its effect on the rest of the bus is indeterminate. Clearly, the hardware bus specification must allow for powered down bus drivers in order to support robust power management.

In some cases, the bus design must meet physical constraints. Especially in mobile systems, devices are routinely inserted and ejected. If the device installation and removal is allowed on a "hot" system, the physical bus characteristics must be carefully designed. Besides the obvious requirement that each device be physically separate (i.e., one device cannot scrape another as it is inserted or removed) there is the less obvious concern for which bus signals will "break" first. Typically, the ground pins for such buses are physically longer than the power pins, ensuring that ground *makes first* and *breaks last* during insertion and removal.

Such trivia aside, all modern buses support power management and incorporate necessary electrical and physical specifications to meet the stated requirements of the bus. PCI, PCMCIA (PC Card), USB, and IEEE 1394 are all designed for the feature. All but PCI allow "hot plug" of devices.

Device Considerations

Besides meeting the requirements for a given bus, individual devices can allow for remarkably clever management of power. For example, a modem card might allow for power down of the UART and bus circuitry, while still powering a ring detection circuit that triggers an awaken sequence.

Indeed, when considering the possible segregation of power to a device's subsections, it becomes clear a device can have multiple states for describing its power condition. Between the extremes of "completely on" and "completely off," a device can implement decreasing power states corresponding to decreasing capability. The transition from state to state can be directed by system or user request, or simply after an extended idle time.

OnNow Initiative

In an attempt to standardize (read: *abstract*) the various approaches to power management, Microsoft developed the OnNow Initiative. This specification builds on the capabilities of the hardware, providing a common platform for the various Microsoft operating systems to exploit.

The OnNow initiative abstracts the power management capabilities of the system (mainboard) and devices. While individual devices may incorporate many clever powering schemes, the OnNow model describes distinct power states that a device or system can occupy at any given time.

Power States

A system exists in one of six power states, designated S0 through S5. The meanings are described in [Table 10.1](#). The overall state of the system largely restricts the maximum power state of individual devices.

Devices, meanwhile, occupy one of four power states, D0 through D3. The meanings are described in [Table 10.2](#). Device power state primarily determines whether or not the hardware is capable of retaining its internal context at the designated Dx level.

Table 10.1. System Power States Defined by OnNow

System Power State	Meaning
S0	CPU fully on Devices may occupy any power state
S1	CPU halted; RAM refreshed
S2	CPU without power; RAM refreshed
S3	CPU without power; RAM in slow refresh mode Power supply output reduced
S4	System off; RAM saved to disk
S5	System off and shutdown; full reboot necessary to restore operation

Table 10.2. Device Power States Defined by OnNow

Device Power State	Meaning
D0	Device fully on
D1	Device in low-power mode
	Device context probably retained
D2	Device in low-power mode
	Device context probably not valid
D3	Device is without power; context lost

A confusing terminology point arises from the power state designations. As the system or device moves to a lower power state, its power state designation *increases*. For example, a device that moves from fully on to fully off has its power state *increase* from D0 to D3. The problem is compounded by the fact that the header file defining the enumeration of the states (wdm.h) specifies *increasing* enumeration values for the *decreasing* power-consuming states. For example, D0, enumerated symbolically as PowerDeviceD0, has the ordinal value 1. PowerDeviceD3 has the ordinal value 4. Thus, ordinal comparisons of power states are confusing. For example, a test of *newState < oldState* is a check to see if the new state consumes more power than the old state.

To further confuse the issue, PowerDeviceMaximum is defined to be the maximum power consumption state of a device. As currently defined within <wdm.h>, its ordinal value is 5.

Power Policies

Devices of the same class share a common power *policy*. The policy is owned by a single driver, typically a high-level driver common to all device stacks of the class (e.g., a class driver). The policy owner makes power decisions for the entire class of devices and is responsible for communicating with the operating system to confirm or reject power state changes.

Once the policy owner decides on a power state change, it issues IRP requests to individual devices or stacks to enact the decision. Such an IRP might pass between many drivers (down to the bus driver, up to the PDO, back to filters and FDO) before the entire request is realized.

Regardless of which driver assumes the role of policy owner, each device stack driver is responsible for acting on individual power requests. This means that each FDO is responsible for the device-specific power manipulation, while the PDO bus drivers take responsibility for manipulating power to the bus slot.

Power State Matrix

Wake Requests

To understand the Windows 2000 architecture for handling wake requests, it is important to first understand the hardware operation. A device that is *armed* for wake detection appears to have the capability to force a sleeping system back to life. In fact, since the device is a slave of the bus, it really has only the capability to signal its intent to the bus. It is the bus hardware that in turn forces the system out of its sleep.

The significance of this is that the software that drives the hardware is ultimately centered at the bus driver. The arming process starts at the top of the device stack, but the actual waiting occurs at the bottom bus driver. Thus, the overall process for the wake process is

1.

A power policy owner (usually a bus driver) initiates a request to arm or disarm the wake mechanism for a device using `PoRequestPowerIrp`, subcode `IRP_MN_WAIT_WAKE`. (A callback routine, fired when the IRP eventually finishes, is set.)

2.

A driver of wake-capable hardware handles requests to arm and disarm the trigger mechanism.

3.

Wake-aware drivers pass down the arm/disarm request to lower-level drivers. The request will pend (perhaps for hours) at the bus driver.

4.

The system enters sleep mode.

5.

The wake hardware (obviously without suspended software support) tickles the bus and awakens the system CPU.

6.

The bus driver (now executing) completes the original `IRP_MN_WAIT_WAKE` IRP.

7.

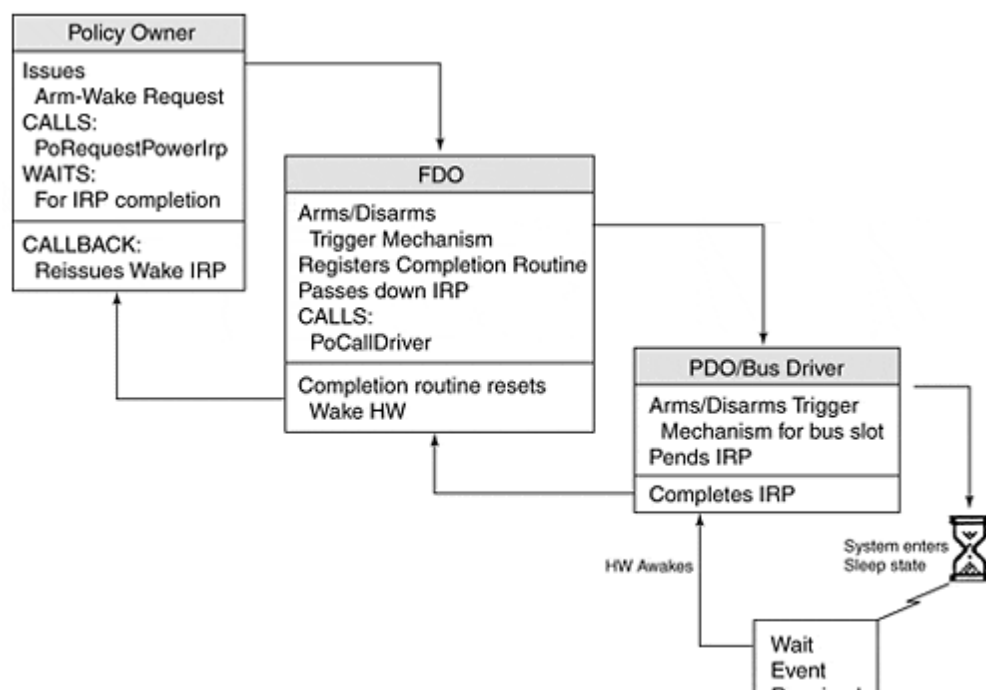
Any higher-level driver with a registered I/O Completion routine will be notified as the IRP works its way back up the device stack. A function driver might need to reset the wake trigger mechanism.

8.

The original creator of the arming request is notified that the (multihour) process is complete. Presumably, the creator re-arms the wake trigger mechanism for the next (multihour) cycle.

This process is depicted in [Figure 10.2](#).

Figure 10.2. The wake process.



Power Management Issues

As part of the OnNow initiative, Microsoft, working with hardware system developers, established the Advanced Configuration and Power Interface (ACPI) BIOS. ACPI replaces the outdated Advanced Power Management (APM) BIOS interface for power management. Proper operation of Windows 2000 power management features depend upon BIOS support of ACPI. When installed on non-ACPI-compliant platforms, Windows 2000 degrades to the capability extended by APM-compliant BIOS.

To fully exercise WDM driver power management code, a system with ACPI BIOS enabled must be used. Further, Windows 2000 should be installed with ACPI enabled (typically a BIOS setup screen option).

While APM-only support is suitable for gross-level power management, the full blown ACPI feature set is required to gain the full range of device power management supported by Windows 2000. APM support is sufficient for battery management in mobile devices.

Idle Management

Since the most obvious form of power savings that occurs is the power down of idle devices (i.e., devices that remain unused for an extended period of time), Windows 2000 provides a convenient mechanism for device drivers to request a power IRP `IRP_MN_SET_POWER` after a specified timeout period. The Power Manager function `PoRegisterDeviceForIdleDetection`, described in [Table 10.8](#), provides this functionality.

The return value from `PoRegisterDeviceForIdleDetection` is a pointer to an idle counter that is incremented each second. When and if the counter reaches the timeout value specified (argument two or three), the Power Manager sends an `IRP_MN_SET_POWER` request of the power state specified in argument four.

The timer/counter must be manually reset by driver code, as appropriate, by calling `PoSetDeviceBusy`, described in [Table 10.9](#). Of course, setting too short a timeout interval burdens a driver with frequent calls to `PoSetDeviceBusy`, so conservative timeout values would seem appropriate.

Table 10.8. Function Prototype for `PoRegisterDeviceForIdleDetection`

PULONG PoRegisterDevice ForIdleDetection	IRQL < DISPATCH_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pDevObj	Pointer to device object to receive power IRP
IN ULONG ConservationIdleTime	Timeout value (seconds) when system is running with scarce power (battery-driven)
IN ULONG PerformanceIdleTime	Timeout value (seconds) when system is running with abundant power (docked)
IN DEVICE_POWER_STATE State	Power state requested when timeout exceeded
Return value	• Pointer to "idle counter"
	• NULL - Idle timeout could not be enabled

Table 10.9. Function Prototype for `PoSetDeviceBusy`

VOID PoSetDeviceBusy	IRQL < DISPATCH_LEVEL
Parameter	Description
IN PULONG pIdleCounter	Pointer to "idle counter"
Return value	- void -

User Interface for Power Management

The suggested user interface for Power Management is the Control Panel applet of Windows 2000, Power Options. To extend the Power Options applet with a driver-customized property sheet requires the use of Microsoft's COM object model and registration of the property sheet. An example of providing such an extension is provided on the companion Web site of this book, www.W2KdriverBook.com

Some knowledge of ATL (Active Template Library) is required to understand the example.

Summary

Power Management within the Windows 2000 WDM model is somewhat complex. The nature of power management with multiple device power states contributes to the complexity. Partly, the child-parent, device-bus relationship is also to blame. And of course, the handling of wake requests from otherwise sleeping devices adds more code to the driver.

Nevertheless, a properly written power-managed WDM driver contributes greatly to the elegant operation of the overall system. Power consumption, heat, and reliability benefit from the incorporation of this feature.

The next chapter deals with the handling of a practical device problem: timeouts. What if a hardware request does not complete within a reasonable period of time?

Chapter 11. Timers

CHAPTER OBJECTIVES

- [Handling Device Timeouts](#)
- [Code Example: Catching Device Timeouts](#)
- [Managing Devices without Interrupts](#)
- [Code Example: A Timer-Based Driver](#)
- [Summary](#)

Software developers have known about it from the beginning. Hardware does not always behave the way it should. For example, error conditions may prevent a device from generating an interrupt when one is expected. Even worse, some devices don't use interrupts to signal interesting state changes. Handling these situations requires some kind of timer or polling mechanism, and this chapter describes just such a process.

Handling Device Timeouts

A driver can never assume that an expected device interrupt will arrive. The device might be offline; it might be waiting for some kind of operator intervention; or, perhaps it's just broken. This section explains how to use I/O Timer routines to detect unresponsive devices.

How I/O Timer Routines Work

An I/O Timer routine is a (somewhat) optional piece of driver code that an **AddDevice** routine attaches to a specific Device object. After the timer is started, the I/O Manager begins calling the I/O Timer routine once every second. These calls continue until the timer is explicitly stopped. [Table 11.1](#) lists the functions available for working with I/O timers.

[Table 11.2](#) shows the prototype for the I/O Timer routine itself. When this callback executes, it receives a pointer to the associated Device object and the context argument that was specified in the call to **IoInitializeTimer**. As always, the address of the Device Extension is a good choice for context.

How to Catch Device Timeout Conditions

In general terms, a driver that wants to catch device timeouts should do the following:

1.
In **AddDevice**, call **IoInitializeTimer** to associate an I/O Timer routine with a specific device.
2.
When a handle is associated with the device (e.g., when a user-mode program calls **CreateFile**), the driver's Dispatch routine for **IRP_MJ_CREATE** calls **IoStartTimer**. As long as the device handle remains open, the device receives I/O Timer calls.
3.
If the driver needs a longer timeout than one second (and it probably does), a separate counter value must be maintained. This counter is initialized in the Start I/O routine to the maximum number of seconds the driver is willing to wait for an interrupt.
4.
The ISR resets or cancels the timer, depending on whether additional device interrupts are expected.
5.
Each time the driver's I/O Timer routine is called, the timer counter is decremented and checked. If the counter reaches zero before an interrupt arrives, the I/O Timer routine stops the device, clears the timeout counter, and processes the request as a timed-out operation.
6.
When the user-mode program calls **CloseHandle**, the Dispatch routine for **IRP_MJ_CLOSE** calls **IoStopTimer**, which disables I/O timer callbacks for the device.

Notice that the Start I/O and I/O Timer routines (running at **DISPATCH_LEVEL** IRQL), and the ISR (running at **DIRQL**) all have access to the timeout counter in the Device Extension. These independent code paths must synchronize their access to the timeout counter. The code example that appears later in this chapter demonstrates such synchronization.

Table 11.1. I/O Timer Routines

Function	Purpose	IRQLM
IoInitializeTimer	Attach a timer to a device	PASSIVE_LEVEL
IoStartTimer	Start receiving callbacks	<= DISPATCH_LEVEL
IoStopTimer	Stop receiving callbacks	<= DISPATCH_LEVEL

Table 11.2. Function Prototype for IoTimer Callback

VOID IoTimer	IRQL = DISPATCH_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pDeviceObject	Pointer to device object whose timer just fired
IN PVOID pContext	Context argument passed with IoInitializeTimer
Return value	- void -

Of course, a driver can choose to use **IoStartTimer** and **IoStopTimer** outside of Dispatch routines. Since the timer callback interval

Code Example: Catching Device Timeouts

This example shows the addition of timeout support to the simple parallel port driver developed in previous chapters.

Device Extension Additions

The Device Extension is modified to include the timeout counter. A counter value of -1 signifies that timeouts are to be ignored.

```
typedef struct _DEVICE_EXTENSION {
    ...
    LONG timeRemaining; // timeout counter - seconds
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

AddDevice Additions

AddDevice is changed to initialize the I/O Timer for the device. The counter value does not need to be initialized until the timer is actually started.

```
NTSTATUS AddDevice(IN PDRIVER_OBJECT pDriverObject,
                 IN PDEVICE_OBJECT pdo) {
    PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
        pDevObj->DeviceExtension;
    ...
    // Near the end of the function, after IoCreateDevice
    // has been called....
    //
    // Create the I/O Timer
    IoInitializeTimer( pdo, IoTimer, pDevExt);
    ...
}
```

Create Dispatch Routine Changes

When the device is "opened" from user-mode (via the Win32 **CreateFile** call), the I/O Timer is started. It continues to tick so long as the handle remains open. Since the time is ticking, the timeout counter must be initialized to show that at present the ticks should be ignored.

```
NTSTATUS DispatchCreate( IN PDEVICE_OBJECT pDevObj,
                       IN PIRP pIrp ) {
    PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
        pDevObj->DeviceExtension;
    ...
    // Near the end of the function, the timeout
    // counter is initialized and the I/O Timer
    // is started.
    pDevExt->timeRemaining = -1;
    IoStartTimer( pDevObj );
    ...
}
```

StartIo Changes

Each time the physical device is started and an interrupt becomes expected, the maximum number of seconds to wait for the interrupt must be set into the timeout counter. An operational counter must be synchronized with all code paths to ensure it does

Managing Devices without Interrupts

Some devices don't generate interrupts with every significant state change. Even when the device does generate interrupts, a case can be made to drive an infrequently used, relatively slow device with a polled technique. Printer drivers are a common example of this technique. Since the printer device buffer is large in comparison to the physical print rate, a device driver has a considerable window in which it can occasionally check for the need to refill the buffer.

Working with Polled Devices

Once a device is started, it is generally unacceptable for driver code to hang in a tight loop waiting for the device to finish. In a multiprocessing and multitasking operating system such as Windows 2000, there are always useful chores that can or must occur in parallel with the device's operation.

By Microsoft edict, a driver is not allowed to stall in a tight polling loop for more than 50 microseconds. With today's extraordinary processor speeds, 50 microseconds is an eternity—a period in which hundreds of instructions could have executed.

When polling is required, drivers can utilize one of four techniques based on the amount of required poll time and the context in which the stall occurs.

- Driver code running at `PASSIVE_LEVEL` IRQL can call **KeDelayExecutionThread**, described in [Table 11.3](#), to suspend a thread's execution for a specified interval. The thread is removed from the "ready to run" queue of threads and thus does not interfere with other "good to go" threads.

This technique is available only to kernel-mode threads started by other driver code or during the device's initialization or cleanup code.
- Driver code can "busy wait" the processor using **KeStallExecutionProcessor**, described in [Table 11.4](#). The function call is equivalent to code hanging in a tight countdown loop, except that the time stalled is processor speed-independent. Threads should not stall the processor for more than 50 microseconds.
- Synchronization objects, such as kernel events and mutexes, can be utilized. The "stalling" code then waits on the synchronization object to enter the signaled state—an act for which another nonstalled code path must take responsibility.
- A driver can utilize `CustomTimerDpc` routines to gain the benefit of I/O Timer functionality with controllable timing granularity.

If a device needs to be polled repeatedly, and the delay interval between each polling operation is more than 50 microseconds, the driver design should incorporate system threads (discussed in [Chapter 14](#)).

Table 11.3. Function Prototype for KeDelayExecution Thread

NSTATUS KeDelayExecutionThread	IRQL = PASSIVE_LEVEL
Parameter	Description
IN KPROCESSOR_MODE waitMode	• KernelMode • UserMode
IN BOOLEAN bAlertable	TRUE - if wait is canceled upon receipt of Async Procedure Call FALSE - in kernel mode
IN PLARGE_INTEGER interval	Wait interval in 100 nanosecond units
Return value	Success or failure code

Table 11.4. Function Prototype for KeStallExecutionProcessor

VOID KeStallExecutionProcessor	IRQL = Any Level
Parameter	Description
IN ULONG interval	Wait interval in microseconds
Return value	- void -

Code Example: A Timer-Based Driver

This modified version of the parallel port driver disables interrupts and uses a CustomTimerDpc routine to transfer data at fixed intervals. The code for this example is included on the companion CD and on the companion Web site, <http://www.W2KDriverBook.com>.

Device Extension Additions

The Device Extension is modified to include the DPC and Timer objects, plus a polling interval value, stored in microseconds.

```
typedef struct _DEVICE_EXTENSION {
    ...
    KDPC pollingDPC;    // reserve custom DPC object
    KTIMER pollingTimer; // and the Timer object
    LARGE_INTEGER pollingInterval; // timeout counter
    // in us
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
// Define the interval between polls of device in us
// 100 us
#define POLLING_INTERVAL 100
```

AddDevice Modifications

AddDevice is changed in that **IoConnectInterrupt** is no longer called for this polling driver. The polling timer and DPC are initialized.

```
NTSTATUS AddDevice(IN PDRIVER_OBJECT pDriverObject,
                 IN PDEVICE_OBJECT pdo) {
    PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
        pDevObj->DeviceExtension;
    ...
    // Near the end of the function, after IoCreateDevice
    // has been called...
    //
    // Calculate the polling interval in microseconds
    // and keep as relative time (negative value)
    pDevExt->pollingInterval =
        RtlConvertLongToLargeInteger(
            POLLING_INTERVAL * -10 );
    //
    // Prepare the polling timer and DPC
    KeInitializeTimer( &pDevExt->pollingTimer );
    // Notice that the DPC routine receives the fdo
    KeInitializeDpc( &pDevExt->pollingDPC,
        PollingTimerDpc,
        (PVOID) pfdo );
    ...
}
```

TransmitBytes Changes

Each time a transfer is started using the TransmitBytes routine, the timer is initialized so that the polling timer DPC runs when the polling interval expires. Too large a polling interval does not keep the printer busy; too small an interval needlessly wastes CPU time.

```
BOOLEAN TransmitBytes( IN PDEVICE_EXTENSION pDevExt) {
```

Summary

This chapter presented two distinct methods of dealing with time within a driver. Typically, a driver needs to keep track of time either to ensure the device is working normally or to poll for further device interaction.

The I/O Timer mechanism is simple but crude, generating callbacks at fixed one-second intervals.

CustomTimerDpc routines are used primarily for efficient polling of devices in lieu of device interrupts.

The next chapter deals with DMA operations within the device and its associated driver.

Chapter 12. DMA Drivers

CHAPTER OBJECTIVES

- [How DMA Works under Windows 2000](#)
- [Working with Adapter Objects](#)
- [Writing a Packet-Based Slave DMA Driver](#)
- [Code Example: A Packet-Based Slave DMA Driver](#)
- [Writing a Packet-Based Bus Master DMA Driver](#)
- [Writing a Common Buffer Slave DMA Driver](#)
- [Writing a Common Buffer Bus Master DMA Driver](#)
- [Summary](#)

Not all devices can depend on the CPU to move data between memory and the peripherals. While this technique is fine for slower hardware, fast devices that transfer large amounts of data incur too much overhead. Such devices are usually capable of directly accessing system memory and transferring data without the CPU's intervention. This chapter explains how to write drivers for these kinds of devices.

How DMA Works under Windows 2000

Just as the Windows 2000 operating system abstracts all other pieces of system hardware, DMA operations also follow a strict abstract model. Drivers that perform DMA within the framework of this abstraction can ignore many of the hardware-specific aspects of the system platform. This section presents the major features of the Windows 2000 DMA framework.

Hiding DMA Hardware Variations with Adapter Objects

The purpose of using DMA is to minimize the CPU's involvement in data transfer operations. To do this, DMA devices use an auxiliary processor, called a *DMA controller*, to move data between memory and a peripheral device. This allows the CPU to continue doing other useful work in parallel with the I/O operation.

Although the exact details vary, most DMA controllers have a very similar architecture. In its simplest form, this consists of an address register for the starting address of the DMA buffer and a count register for the number of bytes or words to transfer. When these registers are properly programmed and the device started, the DMA controller begins moving data on its own. With each transfer, it increments the memory address register and decrements the count register. When the count register reaches zero, the DMA controller generates an interrupt, and the device is ready for another transfer.

Unfortunately, the needs of real-world hardware design complicate the simple picture. Consider the DMA implementation on ISA-based machines, described in [chapter 2](#). These systems use a pair of Intel 8237 controller chips cascaded to provide four primary and three secondary DMA data channels. The primary channels (identified as zero through three) can perform single-byte transfers, while the secondary channels (five through seven) always transfer two bytes at a time. Since the 8237 uses a 16-bit transfer counter, the primary and secondary channels can handle only 64 KB and 128 KB per operation, respectively. Due to limitations of the ISA architecture, the DMA buffer must be located in the first 16 megabytes of physical memory.

Contrast this with the DMA architecture used by EISA systems. The Intel 82357 EISA I/O controller extends ISA capabilities by supporting one-, two-, and four-byte transfers on any DMA channel, as well as allowing DMA buffers to be located anywhere in the 32-bit address space. In addition, EISA introduces three new DMA bus-cycle formats (known as types A, B, and C) to give peripheral designers the ability to work with faster devices.

Even on the same ISA or EISA bus, different devices can use different DMA techniques. For example, slave DMA devices compete for shareable system DMA hardware on the motherboard, while bus masters avoid bottlenecks by using their own built-in DMA controllers.

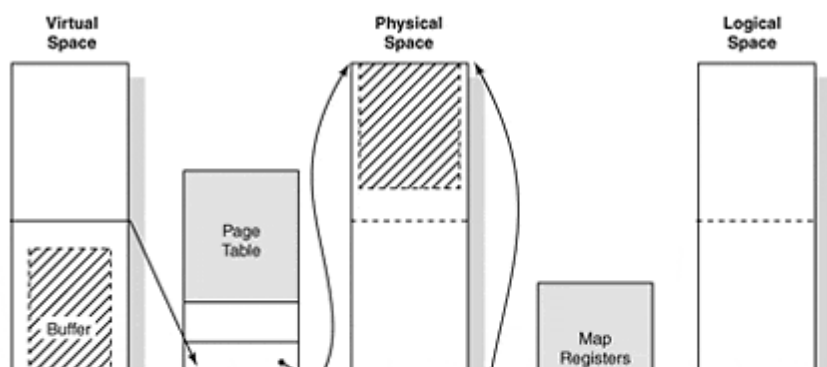
The problem with all this variety is that it tends to make DMA drivers very platform-dependent. To avoid this trap, Windows 2000 drivers don't manipulate DMA hardware directly. Instead, they work with an abstract representation of the hardware in the form of an Adapter object. [Chapter 4](#) briefly introduced these objects and said they help with orderly sharing of system DMA resources. It turns out that Adapter objects also simplify the task of writing platform-independent drivers by hiding many of the details of setting up the DMA hardware. The rest of this section explains more about what Adapter objects do and how to use them in a driver.

The Scatter/Gather Problem

Although virtual memory simplifies the lives of application developers, it introduces two major complications for DMA-based drivers. The first problem is that the buffer address passed to the I/O Manager is a *virtual* address. Since the DMA controller works with *physical* addresses, DMA drivers need some way to determine the physical pages making up a virtual buffer. The next section explains how Memory Descriptor Lists perform this translation.

The other problem (illustrated in [Figure 12.1](#)) is that a process doesn't necessarily occupy consecutive pages of physical memory, and what appears to be a contiguous buffer in virtual space is probably scattered throughout physical memory. The Windows 2000 Virtual Memory Manager uses the platform's address translation hardware (represented by a generic page table in the diagram) to give the process the illusion of a single, unbroken virtual address space. Unfortunately, the DMA controller doesn't participate in this illusion.

Figure 12.1. Address spaces involved in DMA operations.



Working with Adapter Objects

Although the specific details vary with the nature of the device and the architecture of the driver, DMA drivers generally have to perform several kinds of operations on Adapter objects.

-
- Locate the Adapter object associated with the specific device.
-
- Acquire and release ownership of Adapter objects and their mapping registers.
-
- Load the Adapter object's mapping registers at the start of a transfer.
-
- Flush the Adapter object's cache after a transfer completes.

The following sections discuss these topics in general terms. Later sections of this chapter add more detail.

Finding the Right Adapter Object

All DMA drivers need to locate an Adapter object before they can perform any I/O operations. To find the right one, a driver's initialization code needs to call the **IoGetDmaAdapter** function described in [Table 12.2](#).

Given a description of some DMA hardware, *IoGetDmaAdapter* returns a pointer to a structure of function pointers that manipulate the corresponding Adapter object. It also reports a count of the maximum number of mapping registers available for a single transfer. The driver needs to save both these items in nonpaged storage (usually the Device or Controller Extension) for later use.

Table 12.2. Function Prototype for IoGetDmaAdapter

PDMA_ADAPTER	IoGetDmaAdapter IRQL = PASSIVE_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pdo	Points to the physical device object for the device
IN PDEVICE_DESCRIPTION pDeviceDescription	Points to a DEVICE_DESCRIPTION structure
IN OUT PULONG	Input: number of map registers requested.
NumberOfMapRegisters	Output: maximum number of map registers that the driver can allocate for any DMA transfer operation.
Return value	Pointer to DMA_ADAPTER

By returning a structure of function pointers, the Adapter object is truly an encapsulated object—it can only be manipulated through its interface.

The main input to **IoGetDmaAdapter** is the DEVICE_DESCRIPTION block listed in [Table 12.3](#). Unused entries of this input structure must be zero. Some fields of this structure deserve comment.

ScatterGather.

For bus master devices, this field signifies that the hardware supports transfer of data to and from noncontiguous ranges of physical memory. For slave devices, this field indicates that the device can be paused between page transfers, allowing the I/O Manager to repoint the DMA channel's address register to a new page of physical memory.

DemandMode.

DMA demand mode is a transfer protocol that allows a device to hold off the (slave) DMA controller. In normal mode (DemandMode==FALSE), the DMA controller does not allow the device to delay its request to transfer another block of data.

Autoinitialization.

DMA autoinitialization mode allows system DMA channels to restart themselves after a completed transfer. Specified address and count values are automatically reset into the DMA hardware and another operation is "good to go."

IgnoreCount.

Some DMA hardware maintains an improper count of bytes transferred. This can occur because the hardware counts words instead of bytes. If this field is set to TRUE, the HAL manually maintains the transfer count on behalf of the device.

Writing a Packet-Based Slave DMA Driver

In packet-based slave DMA, the device transfers data to or from the locked down pages of the caller's buffer using a shared DMA controller on the mainboard. The system is also responsible for providing scatter/gather support.

How Packet-Based Slave DMA Works

Although the specifics depend on the nature of the device, most packet-based slave DMA drivers conform to a very similar pattern. The following subsections describe the routines of these drivers.

IRP_MN_START_DEVICE Handler

Along with its usual duties, this PnP handler performs the following DMA preparation tasks:

1.
 - Locates the DMA channel used by the device. The DMA resources would normally be sent with the requesting IRP in the stack's `Parameters.StartDevice.AllocatedResourcesTranslated` field.
2.
 - The `DEVICE_DESCRIPTION` structure is built. **IoGetDmaAdapter** is invoked to identify the Adapter object associated with the device.
3.
 - The `DMA_OBJECT` pointer returned from **IoGetDmaAdapter** is saved in the Device Extension.
4.
 - The `DO_DIRECT_IO` bit in the **Flags** field of the Device object is set, causing the I/O Manager to lock user buffers in memory and create MDLs for them.

START I/O ROUTINE

Unlike its counterpart in a programmed I/O driver, the DMA Start I/O routine doesn't actually start the device. Instead, it requests ownership of the Adapter object and leaves the remainder of the work to the Adapter Control callback routine. Specifically, the Start I/O routine does the following:

1.
 - It calls **KeFlushIoBuffers** to flush data from the CPU cache out to main memory (RAM).
2.
 - Start I/O decides how many mapping registers to request. Initially, it calculates the number of registers needed to cover the entire user buffer. If this number turns out to be more mapping registers than the Adapter object has available, it will ask for the maximum available.
3.
 - Based on the number of mapping registers and the size of the user buffer, Start I/O calculates the number of bytes to transfer in the first device operation. This may be the entire buffer or it may be only the first portion of a split transfer.
4.
 - Next, Start I/O calls **MmGetMdlVirtualAddress** to recover the virtual address of the user buffer from the MDL. It stores this address in the Device Extension. Subsequent parts of the driver use this address as an offset in the MDL to set up the actual DMA transfer.
5.
 - Start I/O then calls **AllocateAdapterChannel** to request ownership of the Adapter object. If this function succeeds, the rest of the setup work is performed by the Adapter Control routine, so Start I/O simply returns control to the I/O Manager.
6.
 - If **AllocateAdapterChannel** returns an error, Start I/O fails the requesting IRP, calls **IoCompleteRequest**, and starts processing the next IRP.

ADAPTER CONTROL ROUTINE

The I/O Manager calls back the Adapter Control routine when the necessary Adapter resources become available. Its job is to initialize the DMA controller for the transfer and start the device itself. In essence, it is the second half of Start I/O that occurs with the Adapter object in hand. This routine does the following:

Code Example: A Packet-Based Slave DMA Driver

This example is a skeleton of a packet-based driver for a generic slave DMA device. Although it doesn't actually manage a specific kind of hard-ware, it may help in understanding how these drivers work. The complete code for this example is included on the CD that accompanies this book and on the companion website www.W2KDriverBook.com.

DRIVER.H

This excerpt from the driver-specific header file shows the changes that need to be made to the Device Extension structure.

```
typedef struct _DEVICE_EXTENSION {
    ...
    PDMA_ADAPTER pDmaAdapter;
    ULONG mapRegisterCount;
    ULONG dmaChannel;

    // This is the "handle" assigned to the map registers
    // when the AdapterControl routine is called back
    PVOID mapRegisterBase;

    ULONG bytesRequested;
    ULONG bytesRemaining;
    ULONG transferSize;
    PCHAR transferVA;

    // This flag is TRUE if writing, FALSE if reading
    BOOLEAN bWriting;
    ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

#define MAX_DMA_LENGTH 4096
```

GetDmaInfo Routine

The GetDmaInfo helper routine is responsible for making the call to **IoGetDmaAdapter**. Primarily, this is an example of setting up the DEVICE_DESCRIPTION structure.

```
NTSTATUS GetDmaInfo( IN INTERFACE_TYPE busType,
                   IN PDEVICE_OBJECT pDevObj ) {
    PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
        pDevObj->DeviceExtension;

    DEVICE_DESCRIPTION dd;

    // Zero out the entire structure
    RtlZeroMemory( &dd, sizeof(dd) );

    dd.Version = DEVICE_DESCRIPTION_VERSION1;
    dd.Master = FALSE; // this is a slave device
    dd.ScatterGather = FALSE;
    dd.DemandMode = FALSE;
    dd.AutoInitialize = FALSE;
    dd.Dma32BitAddresses = FALSE;

    dd.InterfaceType = busType;           // as passed in

    dd.DmaChannel = pDevExt->dmaChannel;
    dd.MaximumLength = MAX_DMA_LENGTH;
    dd.DmaWidth = Width16Bits;
    dd.DmaSpeed = Compatible;
```

Writing a Packet-Based Bus Master DMA Driver

In packet-based bus master DMA, the device transfers data to or from the locked down pages of the caller's buffer, using DMA hardware that is part of the device itself. Depending on the capabilities of the device, it might be providing its own scatter/gather support as well.

The architecture of a packet-based bus master driver is almost identical to that of a driver for a slave device. The only difference is the way the driver sets up the bus master hardware. The following sections describe these differences.

Setting Up Bus Master Hardware

A bus master device is more complicated to program because Windows 2000 doesn't know how to program the device's onboard DMA controller. The most the I/O Manager can do is provide the driver with two pieces of information.

-
- An address in DMA logical space, where a contiguous segment of the buffer begins
-
- A count indicating the number of bytes in that segment

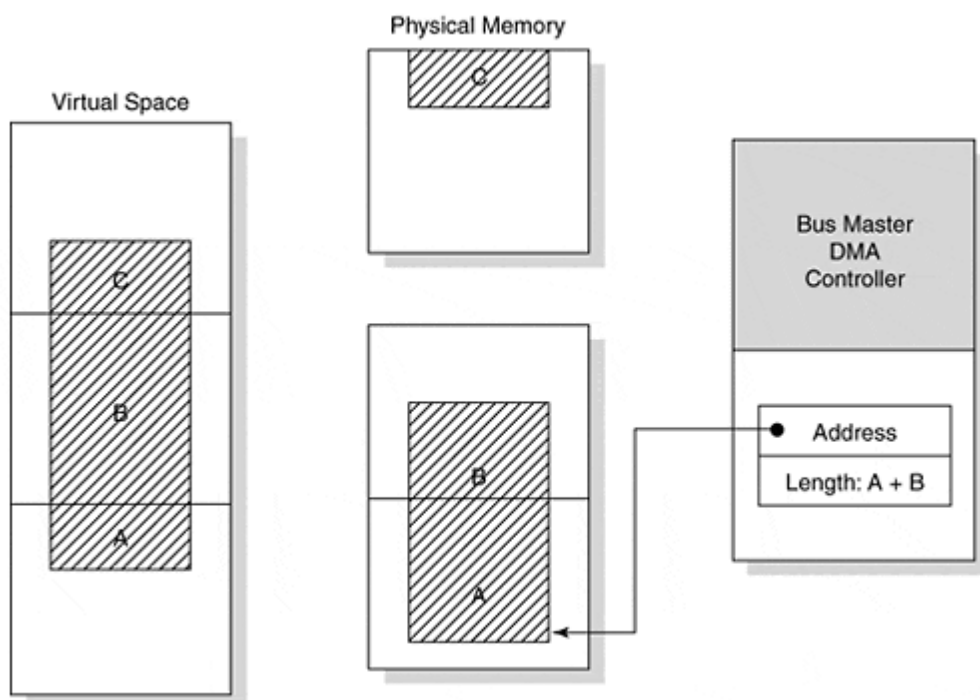
It then becomes the driver's responsibility to load this information into the address and length registers of the device and start the transfer.

The function that performs this work is familiar—**MapTransfer**. If the `DEVICE_DESCRIPTION` supplied in the call to **IoGetDmaAdapter** described a bus mastering device, then **MapTransfer** returns a physical address that can be used to program the device's DMA address register. The physical address corresponds to the MDL and `CurrentVa` arguments of the call.

There is little chance that the user buffer, if it spans more than a page, resides in contiguous physical RAM. Therefore, **MapTransfer** returns the physical *starting* address and the length of the "first" block that can be transferred contiguously. In all likelihood, the length returned by **MapTransfer** is the length of a page—no part of the user buffer is likely to be contiguous. Should two or more pages of the user buffer *happen* to be contiguous, then **MapTransfer** correctly reports such a length.

After the device transfers what it can from the contiguous buffer, it must be reprogrammed to transfer the next part of the buffer from another contiguous block. [Figure 12.4](#) shows how this process needs to work, with the somewhat optimistic case of some pages of the user buffer actually residing in contiguous memory.

Figure 12.4. *MapTransfer points to contiguous buffer segments.*



Supporting bus master devices requires some changes to the driver's `AdapterControl` and `DpcForIsr` routines. The following sections contain fragments of these routines. Compare them with the corresponding routines in the packet-based slave DMA driver in the previous section of this chapter.

AdapterControl ROUTINE

Writing a Common Buffer Slave DMA Driver

In common buffer slave DMA, the device transfers data to or from a contiguous buffer in nonpaged pool, using a system DMA channel. Although originally intended for devices that use the system DMA controller's autoinitialize mode, common buffers can also improve throughput for some types of ISA-based slave devices.

Allocating a Common Buffer

Memory for a common buffer must be physically contiguous and visible in the DMA logical space of a specific device. To guarantee that both these conditions are met, the function **AllocateCommonBuffer**, a method of the Adapter object, is used. It is described in [Table 12.8](#).

The **CacheEnabled** argument for this function is normally set to **FALSE**. Using noncached memory for the common buffer eliminates the need to call **KeFlushIoBuffers**. On some platforms, this can significantly improve the performance of both the driver and the system.

Table 12.8. Function Prototype for **AllocateCommonBuffer**

PVOID AllocateCommonBuffer	IRQL = PASSIVE_LEVEL
Parameter	Description
IN PDMA_ADAPTER pDmaAdapter	Points to the DMA_ADAPTER structure returned by IoGetDmaAdapter
IN ULONG Length	Requested size of buffer in bytes
OUT PPHYSICAL_ADDRESS LogicalAddress	Address of the common buffer in the DMA controller's logical space
IN BOOLEAN CacheEnabled	• TRUE - memory is cacheable • FALSE - memory is not cached
Return value	• Non-NULL - VA of common buffer • FALSE - error

Besides allocating the common buffer, **AllocateCommonBuffer** also allocates map registers (if required) and sets up a translation for the device, loading map registers as necessary. Thus, the buffer is available for immediate and continuous use. The buffer remains usable until **FreeCommonBuffer** is explicitly invoked, typically in the handler routine for **IRP_MN_STOP_DEVICE**.

Using Common Buffer Slave DMA to Maintain Throughput

Common buffer slave DMA is useful if the driver can't afford to set up mapping registers for each data transfer that it performs. The overhead of setting up mapping registers using **MapTransfer** can be significant, especially for ISA buses. There is always the possibility that **MapTransfer** will be forced to copy the transfer buffer into the lowest 16 MB of RAM—clearly an expensive proposition. Since common buffers are guaranteed to be accessible by their associated DMA devices, there is never a need to move data from one place to another.

For example, drivers of some ISA-based tape drives need to maintain very high throughput if they want to keep the tape streaming. They may not be able to do this if the buffer copy must occur during a call to **MapTransfer**. To prevent this, the driver can use a ring of common buffers for the actual DMA operation. Other, less time-critical portions of the driver move data between these common buffers and the actual user buffers.

Consider the operation of the driver for a hypothetical ISA output device. To maintain a high DMA data rate, it uses a series of common buffers that are shared between the driver's **Dispatch** and **DpcForIsr** routines. The **Dispatch** routine copies user-output data into an available common buffer and attaches the buffer to a queue of pending DMA requests. Once a DMA is in progress, the **DpcForIsr** removes buffers from the queue and processes them as fast as it can. [Figure 12.6](#) shows the organization of this driver, and the various driver routines are described in the sections that follow.

Figure 12.6. Using common buffers to improve I/O throughput.



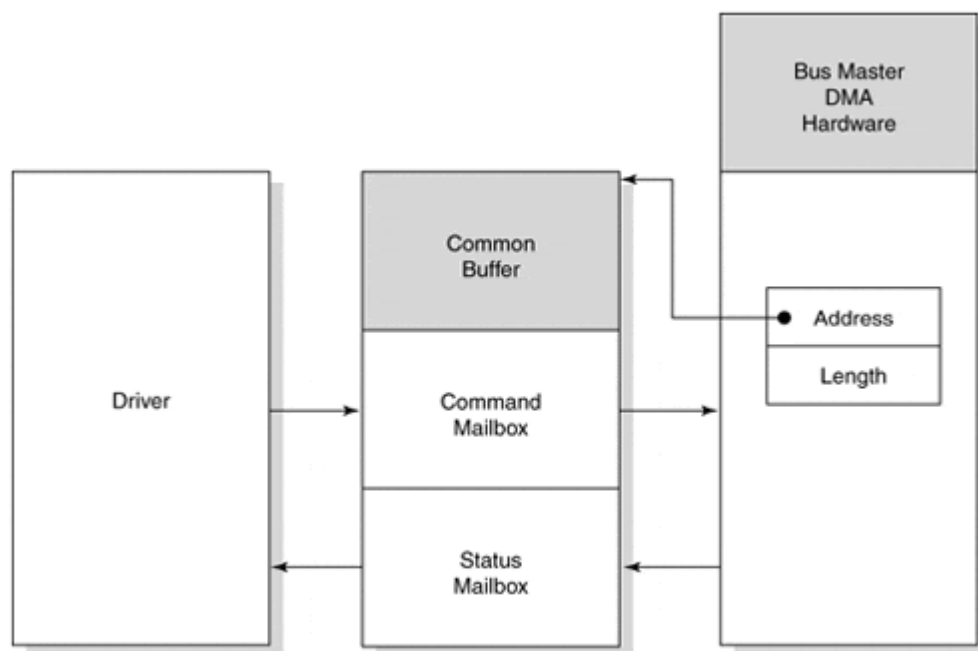
Writing a Common Buffer Bus Master DMA Driver

In common-buffer bus master DMA, the device transfers data to or from a contiguous nonpaged pool buffer, using a DMA controller that is part of the device itself. Frequently, this kind of hardware treats the common buffer as a mailbox for exchanging control and status messages with the driver.

How Common-Buffer Bus Master DMA Works

The exact operation of a common-buffer bus master driver will depend on the physical device. The description that follows is based on the typical architecture. It assumes the device uses one mailbox for commands and another to return status information. [Figure 12.7](#) illustrates this arrangement.

Figure 12.7. Common buffer implementation of message exchange



IRP_MN_START_DEVICE HANDLER

This handler must now perform the following tasks:

1. Invoke **IoGetDmaAdapter** to locate the Adapter object associated with the device.
2. Use **AllocateCommonBuffer** to get a block of contiguous, nonpaged memory that both the driver and the device can access.
3. Store the virtual address of the common buffer in the Device Extension for subsequent use.

START I/O ROUTINE

To send a command to the device, the Start I/O routine performs the following:

1. Builds a command structure in the common buffer, using the virtual address stored in the Device Extension.
2. If needed (CacheEnabled parameter set to TRUE), **KeFlushIoBuffers** is invoked to force data from the CPU's cache out to physical memory.
3. Finally, Start I/O sets a bit in the device control register to notify the device that there is a command waiting for it. In essence, this is equivalent to starting the device.

Summary

Without a doubt, drivers for DMA devices are more complicated than drivers for programmed I/O hardware. In return for this added complexity, the system achieves greater throughput by overlapping CPU activity with data transfers. The I/O Manager tries to simplify the process by providing a generic framework in which to perform DMA. This chapter has presented the details of the Windows 2000 abstract DMA model and discussed the various styles of DMA.

The next chapter begins the discussion of how devices report and log their operational status to the operating system.

Chapter 13. Windows Management and Instrumentation

CHAPTER OBJECTIVES

- [WMI: The Industry Picture](#)
- [The WMI Architecture](#)
- [WMI Summary](#)
- [Conventional Driver Event Logging](#)
- [Summary](#)

Device drivers can be written to report their performance and reliability statistics. For example, it is useful for a network driver to track the number of corrupted packets received. Devices that start to perform erratically should be identified before they fail completely.

Overall device and driver performance rates are vital pieces of information to analyze and improve system performance. Tracking and reporting such instrumentation data begins at the driver level.

Another separate but common requirement is that devices and drivers need a mechanism for dynamic configuration. Traditionally, drivers use Control Panel applets and private IOCTLs to provide this facility.

The WDM driver model includes the capability for standardizing the collection, storage, and reporting of instrumentation data—the Windows Management and Instrumentation (WMI) interface. Drivers that choose to participate in the WMI scheme can collect performance and reliability data in a standardized way for applications to obtain.

The facility also provides a standardized way that clients can use to set a driver state for the purpose of device configuration. A driver can export specialized methods that client applications can invoke (almost) directly.

This chapter starts by describing the relationship of WMI to industry standards for similar functionality. The steps necessary to incorporate WMI into a driver are then presented along with an example of a WMI-enabled driver. Finally, the chapter covers the conventional mechanism of sending event messages to the Windows 2000 event log.

WMI: The Industry Picture

The need for system instrumentation and configuration is not unique to device drivers, Windows 2000, or Microsoft. The Open Software Foundation (OSF) includes a task group, the Desktop Management Taskforce (DMTF), that has published an initiative for unified system management compatible with Internet standards, Web-Based Enterprise Management (WBEM).

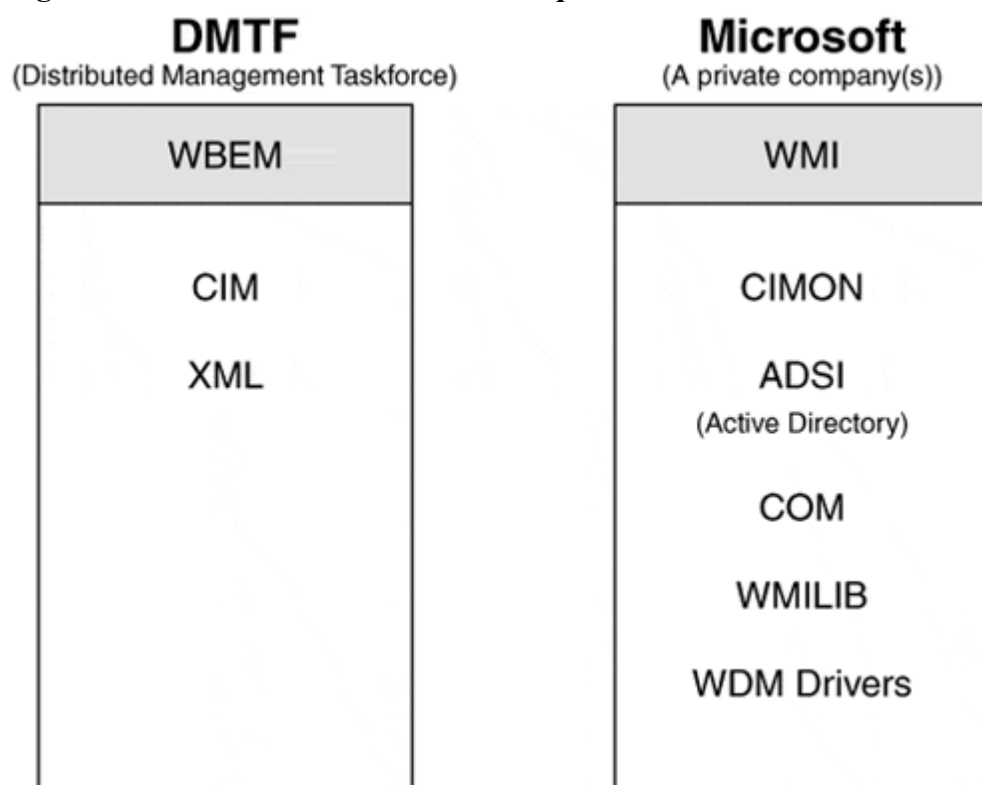
WBEM defines an interface for the purpose of unified management and instrumentation: the Common Information Model (CIM). CIM is designed to encapsulate and accommodate other industry initiatives such as the Simple Network Management Protocol (SNMP) and the Desktop Management Interface (DMI).

CIM is a very general-purpose way of describing properties, methods, and events that clients can use to efficiently obtain system-supplied data. WMI is Microsoft's implementation of WBEM. Therefore, it is based on CIM and extends itself to components within Windows 2000 to provide a functional environment for management and instrumentation.

CIM defines a new language (actually an extension of C) called MOF (Managed Object Format). MOF is loosely based on the Interface Definition Language (IDL) familiar to users of Remote Procedure Calls (RPCs) and Microsoft COM (Component Object Model).

A diagram that describes the relationship between WBEM and WMI is shown in [Figure 13.1](#).

Figure 13.1. WBEM and WMI relationship.



Acronym overload aside, MOF is the language by which the provider and the consumer can precisely communicate and exchange data. For example, a data definition within the MOF file to describe the total packet count transferred by a driver might appear as follows:

```
[read, WmiDataId(1),
    Description("Total Packet Count")]
uint64 PacketsTransferred;
```

A client application, having been built with knowledge of the MOF contents, is now aware that the packet count is available from the provider as a 64-bit unsigned integer.

The advantage of conformance with industry standards for system management and instrumentation is obvious: heterogeneous system integration. With sufficient industry acceptance and conformance, Microsoft-based, Linux, and mainframe systems can cooperate in a peer-to-peer configuration, management, and reporting environment.

The WMI Architecture

The WMI architecture consists of five layers:

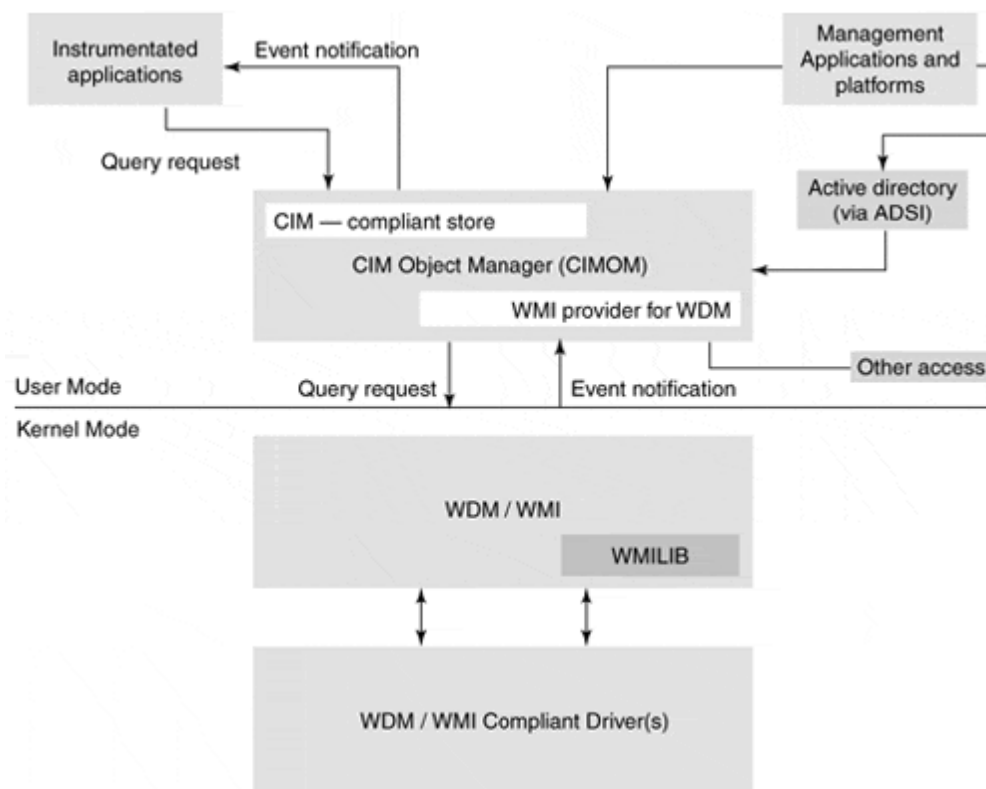
-
- A source of WMI data (e.g., a device driver) designated the *data provider*
-
- A central repository of WMI data that implements a CIM-compliant data store, based on the CIM object model (CIMOM)
-
- A WMI provider (e.g., the WDM WMI provider) that provides the necessary linkage between a WMI data source and the data store
-
- A consumer (or client) of the WMI data
-
- A protocol (e.g., COM) that presents the data store to a consumer

The WMI provider for WDM consists of two parts:

-
- A user-mode Win32 DLL that provides linkage between a user-mode client
-
- A set of kernel-mode services that invoke and respond to driver code. An IRP, IRP_MJ_SYSTEM_CONTROL, is the mechanism by which these services make requests of the driver for WMI data.

A diagram of the WMI architecture is shown in [Figure 13.2](#).

Figure 13.2. WMI architecture.



Providing WMI Support in a WDM Driver

In order to supply WMI data, a driver must perform the following steps:

- 1.

WMI Summary

Clearly, the implementation of WMI can be involved. At this point in the evolution of WBEM and WMI, the payback is minimal. The addition of WMI support to a driver is a plan for the future. The ultimate acceptance of WMI (and WBEM in general) remains to be seen.

A WDM driver is not required to support WMI. It can use the conventional method of logging system events for notification messages, which is described in the next section. For data collection (instrumentation), the registry or custom IOCTLs can be utilized.

Conventional Driver Event Logging

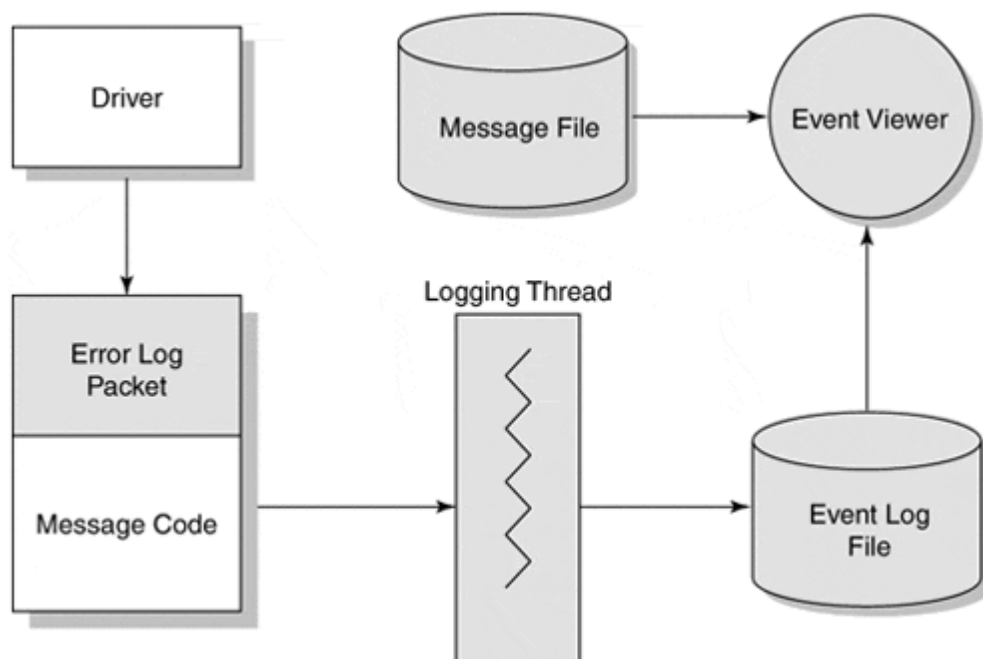
The original Windows NT architecture includes a mechanism that allows software components to keep a record of noteworthy events. This event-logging capability can help monitor the behavior of software that is under development or in production.

How Event Logging Works

The developers of Windows NT had several goals for the event-logging architecture. The first was to provide subsystems with the unified framework for recording information. This framework includes a simple yet flexible standard for the binary format of event-logging entries.

Another goal was to give system administrators an easy and consistent way to view these messages. As part of this goal, viewer utilities must be able to display event messages in the currently selected national language. [Figure 13.3](#) shows the event-logging architecture.

Figure 13.3. *Event-logging architecture.*



The overall process for the generation and display of event messages is described by the following:

1.

All event messages take the form of packets in Windows 2000. When a kernel-mode driver wants to log an event, it first calls the I/O Manager to allocate a message packet from nonpaged pool.

2.

The driver fills this packet with various pieces of descriptive information. One of the key items is a 32-bit message code number that identifies the text to be displayed for this packet. Once the packet is ready, the driver returns it to the I/O Manager.

3.

The I/O Manager takes the message packet and sends it to the system event-logging thread. This thread accumulates packets and periodically writes them to the proper event-log file.

4.

The Event Viewer utility reads binary packets from the log files. To translate a packet's 32-bit message code into text, the Viewer goes to the Registry. There it finds the path names of one or more message files associated with the packet. These message files contain the actual message text (possibly in multiple languages), which the Viewer displays.

Working with Messages

As just described, a driver does not include the actual text for its messages in an event-log entry. Instead, it identifies messages using code numbers. The text associated with these code numbers takes the form of a message resource stored somewhere on disk. This section describes how these message code numbers work and explains how to generate custom message resources.

The code number identifying a specific message is a 32-bit value consisting of several fields. [Figure 13.4](#) shows the layout of a

Summary

The incorporation of WMI and event logging into a driver is tedious, but it is inconceivable that a production driver can be supplied without some system reporting mechanism. The ability to trace back through events after a failure report is essential to discover true faults.

Code examples are included on the CD that accompanies this book and on the book's Web site: *www.W2kDriveBook.com*; they demonstrate the use of WMI and event logging.

The next chapter deals with system threads and their appropriate use. The next chapter deals with system threads and their appropriate use.

Chapter 14. System Threads

CHAPTER OBJECTIVES

- [Definition and Use of System Threads](#)
- [Thread Synchronization](#)
- [Using Dispatcher Objects](#)
- [Code Example: A Thread-Base Driver](#)
- [Summary](#)

The work that must be performed within a driver cannot always be done in response to a request—at least, not at the very moment of the request. Some work must be performed asynchronous to a caller, perhaps at a different priority to other driver activity. Windows 2000 allows the creation of separate threads of execution, each following a code path that is independent of others. The threads perform units of work at their own pace, triggered by appropriate events.

This chapter explains the process of creating a kernel-mode thread. It also explains the synchronization techniques provided by the kernel between different threads.

Definition and Use of System Threads

A thread is a *unit of execution*. Each thread maintains an independent program counter and hardware context that includes a private set of CPU registers. Each thread maintains a *priority* value that determines when it gains control of the system processor(s). In general, the higher a thread's priority, the more likely it is to receive control.

Threads can operate in user mode or kernel mode. A system thread is one that runs exclusively in kernel mode. It has no user-mode context and cannot access user address space. Just like a Win32 thread, a system thread executes at or below APC_LEVEL IRQL and it competes for use of the CPU based on its scheduling priority.

When To Use Threads

There are several reasons to use threads in a driver. One example is working with hardware that has the following characteristics:

-
- The device is slow and infrequently accessed.
-
- It takes a long time (more than 50 microseconds) for the device to make a state transition, and the driver must wait for the transition to occur.
-
- The device needs to make several state transitions in order to complete a single operation.
-
- The device does not generate interrupts for some state transitions, and the driver must poll the device for an extended period.

Such a device could be managed using a CustomTimerDpc routine. Depending on the amount of device activity, however, this approach could saturate the DPC queues and slow down other drivers. Threads, on the other hand, run at PASSIVE_LEVEL and do not interfere with DPC routines.

Fortunately, most modern hardware is designed to operate in a way that allows good system performance. (Granted, there are many contradictions to this statement.) Legacy hardware, however, is legendary in forcing drivers to poll (and retry) for state transitions, thus burdening the driver with considerable and messy design. The most notable examples are floppy disks and other devices attached to floppy controllers.

Another need for threads occurs with devices that take excessive time to initialize. The driver must monitor the initialization process, polling the state transitions throughout. A separate thread is needed because the Service Control Manager gives drivers only about 30 seconds to execute their DriverEntry routine. Otherwise, the Service Control Manager forcibly unloads the driver. The only solution is to put the long-running device start-up code in a separate thread and return promptly from the DriverEntry routine with STATUS_SUCCESS.

Finally, there may be some operations that can be performed only at PASSIVE_LEVEL IRQL. For example, if a driver has to access the registry on a regular basis, or perform file operations, a multi-thread design should be considered.

Creating and Terminating System Threads

To create a system thread, a driver uses **PsCreateSystemThread**, described in [Table 14.1](#). Since this function can only be called at PASSIVE_LEVEL IRQL, threads are usually created in the DriverEntry or AddDevice routines.

When a driver unloads, it must ensure that any system thread it may have created has terminated. System threads must terminate themselves, using **PsTerminateSystemThread**, described in [Table 14.2](#). Unlike Win32 user-mode threads, there is no way to forcibly terminate a system thread. This means that some kind of signaling mechanism needs to be set up to let a thread know it should exit. As discussed later in this chapter, Event objects provide a convenient mechanism for this.

Table 14.1. PsCreateSystemThread Function Prototype

NTSTATUS PsCreateSystemThread	IRQL = PASSIVE_LEVEL
Parameter	Description
OUT PHANDLE ThreadHandle	Handle of new thread
IN ULONG DesiredAccess	0 for a driver-created thread
IN POBJECT_ATTRIBUTES Attrib	NULL for a driver-created thread
IN HANDLE ProcessHandle	NULL for a driver-created thread
OUT PCLIENT_ID ClientId	NULL for a driver-created thread

Thread Synchronization

Like user-mode threads in a Win32 application, system threads may need to suspend their execution until some other condition has been satisfied. This section describes the basic synchronization techniques available to system threads.

Time Synchronization

The simplest kind of synchronization involves stopping a thread's execution until a specific time interval elapses. Although Timer objects, described later in this chapter, can be used, the kernel provides a convenient function, `KeDelayExecutionThread` (illustrated in [Table 14.3](#)) that is easier to use.

Table 14.3. `KeDelayExecutionThread` Function Prototype

NTSTATUS <code>KeDelayExecutionThread</code>	IRQL = PASSIVE_LEVEL
Parameter	Description
IN KPROCESSOR_MODE <code>WaitMode</code>	KernelMode for drivers
IN BOOLEAN <code>bAlertable</code>	FALSE for drivers
IN PLARGE_INTEGER <code>Interval</code>	Absolute or relative due time
Return value	STATUS_SUCCESS: wait completed

General Synchronization

System threads can synchronize their activities in more general ways by waiting for *dispatcher objects*. Thread synchronization depends on the fact that a dispatcher object is always in either the *signaled* or *nonsignaled* state. When a thread asks to wait for a nonsignaled dispatcher object, the thread's execution stops until the object becomes signaled. (Waiting for a dispatcher object that is already Signaled is a no-op.) There are two different functions that can be used to wait for a dispatcher object.

KeWaitForSingleObject

This function, described in [Table 14.4](#), puts the calling thread into a wait state until a specific dispatcher object is set to the signaled state.

Table 14.4. `KeWaitForSingleObject` Function Prototype

NTSTATUS <code>KeWaitForSingleObject</code>	
Parameter	Description
IN PVOID <code>Object</code>	Pointer to an initialized dispatcher object
IN KWAIT_REASON <code>Reason</code>	Executive for drivers
IN KPROCESSOR_MODE <code>WaitMode</code>	KernelMode for drivers
IN BOOLEAN <code>Alertable</code>	False for drivers
IN PLARGE_INTEGER <code>Timeout</code>	• Absolute or relative timeout value • NULL for an infinite wait
Return value	• STATUS_SUCCESS • STATUS_ALERTED • STATUS_TIMEOUT

Optionally, a timeout value may be specified that causes the thread to awaken even if the dispatcher object is nonsignaled. If the timeout argument is NULL, `KeWaitForSingleObject` waits indefinitely.

KeWaitForMultipleObjects

This function, described in [Table 14.5](#), puts the calling thread into a wait state until any or all of a group of dispatcher objects are set to the signaled state. Again, a timeout value for the wait may be specified.

Table 14.5. `KeWaitForMultipleObjects` Function Prototype

NTSTATUS <code>KeWaitForMultipleObjects</code>	
Parameter	Description
IN ULONG <code>Count</code>	Number of objects to wait for
IN PVOID <code>Object []</code>	Array of pointers to dispatcher objects
IN WAIT_TYPE <code>WaitType</code>	• WaitAll: wait until all are signaled • WaitAny: wait until one is signaled
IN KWAIT_REASON <code>Reason</code>	Executive for drivers

Using Dispatcher Objects

Except for Thread objects themselves, the driver must allocate storage for any dispatcher objects that are used. The objects must be permanently resident and are, therefore, usually allocated within the Device or Controller Extension. In any case, they must be in nonpaged memory.

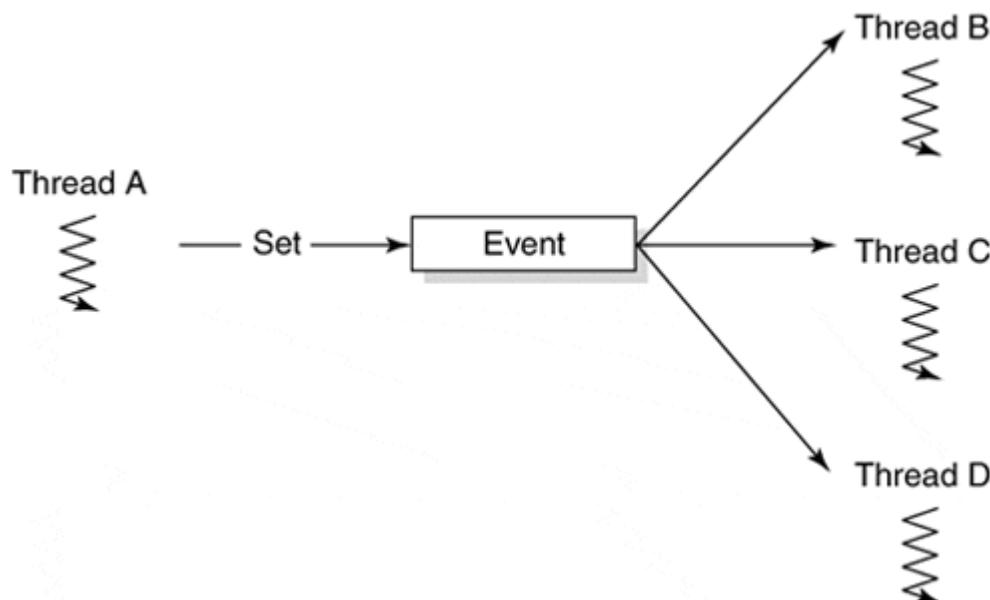
Also, the dispatch object must be initialized once with the proper **KeInitializeXxx** function before it is used. Since the initialization functions can only be called at PASSIVE_LEVEL IRQL, dispatcher objects are usually prepared in the DriverEntry or AddDevice routine.

The following sections describe each category of dispatcher objects in greater detail.

Event Objects

An event is a dispatcher object that must be explicitly set to the signaled or nonsignaled state. An event is analogous to a binary flag, allowing one thread to signal other threads of a specific occurrence by raising (set to signaled) the flag. This behavior can be seen in [Figure 14.1](#), where thread A awakens B, C, and D by setting an event object.

Figure 14.1. *Event objects synchronize system threads.*



These objects actually come in two different flavors: notification events and synchronization events. The type is chosen when the object is initialized. These two types of events exhibit different behavior when put into the signaled state. As long as a notification event remains signaled, all threads waiting for the event come out of their wait state. A notification event must be explicitly reset to put it into the nonsignaled state. These events exhibit behavior like user-mode (Win32) manual-reset events.

When a synchronization event is placed into the signaled state, it remains set only long enough for one call to **KeWaitForXxx**. It then resets itself to the nonsignaled state automatically. In other words, the gate stays open until exactly one thread passes through, and then it shuts. This is equivalent to a user-mode auto-reset event.

To use an event, storage is first allocated for an item of type KEVENT, and then functions listed in [Table 14.6](#) are called.

Notice that either of two functions put an event object into the nonsignaled state. The difference is that **KeResetEvent** returns the state of the event before it became nonsignaled, and **KeClearEvent** does not. **KeClearEvent** is somewhat faster, so it should be used unless the previous state must be determined.

The sample driver at the end of this chapter provides an example of using events. It has a worker thread that needs to pause until an interrupt arrives, so the thread waits for an event object. The driver's DpcForIsr routine sets the event into the signaled state, waking up the worker thread.

Table 14.6. Functions that Operate on Event Objects

How to use event objects		
If you want to	THEN call	IRQL
Create an event	KeInitializeEvent	PASSIVE_LEVEL
Create a named event	IoCreateSynchronizationEvent	PASSIVE_LEVEL
	IoCreateNotificationEvent	

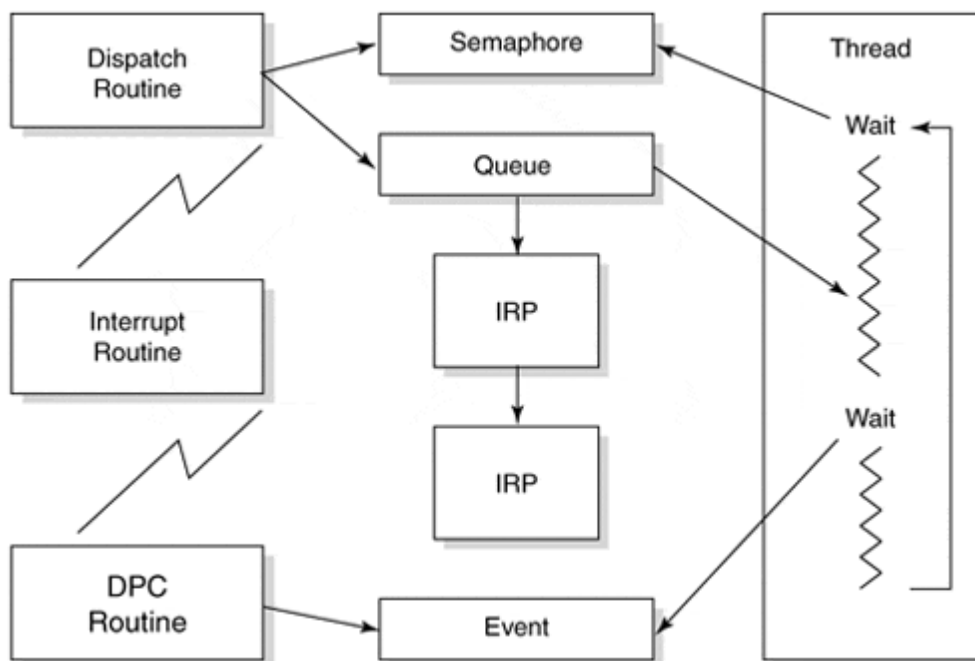
Code Example: A Thread-Based Driver

This section presents a modified version for the packet-based slave DMA driver introduced in [Chapter 12](#). What is different about this driver is that it uses a system thread to do most of the I/O processing. As a result, it spends very little time at DISPATCH_LEVEL IRQL and does not interfere as much with other system components. Code examples can be found on the CD that accompanies this book or on the book's Web site: <http://www.W2KDriverBook.com>.

How the Driver Works

[Figure 14.6](#) gives a high-level view of the sample driver architecture. One of the first things to notice is that the driver has no Start I/O routine. When a user-mode I/O request arrives, one of the driver's Dispatch routines simply adds the IRP to a work queue associated with the Device object. Then the Dispatch routine calls **KeReleaseSemaphore** to increment a Semaphore object that keeps track of the number of IRPs in the work queue. A nonzero Semaphore count indicates the number of IRPs within the work queue yet to be processed.

Figure 14.6. Architecture of a thread-based DMA driver.



Each Device object has its own system thread that processes these I/O requests. This thread is in an endless loop that begins with a call to **KeWaitForSingleObject** on the Semaphore. If the Semaphore object has a nonzero count, the thread removes an IRP from the work queue and performs the I/O operation. On the other hand, if the count is zero the thread goes into a wait state until the Dispatch routine inserts another IRP in the queue.

When the thread needs to perform a data transfer, it starts the device and then uses **KeWaitForSingleObject** to wait for an Event object. The driver's DpcForIsr routine sets this Event into the signaled state after an interrupt arrives. The Event object effectively synchronizes the interrupt service code (actually the DPC) with the worker thread that dequeues IRPs.

When the driver's RemoveDevice routine needs to kill the system thread, it sets a flag in the Device Extension and increments the Semaphore object. If the thread was asleep waiting for the Semaphore object, it wakes up, sees the flag, and terminates itself. If it is in the middle of an I/O operation, it won't see the flag until it completes the current IRP.

The DEVICE_EXTENSION Structure

This file contains all the usual driver-defined data structures. The following excerpt shows only additional fields that the driver needs in order to manage the system thread and its work queue. Other fields are identical to those in the packet-based slave DMA example of [chapter 12](#).

```
typedef struct _DEVICE_EXTENSION {
    ...
    // Pointer to worker thread object
    PETHREAD pThreadObj;

    // Flag set to TRUE when worker thread should quit
    BOOLEAN bThreadShouldStop;
}
```

Summary

This chapter presented the use of system threads within drivers to accomplish specific tasks in parallel with others. Although not all drivers should utilize a multithreaded architecture, there are several scenarios where the technique is appropriate.

The next chapter deals with layering drivers to accomplish organization and reuse of code.

Chapter 15. Layered Drivers

CHAPTER OBJECTIVES

- [An Overview of Intermediate Driver](#)
- [Writing Layered Drivers](#)
- [Writing I/O Completion Rules](#)
- [Allocating Additional IRPs](#)
- [Writing Filter Drivers](#)
- [Code Example: A Filter Driver](#)
- [Writing Tightly Coupled Drivers](#)
- [Summary](#)

The ability to break a large unit of work into several smaller pieces is the cornerstone of software development. The Windows 2000 device driver model supports this necessary abstraction by allowing the work of a driver to be implemented in multiple layers. Besides simplifying the overall job of writing a driver, this allows different vendors to supply different parts of the implementation.

Indeed, the WDM driver model is based upon layering *functional* drivers on top of *physical* drivers, with optional *filter* drivers surrounding the functional layer. In most cases, a physical driver (that interacts with a standard bus) is never written. And in many cases, only a filter driver must be written to affect the desired behavior from an existing functional driver.

This chapter describes the details of layering a driver design into a hierarchy.

An Overview of Intermediate Drivers

Before moving into the intricacies of the implementation of intermediate drivers, a few definitions are in order. This section also explores some of the trade-offs inherent in using a hierarchical driver architecture.

Intermediate Drivers Defined

For the purposes of this chapter, an *intermediate driver* is any kernel-mode driver that issues I/O requests to another driver. Intermediate drivers are often not responsible for any direct, register-level manipulation of hardware resources. Instead, they depend on a lower-level device driver, such as the physical driver, to perform hardware operations. In fact, intermediate drivers can assume a wide variety of forms.

The term "intermediate driver" is so generic that it must be classified in order to provide useful information. The possible classifications depend on whether or not the driver conforms to the WDM model.

For WDM-compliant driver implementations, there are three kinds of drivers:

- - Bus drivers**
 - Provide a hardware bus interface on a per slot basis and create one or more Physical Device Objects (PDOs).
- - Function drivers**
 - Provide read, write, and other functional logic for an individual device. They create and manage one or more Functional Device Objects (FDOs)
- - Filter drivers**
 - Provide modifications to an I/O request before presentation to lower layer drivers. Filters can be placed around the functional driver or on top of a bus driver.

Note the relationship between a bus driver and a physical device object. The bus driver constructs and manages PDOs and is therefore often called a physical driver.

Outside the WDM model, driver architectures can choose to use layered approaches in other ways:

- - Filter drivers**
 - A driver can be written to transparently intercept requests intended for some other driver. This allows behavior modifications of an existing driver.
- - Tightly coupled drivers**
 - This category includes drivers that define a private interface between them. The interface does not use the I/O Manager's calling mechanism for inter-driver communication.

With the exception of tightly coupled drivers, all intermediate drivers are a form of layered drivers. Later parts of this chapter explain how to develop drivers in each of these families.

When To Use a Layered Architecture

One early and important driver design issue is whether to implement the driver as a series of layers, or whether it should be structured as a single, monolithic unit. The following section provides the trade-offs of using a layered approach.

PROS OF LAYERED ARCHITECTURE

Depending on the goals, multiple driver layers can provide a number of benefits. Using layers allows the separation of higher-level protocol issues from management of the specific underlying hardware. This makes it possible to support a wider variety of hardware without having to rewrite large amounts of code. It also promotes flexibility by allowing the same protocol driver to plug into different hardware drivers at runtime. This is the approach taken by Windows 2000 network drivers.

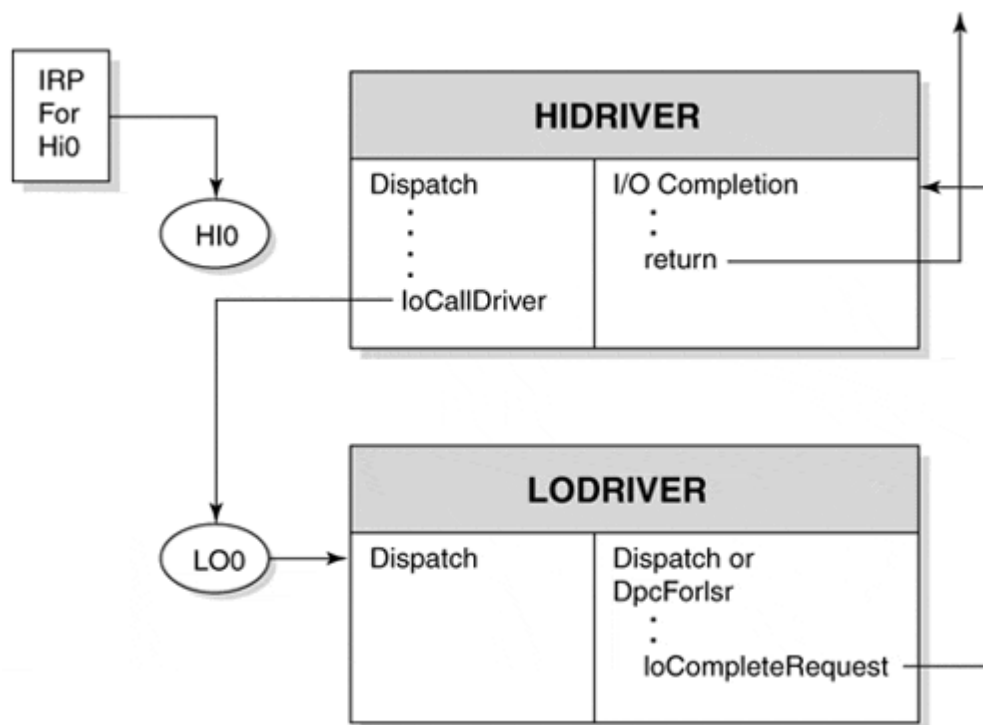
Writing Layered Drivers

Layered drivers are the most general type of intermediate driver. They depend on a well-defined inter-driver calling mechanism provided by the I/O Manager. The following three sections explain how this mechanism works and what a driver needs to do if it wants to use another driver as a component.

How Layered Drivers Work

As shown in [Figure 15.1](#), a layered driver may expose one or more named Device objects to which clients send I/O requests. When an IRP representing one of these requests arrives, the layered driver can proceed in two different ways.

Figure 15.1. Layered driver operation.



- Send the IRP directly to a lower-level driver.
- Hold the IRP in a pending state while it allocates additional IRPs and sends them to one or more lower-level drivers.

If the layered driver needs to regain control after a lower-level driver finishes with an IRP, it can attach an I/O Completion routine to the IRP. This routine executes when the lower driver calls **IoCompleteRequest**.

Initialization and Cleanup in Layered Drivers

Like every other kernel-mode driver, a layered driver must expose a main entry point called **DriverEntry**. It must also supply a corresponding **Unload** routine. As a WDM driver, a layered driver must support **AddDevice** and **RemoveDevice**, along with other PnP request handlers as appropriate. The following sections describe what these routines must do.

DriverEntry

The initialization steps performed by a layered driver are similar to those of a regular driver. The prime difference is that a layered driver must determine which I/O requests it handles directly and which it passes through to lower-level drivers. The layered driver entry points are initialized to routines that perform the appropriate action on the various IRP requests.

AddDevice

The work performed by a layered driver is a variation of the work that any WDM driver must perform. It includes

1.

Calling **IoCreateDevice** to build the upper-level device objects seen by the outside world. Like the device objects created by other drivers, the device name must be unique.

Writing I/O Completion Routines

An I/O Completion routine is an I/O Manager callback that notifies a higher-level driver when the lower-level driver has finished work on an IRP. This section explains how to use I/O Completion routines in intermediate drivers.

Requesting an I/O Completion Callback

To regain control of an IRP after it has been processed by a lower-level driver, use **IoSetCompletionRoutine** (described in [Table 15.1](#)). This function puts the address of an I/O Completion routine in the IRP stack location associated with the next lower driver. When some lower-level driver calls **IoCompleteRequest**, the I/O Completion routine executes as the IRP bubbles its way back to the top of the driver hierarchy.

Except for the driver on the bottom, each driver in the hierarchy can attach its own I/O Completion routine to an IRP. This allows every level to receive notification when an IRP completes. The I/O Completion routines execute in driver-stacking order, from bottom to top.

Table 15.1. Function Prototype for IoSetCompletionRoutine

VOID IoSetCompletionRoutine	IRQL <=DISPATCH_LEVEL
Parameter	Description
IN PIRP pIrp	Address of IRP the driver wants to track
IN PIO_COMPLETION_ROUTINE CompletionRoutine	Routine to call when a lower driver completes the IRP
IN PVOID pContext	Argument passed to I/O Completion routine
IN BOOLEAN bInvokeOnSuccess	Call routine if IRP completes successfully
IN BOOLEAN bInvokeOnError	Call routine if IRP completes with error
IN BOOLEAN bInvokeOnCancel	Call routine if IRP is canceled
Return value	- void -

The three BOOLEAN parameters passed to **IoSetCompletionRoutine** allow the callback to be invoked based on the disposition of the IRP by the lower level. The I/O Manager uses the **IoStatus.Status** field of the IRP to decide whether to invoke the I/O Completion routine.

Execution Context

By the time an I/O Completion routine is called, the I/O Manager has already popped the I/O stack pointer, so the current stack location is the one belonging to the driver. [Table 15.2](#) lists the arguments passed to an I/O Completion routine.

The execution context of an I/O Completion routine is the same as that of the caller of **IoCompleteRequest**. This could be either **PASSIVE_LEVEL** or **DISPATCH_LEVEL**, depending upon whether a DPC routine is used to complete the IRP. Since the design of the lower level is not likely to be under the control of the higher level, an I/O completion routine must assume the more restrictive context of **DISPATCH_LEVEL IRQL**.

When an I/O Completion routine is finished, it should return one of two status codes. Returning **STATUS_SUCCESS** allows the IRP to continue its journey back toward the original requester. Along the way, other I/O Completion routines attached to higher-level drivers execute. This is normally the appropriate return value when the IRP originated from a higher level.

To suspend further processing of the "completed" IRP, an I/O Completion routine can return **STATUS_MORE_PROCESSING_REQUIRED**. This value blocks the execution of any higher-level I/O Completion routines attached to the IRP. It also prevents the original caller from receiving notification that the IRP has completed. An I/O Completion routine should return this code if it either plans to send the IRP back down to a lower-level driver (as in the case of a split transfer) or if the IRP was allocated by this driver and the I/O Completion routine is going to deallocate it.

Table 15.2. Function Prototype for an I/O Completion Routine

NTSTATUS IoCompletion	IRQL ==PASSIVE_LEVEL or IRQL == DISPATCH_LEVEL
Parameter	Description
IN PDEVICE_OBJECT pDevObj	Device object of the just completed request
IN PIRP pIrp	The IRP being completed
IN PVOID pContext	Context passed from IoSetCompletionRoutine
Return value	• STATUS_MORE_PROCESSING_REQUIRED • STATUS_SUCCESS

Allocating Additional IRPs

There are some situations where an intermediate driver needs to allocate additional IRPs to send to another driver. For example, the initialization code in one driver might want to query the capabilities of a lower-level driver by issuing an IOCTL request. An example filter driver that implements this strategy is listed later in this chapter.

As another example, a fault-tolerant disk driver, implemented as an intermediate driver, might allocate an additional IRP to send to a second (mirror) driver. This second IRP would mirror the original request.

Yet a third example occurs when a higher-level driver exposes an abstract command to a client. The command itself is implemented through a series of lower-level calls, each requiring the allocation of a new IRP. The SCSI class driver implements this strategy when relying upon the lower-level SCSI port driver.

The IRP's I/O Stack Revisited

Before explaining the available IRP allocation techniques, it is important to have a clear understanding of the IRP stack operation. As already described, each driver that receives an IRP is supplied a unique IRP stack location that is easily obtained with a call to **IoGetCurrentIrpStackLocation**.

If an intermediate driver plans to pass an incoming IRP to a lower-level driver, it has to set up the I/O stack location for the next lower driver. To obtain a pointer to the lower driver's I/O stack slot, the intermediate driver uses **IoGetNextIrpStackLocation**. After setting up the lower stack slot (perhaps by copying the current slot into the next slot), the intermediate driver uses **IoCallDriver** to pass the IRP down. This function, **IoCallDriver**, automatically pushes the I/O stack pointer so that when the lower driver calls **IoGetCurrentIrpStackLocation**, it will get the right address (i.e., one lower than its caller).

When the lower driver calls **IoCompleteRequest**, the completed IRP's I/O stack is popped. This allows an I/O Completion routine belonging to the higher driver to call **IoGetCurrentIrpStackLocation** if it needs to access its own stack location. As the IRP bubbles its way back up to the original caller, the I/O stack is automatically popped again for each driver in the hierarchy. [Table 15.3](#) summarizes the effects of these functions on an IRP's I/O stack pointer.

To maintain consistent behavior with driver-allocated IRPs, the I/O Manager initializes the new IRP's I/O stack pointer so that it points to a nonexistent slot one location before the beginning of the stack. This ensures that when the driver passes the IRP to a lower-level driver, **IoCallDriver**'s "push" operation sets the stack pointer to the first real slot in the stack. Thus, the higher-level driver must call **IoGetNextIrpStackLocation** to retrieve a pointer to the I/O stack slot intended for the target driver.

Table 15.3. Effect of Functions on IRP's I/O Stack Pointer

Working with the IRP Stack Pointer	
Function	Effect on the IRP stack pointer
IoGetCurrentIrpStackLocation	No change
IoGetNextIrpStackLocation	No change
IoSetNextIrpStackLocation	Pushes stack pointer by one location
IoCallDriver	Pushes stack pointer by one location
IoCompleteRequest	Pops stack pointer by one location

Controlling the Size of the IRP Stack

When a driver receives an IRP from an outside caller, the number of I/O stack slots is determined by the **StackSize** field of the driver's Device object. If the intermediate driver plans to pass incoming IRPs to a lower-level driver, it needs to increment this field to one more than the number of slots reserved by all lower drivers. That is, it must set the Device object's **StackSize** field to the value of the lower Device object's **StackSize** field plus one. This ensures that there are enough stack slots for all drivers within the hierarchy. Of course, the technique requires that drivers pile on top of each other, with the lower driver initialized prior to the higher driver.

The value of **StackSize** in a Device object represents the number of stack slots needed by all lower drivers, including one slot for itself. That is, it represents the maximum call depth *beneath* the current level plus one.

The I/O Manager constructs IRPs upon request of a driver when any of the following calls are made:

-
- **IoBuildAsynchronousFsdRequest**
-
- **IoBuildDeviceIoControlRequest**
-

Writing Filter Drivers

A filter driver is a special type of intermediate driver. Filter drivers perform their work surreptitiously. They sit on top of some other driver and intercept requests directed at the lower driver's Device objects. Users of the lower driver are completely unaware that their requests are being preprocessed or intercepted by the filter driver. Some examples of the use of filter drivers include the following:

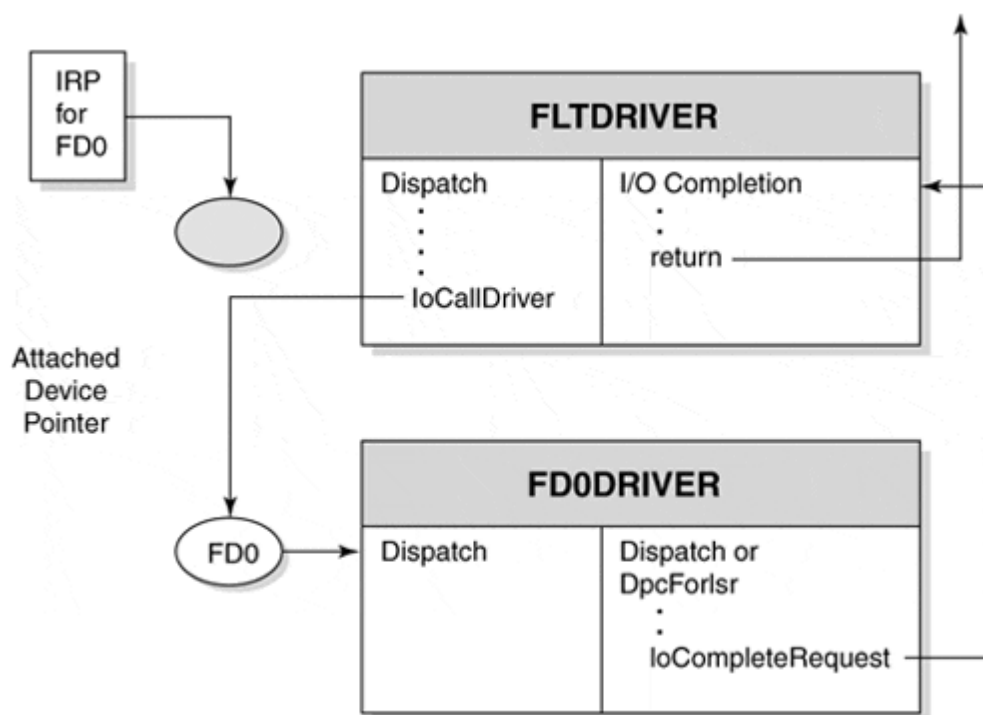
- Filters allow modification of some aspect of an existing driver's behavior without rewriting the entire driver. SCSI filter drivers work this way.
- Filters make it easier to hide the limitations of lower-level device drivers. For example, a filter could split large transfers into smaller pieces before passing them on to a driver with transfer size limits.
- Filters allow the addition of new features like compression or encryption to a device without modifying the underlying device driver or the programs that use the device.
- Filters allow the addition or removal of expensive behavior (like performance monitoring) a driver may not perform at all times. The disk performance monitoring tools in Windows 2000 work this way.

The remainder of this section explains how to write filter drivers. Bear in mind that driver-allocated IRPs and I/O Completion routines work the same in a filter driver as they do in a regular layered driver.

How Filter Drivers Work

The main distinction between filter drivers and other layered drivers is in the Device objects they create. Whereas a layered driver exposes Device objects with their own unique names, filter drivers' Device objects have no names at all. Filter drivers work by attaching one of these nameless Device objects to a Device object created by some lower-level driver. [Figure 15.2](#) illustrates this relationship.

Figure 15.2. Filter driver operation.



In the diagram, **FLTDRIVER** has attached a filter Device object to **FDO**, one of **FDODRIVER**'s Device objects. Any IRPs sent to **FDO** are automatically rerouted to the Dispatch routines in **FLTDRIVER**. It works as follows:

1.

The **AddDevice** routine in the filter driver creates an invisible Device object and attaches it to a named Device object belonging to a driver beneath it.

Code Example: A Filter Driver

This example demonstrates a basic filter driver, called the HIFILTER, that intercepts all requests intended for a lower-level driver (LODRIVER). The purpose of the filter is to hide the lower driver's limited output transfer size. To do this, it breaks large transfer requests into smaller pieces. It also overrides an IOCTL from the lower driver that returns the maximum size of an output buffer. All other major function codes supported by the lower driver are passed through from the filter.

The code for this example is included on the accompanying disk and at the book's Web site: <http://www.W2KDriverBook.com>.

The DEVICE_EXTENSION Structure

The declarations of the Device Extension for the filter driver are minimal; the filter driver has few direct responsibilities.

```
// BUFFER_SIZE_INFO is a driver-defined structure
// that describes the buffers used by the filter
typedef struct _BUFFER_SIZE_INFO
{
    ULONG MaxWriteLength;
    ULONG MaxReadLength;
} BUFFER_SIZE_INFO, *PBUFFER_SIZE_INFO;

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDeviceObject;    // Back pointer

    PDEVICE_OBJECT pTargetDevice;    // Lower device

    BUFFER_SIZE_INFO bufferInfo;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

The DriverEntry Function

The **DriverEntry** function for a filter is unique in that its Dispatch routine announcement must closely follow the functionality exposed by the lower driver. If the lower driver supports a Dispatch entry, the filter must expose it as well. For those entries that the filter is modifying (overriding), the filter installs its own Dispatch function address. For entries that the filter does not override, a pass-through dispatch function is installed.

```
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT pDrvObj,
    IN PUNICODE_STRING pRegPath ) {
    // Export other driver entry points
    pDrvObj->DriverUnload = DriverUnload;
    pDrvObj->DriverExtension->AddDevice = AddDevice;

    // Assume (initially) nothing is overridden
    for (int i=0; i<=IRP_MJ_MAXIMUM_FUNCTION; i++)
        if (i!=IRP_MJ_POWER)
            pDriverObject->MajorFunction[i] = DispatchPassThru;

    // Export the overridden MajorFunctions
    pDrvObj->MajorFunction[ IRP_MJ_WRITE ] =
        OverriddenDispatchWrite;
    pDrvObj->MajorFunction[ IRP_MJ_DEVICE_CONTROL ] =
        OverriddenDispatchIoControl;

    :
    return STATUS_SUCCESS;
}
```

The AddDevice Function

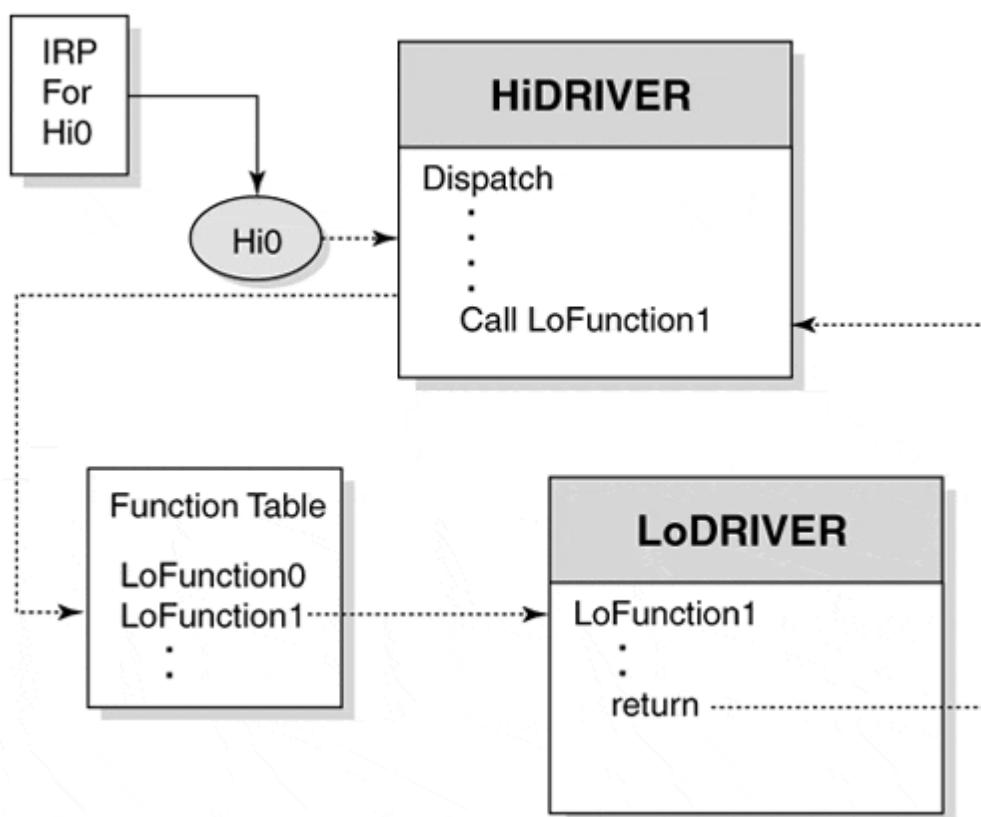
Writing Tightly Coupled Drivers

Unlike layered and filter drivers, tightly coupled drivers do not use the I/O Manager's **IoCallDriver** function to communicate with each other. Indeed, tightly coupled drivers do not need to conform to the strict stacking architecture that the WDM model constructs. Instead, they define a private calling interface. The advantage of this approach is that it is usually faster than the IRP passing model supported by the I/O Manager. In trade for improved performance, the mechanics of the interface between tightly coupled drivers is left as an implementation detail. As a result, it is difficult for tightly coupled drivers from different vendors to interoperate.

How Tightly Coupled Drivers Work

Since the interface between two tightly coupled drivers is completely determined by the driver designer, it is impossible to provide a single, unified description of how all tightly coupled drivers work. Instead, this section presents some general architectural guidelines. [Figure 15.3](#) shows one common method of tightly coupling a pair of drivers.

Figure 15.3. Tightly coupled driver operation.



In [Figure 15.3](#), the lower driver has exposed an interface. During the upper driver's initialization, the interface is retrieved from the lower driver using the PnP request `IRP_MN_QUERY_INTERFACE`. The interface is returned as a structure of data and function pointers. When the upper driver needs the services of the lower driver, it calls one of the functions in this structure directly, rather than using **IoCallDriver**. Before unloading, the upper driver calls another function in the interface, **InterfaceDereference**, to disconnect it from the lower driver.

Initialization and Cleanup in Tightly Coupled Drivers

The following sections describe in general terms how a pair of tightly coupled drivers might initialize and unload. Of course, the exact steps vary depending upon the driver design.

Lower AddDevice Routine

The "lower" driver acts as a server for the upper driver, exposing an interface that is used by this "client." The following steps describe the work performed by the lower driver.

1.

Using the technique discussed in chapter 9, the lower driver exposes an interface using **IoRegisterDeviceInterface**. The interface exposed has a unique GUID and the interface structure is known to all clients a priori, including the higher driver.

2.

The lower driver calls the `IoCallDriver` function to call the `IoCallDriver` function in the higher driver.

Summary

The layered architecture of Windows 2000 allows the simplified design of device drivers. Breaking a monolithic driver into smaller, logically distinct pieces makes implementation and maintenance easier, reduces debugging time, and increases the likelihood that some of the software will be reusable.

In this chapter, different methods of stacking drivers on top of one another were presented. Most of the techniques depend on the I/O Manager's standard calling mechanism to send IRPs from one driver to another. When this mechanism proves insufficient, a private interface can be defined between a pair of drivers. In general, private interfaces make the design more fragile and harder to maintain.

The next chapter describes the details necessary to professionally install device drivers.

Chapter 16. Driver Installation

CHAPTER OBJECTIVES

- - [Installation of a Driver](#)
- - [Auto-Install Using INF Files](#)
- - [Using a Driver INF File](#)
- - [Controlling Driver Load Sequence](#)
- - [Digital Signing of a Driver](#)
- - [Summary](#)

The convenient, automated, and trouble-free installation of a device driver is an essential step for users and support personnel. For Plug and Play drivers, the installation process is fundamental in providing the user with effortless device management.

This chapter focuses on the details necessary to ensure proper device driver installation.

Installation of a Driver

The automated installation of a device driver is controlled by a text file with the INF extension. The INF format resembles the old-style .INI files of Windows 3.x fame, but is deceptively more complex. An INF file, properly stored on a Windows 2000 system, allows for an automated or dialog-assisted installation of driver files.

It is important to understand that the end result of a driver installation consists of two persistent changes to a system:

- - System registry entries describing the driver, its load order, and any appropriate configuration data
- - Driver files, copied to a suitable system directory

While the INF file is the standard mechanism for affecting these changes, it is also true that a custom installation program can be provided, if appropriate, to force the same resulting system changes. The next section deals with the syntax and operation of an INF file.

Auto-Install Using INF Files

INF files are typically provided with the hardware and driver on diskette or CD. The structure and content of the INF file is the responsibility of the driver author.

INF File Structure

An INF file is a simple text file divided into sections, with each section designated by an identifier within closed braces ([]). Some section names are required, while others are driver-specific. Entries beneath each section control some installation action, or they link or enumerate other sections.

The order that sections appear in a file is unimportant, since each section is named and linked. A section continues until either another section or the end-of-file is encountered. The unique name that specifies a section is case-insensitive and should be limited to 28 characters in length to maintain compatibility with Windows 98. Section names can include spaces, but only if the entire name is quoted. The underscore and dot characters are allowed.

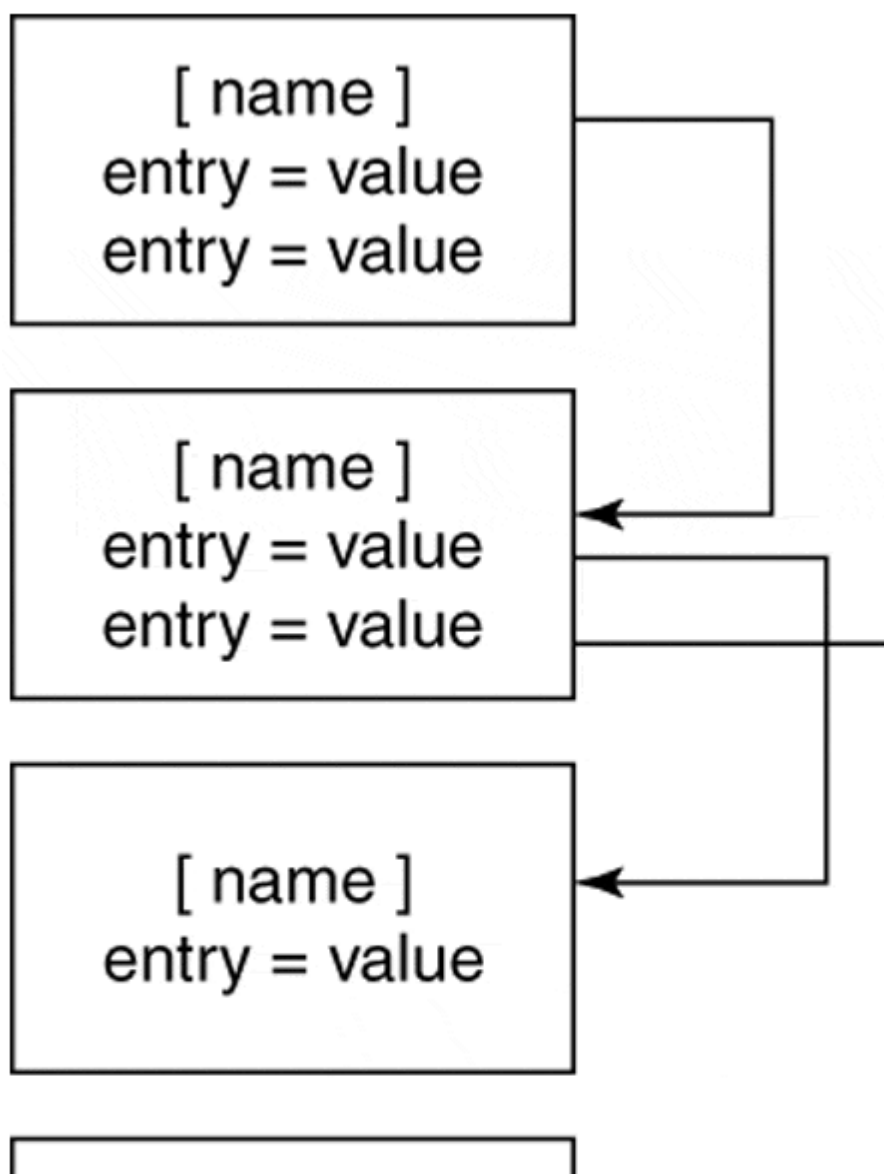
The general format of section entries is

```
entry = value [, value...]
```

where *entry* is a directive, keyword, or filename, and *value* is the attribute that is to be applied to *entry*.

An illustration of the linkage of section names is shown in [Figure 16.1](#).

Figure 16.1. *INF file section linkage.*



Using a Driver INF File

Once the driver's INF file is created, it must be processed for it to take effect. Obviously, the strict adherence to INF syntax suggests that an engine exists for just such a purpose. This section describes the mechanics of running the engine, either automatically or manually.

Manual Installation

To process an INF file manually, simply use the Windows Explorer file manager to select the INF file. Right-clicking the file offers the option to Install (process) the file.

In a true Plug and Play environment, the insertion or removal of a device triggers the installation, load, and unload of an appropriate driver. Therefore, the use of manual installation is reserved for initial testing and debugging of a driver.

Automatic Installation

When a PnP device is inserted into a system, several subsystems interact to force the loading of a new driver, if necessary. The steps are outlined below.

1.

When the device is inserted, the hardware, using auto-detection and notification, alerts the bus driver that the device is present. Depending on the bus hardware, this might involve notifying the bus driver that a change has occurred that warrants a new enumeration of bus devices. Regardless, at the end of this step, the bus driver is aware that the new device is present and that it has a specific device ID.

2.

The kernel-mode PnP Manager notifies the user-mode PnP Manager that a new device with a specific ID is present in the system.

3.

The user-mode PnP Manager, using the Setup API library of Windows 2000, constructs a likely list of drivers for the new device. The INF directory of the system (e.g., WINNT\INF) is searched using the class and model information from the new device for a suitable match.

4.

If a suitable INF file cannot be located, the system delays further action until a privileged user logs on. Then the user is presented with a dialog-driven New Hardware wizard. The user supplies the location of the drivers (CD, diskette, Web location, etc.) and the appropriate INF file is located.

5.

Once the INF file is found, it is processed using the CfgMgr API library. The driver files and Registry entries are installed and modified. This step is primarily carried out by the kernel-mode PnP Manager.

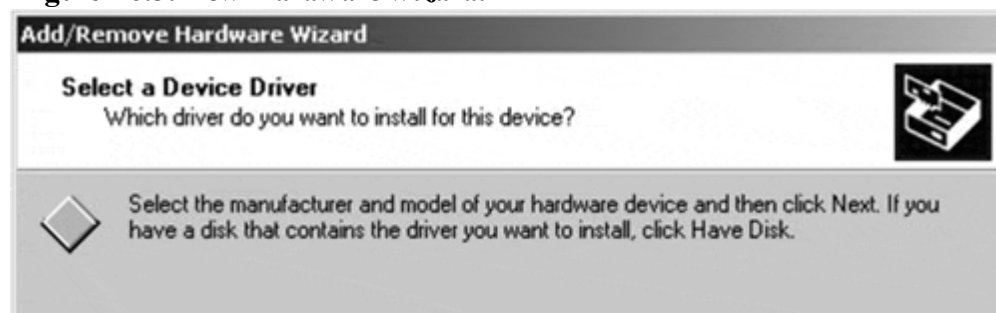
6.

Based on the directives of the INF file, the kernel-mode PnP Manager loads any lower filter drivers, then the functional driver, and finally any upper filter drivers for the device. The top driver in the stack is then sent appropriate PnP messages, including IRP_MN_START_DEVICE.

The Add/Remove Hardware Wizard

As just explained, the user must sometimes interact with even an automated installation. The prime interface for this interaction is the New Hardware wizard. A typical screen shot from this (all too) familiar tool is shown in [Figure 16.3](#). Notice that there is a column for Manufacturers (listed from INF files [**Manufacturers**] section), and for Models (listed from INF files [**Models**] sections).

Figure 16.3. *New Hardware wizard.*



Controlling Driver Load Sequence

Sometimes, the order in which a driver loads is important. Perhaps one driver depends on another to complete its initialization. When such dependencies arise, the Service Control Manager can be informed with the Registry entries **LoadOrderGroup** and **Dependencies**. These values can also be specified in the **[ServiceInstall]** section of an INF file.

The **LoadOrderGroup** identifies by name a single group in which the specified driver is a member. The **Dependencies** value specifies which other groups must load prior to this driver loading successfully.

The need for such interdependencies is not common, nor should it be. In a truly device-centric environment (which Plug and Play purports), drivers should be in a position to load and unload with few strings attached. On the other hand, heavily layered driver implementations may have legitimate needs for such specifications.

Driver Stack Order

A related issue concerns the order in which drivers stack. In a layered driver environment, any number of drivers can stack on top of each other to distribute and reuse code. In a typical WDM model, driver stacks exist that contain a PDO (Physical Device Object) at the bus level, a possible bus filter object, an FDO (Functional Device Object), and surrounding upper and lower filter devices. (refer to [chapters 1](#) and [15](#).)

In such environments, it is the Registry entries (once again) for each driver layer that determine the order in which drivers stack. Notice that the *load* order is independent of the *stack* order. When a driver invokes **IoAttachDeviceToDeviceStack**, it is the responsibility of the I/O Manager to link drivers appropriately, independently of their load order.

The Registry entries for a driver may contain values named **UpperFilters** and **LowerFilters**. These are REG_MULTI_SZ string arrays, each containing the names of drivers that stack above and below the given driver. The order of the string array determines the stack order for multiple upper or lower filters.

A segment of an INF file appears below to demonstrate the installation of a filter driver. Notice that the upper filter is installed as a *demand start* driver. Even though it has announced that it stacks on top of the functional device, the driver may not load and insert itself into the stack until it is manually loaded.

```
...
[upperFilter_install] ; DDInstall section
CopyFiles = @upperFilter.SYS
AddReg = upperFilter_addreg

[upperFilter_addreg] ; AddReg section
; append this filter to the list of filters
HKR,, "UpperFilters", 0x10008, "upperFilter"

[upperFilter_install.Services] ; DDInstall.Services
AddService = Launcher, 0x2, Launcher_Service
AddService = upperFilter, , upperFilter_Service

[upperFilter_Service]
DisplayName = "Upper Filter Service for Launcher"
ServiceType = 1 ; kernel-mode driver
StartType = 3 ; demand start
ErrorControl = 1
ServiceBinary = %12%\upperFilter.SYS
```

Digital Signing of a Driver

Many third-party drivers ship with the Microsoft Windows 2000 CD distribution. In order to participate in this program, several requirements must be met.

Additionally, whenever a naive attempt to install a driver on Windows 2000 is made, a warning message is issued stating that the device driver is not *digitally signed* and its authenticity cannot be verified by Microsoft.

This section explains the role of Microsoft in verifying driver authenticity and the burden placed on a driver author and hardware manufacturer to certify their package for inclusion into future Windows 2000 CD distribution.

Why Microsoft Verifies Drivers

It is in Microsoft's best interest to promote two (conflicting) goals.

- - Provide or promote as much interoperability for Windows 2000 and varied hardware devices as possible through the convenient distribution of device drivers.
- - Ensure that device drivers are stable and do not compromise the integrity of the system.

Since device drivers operate in kernel mode, they have the capability to slowly or quickly crash a system. Since instability of the system will often be blamed on the kernel itself, it is clearly in Microsoft's interest to maintain a list of *certified* vendors and drivers for their operating systems.

Of course, stating that Windows 2000 interoperates with more hardware devices than other OSs is a strong selling feature. Therefore, Microsoft often works with hardware vendors to ensure timely release of compatible drivers.

To accomplish the two goals, Microsoft has established a specialized group, the Windows Hardware Quality Labs (WHQL), that provides a certification of hardware and the device driver. The participation benefits for hardware vendors include

- - Use of the Windows logo on the certifying hardware and software
- - Inclusion in the official list of supported and certified hardware for the various Microsoft operating system offerings (see <http://www.microsoft.com/hcl>)
- - The opportunity to distribute the driver with future releases of the OS
- - A digital signature that ensures tamper-proof code and allows customers to obtain the latest versions of drivers through Microsoft's Windows Update site

To participate in the program, visit the site <http://www.microsoft.com/hwtest> for procedures and pricing.

Digital Signatures

As part of the WHQL program, a certified driver obtains a digital signature that permits Windows 2000 to install the driver without the unprofessional warning of "imminent danger." The digital signature consists of several components.

- - A catalog file (.CAT) that is included with the distributed driver package. It contains the actual digital signature assigned by Microsoft.
- - An INF file entry in the [Versions] section that references the .CAT file.
- - Windows 2000 policy that restricts whether or not an unsigned driver can be installed.

The digital signature is allegedly tamper-proof and ensures that the driver being installed is the original code supplied by the vendor. It uses cryptographic technology to achieve this goal. The signature itself does not alter the code in any way.

Summary

For those familiar with the installation of drivers into earlier implementations of Windows, the automated process of Windows 2000 and Windows 98 is a great step forward for users. However, the step requires additional work on the part of the driver author in that, at a minimum, an INF file must be supplied. In the end, all parties benefit from the common, standardized mechanism of driver installation.

The final chapter in the book deals with the necessary subject of debugging device drivers using Microsoft tools.

Chapter 17. Testing and Debugging Drivers

CHAPTER OBJECTIVES

- - [Guidelines for Driver Testing](#)
- - [Why Drivers Fail](#)
- - [Reading Crash Screens](#)
- - [An Overview of WinDbg](#)
- - [Analyzing a Crash Dump](#)
- - [Interactive Debugging](#)
- - [Writing WinDbg Extensions](#)
- - [Code Example: A WinDbg Extension](#)
- - [Miscellaneous Debugging Techniques](#)
- - [Summary](#)

In many ways, this chapter should be first in the book. After all, it is not possible to design software (that works, anyway) without considering a testing and debugging strategy from the very beginning. Of course, since the purpose of the book is to present the Windows 2000 driver architecture, the focus has been to explain the way drivers work, not fail.

The purpose of this chapter is to introduce the concept of writing *defensive* driver code. By considering the ways in which code can fail, the design and implementation of the driver can facilitate the isolation and reporting of the error. The techniques covered in this chapter include a presentation of trace methods and procedures.

The chapter also presents some tools provided by Microsoft with the DDK and elsewhere. For example, the very useful WinDbg debugger operation is explained.

And, by placing this chapter at the end, it does make for easy reference whenever it is needed.

Guidelines for Driver Testing

In many ways, driver testing is like all software testing: develop test cases that exercise boundary and stress conditions and measure the results. At a more practical level, however, driver testing requires innovation, real-time skills, hardware knowledge, and above all, patience.

A Generalized Approach to Testing Drivers

No complex body of code can ever be bug-free. Everyone is familiar with the phenomena that fixing bugs introduces new bugs. Bugs cannot be stopped; they can only be contained. This is especially true when software interacts with other vendors' software and hardware. Further, as software design occurs in layers and components, the actual mix of software and hardware versions may never have been tested as a system. This classic problem appears time and again with DLLs. Vendors test their code with one version of DLLs, but by the time the product is deployed en masse, the DLL versions have changed on the end users' systems.

Therefore, every test plan must be reasonable—searching for the knee in the curve beyond which diminishing returns on the testing effort occur. The real point is that test design is every bit as challenging as software design.

WHEN TO TEST

Experience shows that incremental testing of software components as they are developed is far more effective than waiting until the entire system is constructed. Although incremental testing requires a large number of small tests, bug isolation makes the technique worthwhile. Additionally, predicting the ship date of progressively tested software is more reliable than predicting the ship date of code that has never been tested.

The small test programs developed for this strategy also form the basis of a more formal regression test. As future changes are made to the code base, the small tests help ensure that new bugs are not introduced.

Yet another advantage of incremental testing throughout the driver development phase is that hardware design flaws are identified early. This is especially important for new hardware under design. Nothing kills a schedule more than identifying another spin on a custom ASIC late in the development phase.

WHAT TO TEST

Generally, driver tests can be categorized as follows:

- - Hardware tests**
 - verify the operation of the hardware. These tests are essential when the device and driver are being developed in parallel.
- - Normal response tests**
 - validate the complete and accurate functionality of the driver. Does the driver respond to each command as promised?
- - Error response tests**
 - check for appropriate action when a bad stimulus is applied to the driver. For example, if the device reports an error, does the driver respond by reporting and logging the error? The error stimulus can also be bad data from a user response.
- - Boundary tests**
 - exercise the published limits of the driver or device. For example, if there is a maximum transfer size, does the driver deal appropriately when presented with one more byte? Speed boundaries are also addressed in this category.
- - Stress tests**
 - subject the driver and device to high levels of sustained activity. The amount of stimulus is ideally just beyond what will be encountered in the real world. Within this category, different subcategories of stress can be applied. For example, limited CPU availability, limited memory, and heavy I/O activity are all dimensions where stress can be applied.

HOW TO DEVELOP THE TESTS

Why Drivers Fail

While testing uncovers the *presence* of bugs, the more serious challenge is to analyze, isolate, and correct the source of the bug. The goal of this section is to provide the basis for the analysis of driver bugs. Drivers fail for specific reasons and some up-front thought about the ways in which they fail can start the elusive search for a bug in an orderly fashion.

Categories of Driver Errors

Drivers can fail in any number of interesting ways. Although it is not possible to give a complete list, the following sections describe some of the more common types of driver pathology.

HARDWARE PROBLEMS

It goes without saying (to a software developer, anyway) that there is always an even chance that the hardware itself is the source of a problem. In fact, when developing drivers for new, undeployed, untested hardware, the chances of a hardware problem rise significantly. Symptoms of hardware problems include

- - Errors occur during data transfer.
- - Device status codes indicate an error (when a device reports an internal error, it is the equivalent of a confession).
- - Interrupts do not arrive or they arrive spuriously.
- - The device does not respond properly to commands.

The cause might be as simple as undocumented behavior in the device, and hardware designers have been known to alter documentation after witnessing the results of their work. There might be a restriction on command timing or sequencing. The firmware on the device might be faulty. There could be a bus protocol problem resulting in sporadic failures as other devices on the bus engage. Then again, the device might just be broken.

Because attempting to debug a problem whose ultimate source is hardware is so frustrating, it is best to eliminate (within reason) this category of fault before proceeding.

The best approach to validate a hardware problem is to employ a logic analyzer or hardware emulator. Hardware and software designers should work closely on unstable platforms until the first level of fault isolation can be determined.

SYSTEM CRASHES

Because driver code operates in kernel mode, it is straightforward for such code to kill the entire system. While many driver logic errors produce a crash, the most common stem from access violations (e.g., referencing a logical address that has no physical memory behind it) through use of a bad C pointer. Among the more difficult-to-trace scenarios within this category is setting DMA addresses incorrectly into the mapping registers. The device scribbles into random memory with a resulting crash seemingly caused by another entire subsystem.

A later section in this chapter deals with analyzing system crashes to determine the ultimate source.

RESOURCE LEAKS

Because kernel-mode driver code is trusted code, the system does not perform any tracking or recovery of system resources on behalf of a driver. When a driver unloads, it is responsible for releasing whatever it may have allocated. This includes both memory from the pool areas and any hardware the driver manages.

Even while a driver is running, it can leak memory if it regularly allocates pool space for temporary use and then fails to release it. High-layered drivers can leak IRPs by failing to free those fabricated and passed to lower levels. Resource leaks force sluggish system performance and, ultimately, a system crash.

Windows 2000 allows memory allocated by kernel-mode code to be tagged with an ID. When analyzing system memory after a crash, the tags help determine where the blocks are located, their size, and most importantly, which subsystem allocated them. Tools such as GFLAGS, supplied with the Platform SDK, enable the pool tagging feature globally for the system.

Resource leaks in progress can sometimes be determined by careful monitoring of system objects. Tools such as WINOBJ (supplied with the Platform SDK or by www.sysinternals.com) assist in this monitoring.

Reading Crash Screens

System crashes, which Microsoft euphemistically terms "STOP messages," are a dramatic sign that a driver has failed. This section describes how STOP messages are generated and explains how they provide useful information.

What Happens When the System Crashes

Despite its name, a system crash is quite orderly. The OS is reporting that it has detected an internal state that is inconsistent with continued operation. As a result, the OS believes it is better to shutdown *now* rather than continue operation with the illegal state. This is appropriate in many cases since the illegal state could cause real damage, perhaps to the file system. Continuing operation under such circumstances could also breach security.

Two different sequences of events can lead to a system crash. First, some kernel-mode component can be the detector of an illegal state and elect to force a system STOP. For example, if the I/O Manager discovers that a driver is passing an already completed IRP to **IoCompleteRequest**, the I/O Manager forces a system STOP.

The second scenario is more indirect. A kernel-mode component is the direct source of the failure, generating an exception within its code operation. For example, a memory access violation or integer division by zero result in an exception being generated. Referencing paged memory at an elevated IRQL level is also an exception. If the kernel-mode routine that performs the illegal operation does not trap the exception directly, the result is an *unhandled exception* within kernel mode.

When this occurs, a standard exception handler within the kernel responds and initiates a system STOP.

Regardless of which kernel subsystem initiates the system STOP, one of two calls actually performs the operation.

```
VOID KeBugCheck( Code );
VOID KeBugCheckEx( Code, Arg1, Arg2, Arg3, Arg4 );
```

Either function generates the STOP screen and optionally saves a crash dump file to disk. Depending on the system administrator's choice, the system either halts, reboots, or starts the kernel's debug client.

The *Code* argument to the **KeBugCheck** routines specifies the cause of the crash as a simple DWORD value. It is also known as a *bugcheck* code. The additional four arguments to **KeBugCheckEx** appear within the STOP message screen as further information for the user to report (and hopefully assist in isolating the source of the STOP).

Bugcheck codes are either defined by Microsoft (see BUGCODES.h within the DDK and [Appendix B](#) of this book) or are driver-specific.

Calls to either **KeBugCheck** function can be made from a driver. In fact, it is common for a debug version of a driver to force system STOPS at various points of interest.

The Blue Screen of Death

A system STOP message appears on an entire screen of blue background, earning it the name *Blue Screen of Death*, or BSOD. In versions of NT prior to Windows 2000, the STOP message filled the screen with information of questionable value. In Windows 2000, the STOP message is simplified, containing the most relevant information about the cause and location of the crash. A sample BSOD is illustrated in [Figure 17.1](#).

Figure 17.1. The STOP message screen (Blue Screen of Death).

```
*** STOP: 0x000000D1 (0x00000000,0x00000002,0x00000000,0xFCE10796)
DRIVER_IRQL_NOT_LESS_OR_EQUAL

*** Address FCE10796 base at FCE10000, DateStamp 398aeb8f - Driver.sys

Beginning dump of physical memory
Physical memory dump complete. Contact your system administrator or
technical support group.
```

An Overview of WinDbg

WinDbg (affectionately pronounced *wind bag*) is a hybrid kernel-mode and user-mode debugger that can be used to analyze crash dump files, executing driver code, and application dump files created by the Dr. Watson utility. This section provides an overview of the WinDbg capabilities. The DDK documentation provides additional details on the use of WinDbg in various debugging scenarios.

WinDbg is a hybrid debugger in that it serves a purpose in both kernel mode and user mode. It also combines the look and feel of a GUI (graphical) debugger with a window for entering old-style keyboard commands.

WinDbg is supplied from several sources, including the DDK and Platform SDK. Additionally, it is supplied on the Customer Support and Diagnostics Tools CD of the Windows 2000 distribution.

The Key to Source Code Debugging

One of WinDbg's most powerful features is its ability to debug kernel-mode components at the source-code level. Source code and symbol files must be available to WinDbg during the debug session to allow this mode of operation.

SYMBOL DIRECTORIES

Symbol files are optionally generated with the compile and link process of a project. They include names for local and global variables, linked function addresses, and typedef information. Line number information is also provided by symbol files so that compiled machine instructions can be associated with source code lines. Microsoft tools can supply symbol files in several formats including an older (but more standard) COFF format (Common Object File Format) and the PDB format (Program Database). Compiler and linker switches determine the format of the symbols generated.

The Windows 2000 operating system itself is supplied with associated symbol files. They are optionally installed from the Customer Support and Diagnostics Tools CD. Since symbol files change with each build, a service pack distribution of the operating system requires that the symbol files be updated.

Access to driver and OS symbols is essential for any serious debugging attempt. They provide intelligible call stack information and display of source code. Trying to isolate a bug from screens of disassembly is an exercise in frustration.

Once symbols are installed, WinDbg must be informed of their install location. This is accomplished by using the Symbols tab of the Options dialog box under the View menu. Operating system symbols are usually installed in the %SystemDir%\Symbols directory. Symbol files for a driver (.DBG or .PDB extension) are often kept with the binary .SYS file.

SOURCE CODE DIRECTORIES

Besides the symbol files, WinDbg needs access to the source code (.C and .H) for the driver in order to display it during debugging. A path for the source code directories can be supplied through the Source Files tab of the Options dialog under the View menu in WinDbg.

Some WinDbg Commands

Although WinDbg is a GUI program, its most useful operations are initiated from a command-line window. The command set is similar to other command-line debuggers supplied by Microsoft (e.g., NTSD and KD), so knowledge of the command set is leveraged. [Table 17.1](#) provides a quick overview of the more common WinDbg commands. The WinDbg Help option provides more detailed information. Also, see the DDK for driver-specific (kdextx86.DLL) extension command help. Extension commands begin with a bang (!).

Table 17.1. Common WinDbg Commands and Extensions

WinDbg Common Commands and Extensions	
Command	Description
Help	Display list of basic WinDbg commands
k, kb, and kv	Display stack trace
dd <i>address</i>	Dump the contents of memory
ln	Display symbol names to the break point
r	Display or modify CPU register(s)
p	Trace over
t	Trace into
g	go (continue execution)
.reboot	Reboot target computer
!help	Display help for WinDbg extension commands

Analyzing a Crash Dump

When a crash occurs, Windows 2000 can save the state of the system in a dump file. To enable this feature, the System Control Panel applet (Advanced tab, Startup and Recovery) must be configured. Crash dumps allow an immediate reboot of the system without losing the state of memory at the moment of failure. This section explains how to analyze a system crash dump.

Goals of the Analysis

With WinDbg and a crash dump file, the state of the failed system can be examined. It is possible to find out almost as much information as if it were still running or if a live debugger were attached at the moment of failure. This kind of forensic pathology can help develop a convincing explanation of what led to the crash. Some of the questions that should be asked during the analysis include

-
- Which drivers were executing at the time of the crash?
-
- Which driver was responsible for the crash?
-
- What was the sequence of events leading to the crash?
-
- What operation was the driver trying to perform when the system crashed?
-
- What were the contents of the Device Extension?
-
- What Device object was it working with?

Starting the Analysis

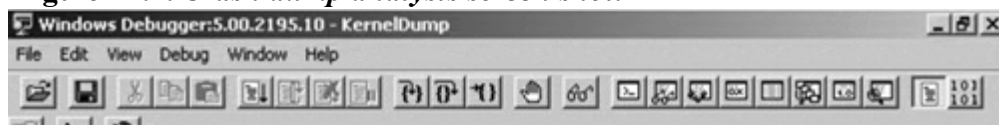
To begin the analysis, the crash dump file must be obtained. If WinDbg is available on the target system (the system that crashed), the file is present wherever the Control Panel configuration specified (e.g., WINNT\Memory.DMP). If the system that crashed is at a remote site, the dump file must be transported. Since dump file sizes range from large to *very* large, be sure to use appropriate techniques for transport (e.g., compression and/or CD-R media).

On the analyzing machine, invoke WinDbg. Then choose the menu option File and select Open Crash Dump. Choose the dump file name to open. After the dump file loads, information is displayed as shown in the following excerpt:

```
Kernel Debugger connection established for D:\WINNT\MEMORY.DMP
Kernel Version 2195 Free loaded @ ffffffff80400000
Bugcheck 0000001e : c0000005 f17a123f 00000000 00000000
Stopped at an unexpected exception: code=80000003 addr=8045249c
Hard coded breakpoint hit
...
Module Load: CRASHER.SYS (symbol loading deferred)
```

The initial information reveals the same STOP message information from the original blue screen. The bugcheck code is 0x1E, signifying KMODE_EXCEPTION_NOT_HANDLED. The second bugcheck argument is the address where the problem occurred (0xF17A123F). To see where this instruction falls within the source code, choose Edit, then Goto Address and enter the address from the bugcheck information. If symbol information is located (don't forget to set the symbol path from the Options dialog of the View menu), the source file is opened. For this example, a function that purposefully generates an unhandled exception, **TryToCrash**, is displayed with the cursor placed on the line of code that was executing. The screen shot of [Figure 17.2](#) displays this remarkably helpful feature.

Figure 17.2. Crash dump analysis screen shot.



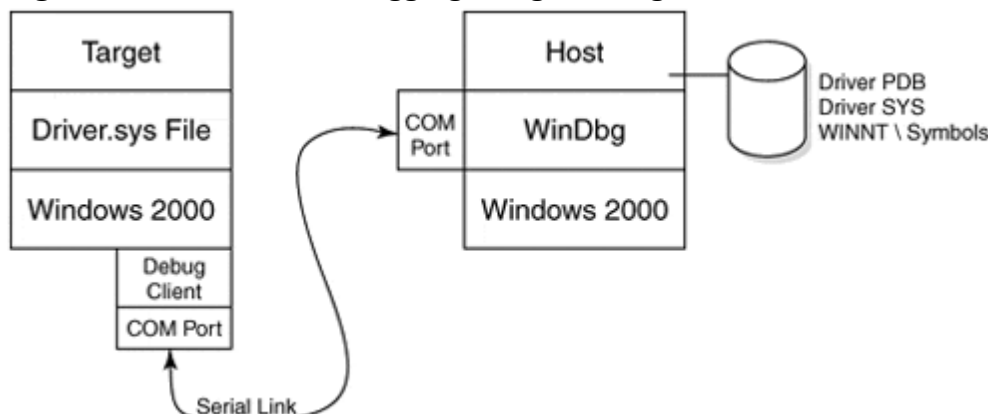
Interactive Debugging

Post mortem analysis is a necessary skill, but many driver problems are easier to diagnose while the driver is actively running. This section briefly describes how to debug driver code interactively.

Starting and Stopping a Debug Session

WinDbg is the primary tool for interactive debugging. To use it, two machines are needed: the target (machine under test) and the host (development machine running WinDbg). [Appendix A](#) describes the basics of making the connection between machines, but essentially a serial link is used to control and monitor activity on the target. [Figure 17.3](#) shows the positioning of the host and target machines and files.

Figure 17.3. Interactive debugging using WinDbg.



The first secret to successful interactive debugging is to ensure that the necessary files are installed on the proper machines. The steps are outlined below.

- 1.
2. Install the binary driver file (.SYS) on the target. The target does not need symbol files or source code.
3. The host requires symbol information for both the driver and the operating system of the target. For simplicity, it is a good idea to ensure that both target and host are running the same version (including service pack) of Windows 2000, but this is not strictly required.
4. On the host, launch WinDbg. Set up the symbols and source directory options to include a path to the appropriate information. Remember, the symbols and source information must match the target's binary file.
5. Select kernel mode debugging from the View menu under the Options Kernel Debugger tab of WinDbg.
6. Select the appropriate COM port and baud rate from this dialog box.
7. Choose the GO toolbar button, or type g in the command window. This places WinDbg into a mode where it waits to connect to the target.
8. Reboot the target machine with the kernel's debug client enabled. (Details are included in [Appendix A](#).) As the system boots, a connection between host and target is established. Once connected, the command window of WinDbg displays considerable information.

The host is now in complete control of the target. The interactive version of a WinDbg session is a superset of the post mortem mode of crash dump analysis. Breakpoints can be established, even on top of code that has yet to be loaded into kernel memory.

To disconnect a debug session, perform the following steps:

1. Pause the target by typing Ctrl-C in the WinDbg command window. You can also press the SYSREQ key on the target.

Writing WinDbg Extensions

One of WinDbg's strengths is that its capabilities can be expanded by writing custom extension commands. This can be quite convenient, particularly for formatting the display of driver-specific data structures. This section explains the process of adding extension commands to WinDbg.

How WinDbg Extensions Work

A WinDbg extension is just a user-mode DLL that exports various commands in the form of DLL functions. The extension DLL also contains several support routines that perform initialization and version-checking operations.

The linkage between target system memory (whether crash file dump or a live target) is established with callback routines that the extension DLL uses to reference the debug target. This means the DLL has the same view of the target system's memory as WinDbg. In particular, extension commands cannot reference memory that was paged out at the time a crash or breakpoint occurs.

Initialization and Version-Checking Functions

To write an extension DLL for WinDbg (or NTSD or KD, for that matter), two DLL export functions are required for initialization. Optionally, a third version-checking function can be provided. These functions are described below.

WinDbgExtensionDllInit

WinDbg calls this function when the user loads the extension DLL. Its job is to save the address of the callback table so that other parts of the DLL can use it. This function (described in [Table 17.3](#)) is required.

Table 17.3. Function Prototype for WinDbgExtensionDllInit

VOID WinDbgExtensionDllInit	
Parameter	Description
PWINDBG_EXTENSION_APIS	Address of table containing pointers to WinDbg
pExtensionApis	callback functions
USHORT MajorVersion	0xF for free build of Windows 2000
	0xC for checked build
USHORT MinorVersion	Build number of Windows 2000
Return value	- void -

ExtensionApiVersion

WinDbg calls this function when it attempts to load an extension DLL. Its purpose is to validate version compatibility between WinDbg and the extension. It does this by returning a pointer to the version structure associated with the extension DLL. This function (shown in [Table 17.4](#)) is required.

Table 17.4. Function Prototype for ExtensionApiVersion

LPEXT_API_VERSION ExtensionApiVersion	
Parameter	Description
Void	None
Return value	Address of the DLL's EXT_API_VERSION structure

CheckVersion

Each time WinDbg executes a command in the DLL, it calls this function before calling the command routine. CheckVersion's job is to ensure that the version of the extension DLL is compatible with the version of Windows 2000 being debugged. If not, an error message should inform the user of the problem. The function (described in [Table 17.5](#)) is optional.

Table 17.5. Function Prototype for CheckVersion

VOID CheckVersion	
Parameter	Description
Void	None
Return value	None

Writing Extension Commands

Code Example: A WinDbg Extension

This example shows how to write a simple WinDbg extension DLL. The code is contained on the accompanying CD and on the book's web site: <http://www.W2KDriverBook.com>. The purpose of the extension is to provide one command: !devext that formats and displays the driver's Device Extension data structure.

DBG.C

All the code for this extension resides in a single source file. Of course, the file #includes other files, but they are sourced from elsewhere, such as the driver.

HEADER

This part of the code contains the definitions for the extension DLL.

```
// The ordering of #include files is important
#include <ntddk.h>
#include <windef.h>

// The following items are from WINBASE.H
// in the Win32 SDK. (WINBASE.H itself can't
// coexist with NTDDK.H, yet we need to be
// able to get at DDK and driver-defined data
// structures and types.)
#define LMEM_FIXED          0x0000
#define LMEM_MOVEABLE      0x0002
#define LMEM_NOCOMPACT     0x0010
#define LMEM_NODISCARD     0x0020
#define LMEM_ZEROINIT      0x0040
#define LMEM_MODIFY        0x0080
#define LMEM_DISCARDABLE   0x0F00
#define LMEM_VALID_FLAGS   0x0F72
#define LMEM_INVALID_HANDLE 0x8000

#define LPTR (LMEM_FIXED | LMEM_ZEROINIT)

#define WINBASEAPI
WINBASEAPI
HLOCAL
WINAPI
LocalAlloc(
    UINT uFlags,
    UINT uBytes
);

WINBASEAPI
HLOCAL
WINAPI
LocalFree(
    HLOCAL hMem
);

#define CopyMemory RtlCopyMemory
#define FillMemory RtlFillMemory
#define ZeroMemory RtlZeroMemory

// Now we can bring in the WINDBG extension
// definitions...
#include <wdbgexts.h>

// Other header files...
#include <stdlib.h>
#include <string.h>
```

Miscellaneous Debugging Techniques

Often the main problem in correcting driver bugs is just getting enough information to make an accurate diagnosis. This section presents a variety of techniques that may help.

Leaving Debugged Code in the Driver

In general, it is a good idea to leave debugging code in place, even after the driver is ready for release. That way, it can be reused if the driver must be modified at some later date. Conditional compilation makes this easy.

The BUILD utility defines a compile-time symbol called **DBG** that can be used to conditionally add debugging code to a driver. In the checked BUILD environment, **DBG** has a value of 1; in the free environment, it has a value of 0. Several of the macros described below use this symbol to suppress the generation of extraneous debugging code in free versions of drivers. When adding debugging code to a driver, it should be wrapped in **#if DBG** and **#endif** directives.

Catching Incorrect Assumptions

As in real life, making unfounded assumptions in kernel-mode drivers is a dangerous practice. For example, assuming that some function argument will always be non-NULL, or that a piece of code is only called at a specific IRQL level can lead to disaster if these expectations are not met.

To catch unforeseen conditions that could lead to driver failure, two things must be done. First, the explicit assumptions made by code must be documented. Second, the assumptions must be verified at runtime. The **ASSERT** and **ASSERTMSG** macros help with both these tasks. They have the following syntax:

```
ASSERT( Expression );
ASSERTMSG( Message, Expression );
```

If *Expression* evaluates to FALSE, **ASSERT** writes a message to WinDbg's command window. The message contains the source code of the failing expression, plus the filename and line number of where the **ASSERT** macro was called. It then provides the option of taking a breakpoint at the point of the **ASSERT**, ignoring the assertion failure, or terminating the process or thread in which the assertion occurred.

ASSERTMSG exhibits the same behavior, except that it includes the text of the *Message* argument with its output. The *Message* argument is just a simple string. Unlike the debug print functions described earlier, **ASSERTMSG** does not allow **printf**-style substitutions.

It should be noted that both assertion macros compile conditionally and disappear altogether in free builds of the driver. This means it is a very bad idea to put any executable code in the *Expression* argument.

Also, the underlying function used by these macros, **RtlAssert**, is a no-op in the free version of Windows 2000 itself. So, to see any assertion failures, a checked build of a driver must be run under a checked version of Windows 2000.

Finally, a warning is in order. The checked build of Windows 2000 crashes with a KMODE_EXCEPTION_NOT_HANDLED error if an assertion fails and the Kernel's debug client is not enabled. If the debug client is enabled, but there is no debugger on the other end of the serial line, the target machine simply hangs if an assertion fails. Recovery can be attempted by starting WinDbg on the host machine, but the text of the assertion that failed is lost.

Using Bugcheck Callbacks

A bugcheck callback is an optional driver routine that is called by the kernel when the system begins to crash. These routines provide a convenient way to capture debugging information at the time of crash. They can also be used to place hardware in a known state before the system goes away. They work as follows:

1.

In **DriverEntry**, use **KeInitializeCallbackRecord** to set up a KBUGCHECK_CALLBACK_RECORD structure. The space for this opaque structure must be nonpaged and must be left untouched until it is released with a call to **KeDeregisterBugCheckCallback**.

2.

Also in **DriverEntry**, call **KeRegisterBugCheckCallback** to request notification when a bugcheck occurs. The arguments to this function include the bugcheck callback record, the address of a callback routine, the address and size of the driver-defined crash buffer, and a string that is used to identify this driver's crash buffer. As with the

Summary

When kernel-mode drivers execute, few limits are placed on what can be done to the system. With such power comes the heavy burden of ensuring that a driver does not compromise system integrity. Driver failures that overtly crash the system, as well as failures that cause more subtle damage, must be detected and corrected. This chapter has presented some techniques for detection, isolation, and elimination of driver failures, both early in the development cycle and later when the driver is distributed to the world.

Appendix A. The Driver Debug Environment

CHAPTER OBJECTIVES

- [Hardware and Software Requirements](#)
- [Debug Symbol Files](#)
- [Enabling Crash Dumps on the Target System](#)
- [Enabling the Target System's Debug Client](#)

The standard Microsoft tools for debugging device drivers require that two systems be used: a host and target. The purpose of this appendix is to describe the details required to successfully set up this debugging environment.

Hardware and Software Requirements

The terminology of the debug environment for drivers should be clear. The target machine executes the driver under test. The host machine, sometimes referred to as the *development system*, runs the debugger (e.g., WinDbg) and therefore controls the operation of the target. The two machines are connected together using a null-modem serial cable.

Another important but confusing term is *debug client*. The target machine must be booted into a special environment whereby it installs the debug client between the serial port and its operating system code. The debug client is the small block of code that allows the host debugger, via the serial port, to control the operation of the target system. The term can be confusing because the target installs and executes the debug client, yet the code makes the target a slave of the host.

For convenience, a network connection between machines is usually established. This ensures that files can be quickly exchanged—providing that both machines are operational. (Never forget that while the target is paused by WinDbg, it cannot participate in network operations.)

Host System

The host system is typically used to compile and link the test driver, and it runs WinDbg as a kernel debugger. Therefore, if a choice exists, choose the machine with the more powerful set of hardware resources as the host system. The list of software that a host should contain follows:

- Windows 2000 retail build
- Visual C++
- Platform SDK
- Windows 2000 DDK
- Symbol files for target OS build(s)
- Driver source code
- Driver symbol file(s)

The host does not necessarily need to execute the same OS version as the target, but it is more convenient. The host requires the symbol file for the OS version running on the target (and it cannot be networked from the target). Thus, if the host and target OS versions do not match, two versions of OS symbols may conceivably be present on the host system. With each launch of WinDbg, thought must be given to which OS symbol file set to use for the session.

Target System

The target system provides the execution environment for the test driver. It is typically configured with the following:

- Windows 2000 retail *and* checked builds with debugging enabled for both
- Driver executable (e.g., .SYS file)
- Driver's hardware
- Full crash dump enabled

Debug Symbol Files

One of the more puzzling aspects of debugging is the role of symbol files. Symbol files falls into two categories: OS and driver.

Operating System Symbols

With the release of each version (i.e., service pack) of Windows 2000, Microsoft releases the public symbols for that build. Public symbols are essentially the information that could have been gleaned from the linker's load map, except that they are in debugger-readable format. The information is read, used, and displayed by the debugger during a stack trace operation, a trace or step operation, or when evaluating and setting breakpoints.

Microsoft does not publish private symbol information for its operating systems. Such information is deemed proprietary, and unless a bug were being chased in the OS itself, it is not particularly helpful. The public symbol information, however, is quite useful and is a recommended installation for application as well as driver debugging.

OS symbols are unique to each build (service pack) of an OS. Always install the symbols for the base OS first by using the Windows 2000 Customer Support Diagnostics CD. Then, after a service pack is applied to the OS, locate the symbol file for the service pack (the location varies based on where the update is obtained) and install it *over* the existing OS symbols.

By default, OS symbols install into a directory %SystemRoot%\Symbols. The debugger must point to appropriate directories for a given debug session. For WinDbg, the symbol path can be configured from the Options dialog of the View menu, Symbols tab.

Driver Symbols

The driver symbols are created when the driver source compiles and links. The symbols for a driver (as for the OS itself) fall into public and private categories. The public symbols provide link map information (correlating function names and addresses), while the private symbols information includes source-line linkage, data type definitions (e.g., typedefs), and static declarations (functions and data).

To generate public symbol information for a driver, the link switch /pdb:"filename" can be used. To generate private symbol information, the compiler switch /Zi is used. The slick Edit and Continue feature should not be used for driver development. Linker and compiler switches can be set for a command-line build (e.g., using the BUILD utility) or from Visual Studio if the IDE is used.

Incidentally, the *generation* of public and private symbols is a separate issue from the *inclusion* of symbol data into a binary distribution file. While it is obvious that a debug (checked) build of a driver should include symbol generation, it is less obvious, but nonetheless critical, that symbol information be generated for a release build. The symbol information (.PDB or .DBG files) should then be archived, but not distributed, with the released binary driver. (The REBASE utility's -x switch can be used to strip a binary (.SYS) file of its symbol data.)

As with the OS symbol data, the debugger must be informed of the path of the driver symbol data.

Enabling Crash Dumps on the Target System

As presented in [Chapter 17](#), crash dumps that occur after a driver failure can be very helpful in isolating and locating bugs. To enable this feature on a target machine, the following steps are required:

1. From the Control Panel, select the System applet.
2. Select the Advanced tab.
3. Choose the Startup and Recovery button.
4. Select either Complete Memory Dump or Kernel Memory Dump from the Write Debugging Information group. It is prudent to also select Overwrite any existing file.
5. Note (or change) the location and filename of the crash dump file. By default, it is %SystemRoot%\MEMORY.DMP.
6. Select OK.
- 7.

Reboot the system, as the changes made to this reconfiguration take effect only with the next boot.

When a crash occurs, the system copies an image of physical memory into the paging file of the root partition. During the next boot, the image is copied into the file specified with Control Panel.

Forcing a Memory Dump With a Keystroke Sequence

By modifying a registry entry, Windows 2000 allows the user to generate a keystroke sequence, Ctrl-Scroll Lock (twice), to force a system Stop with the message:

```
*** STOP: 0x000000E2 (0x00000000,0x00000000,0x00000000,0x00000000)
```

The end-user manually generated the crashdump.

The Registry entry is located in the key:

HKLM\System\CurrentControlSet\Services\i8042prt\Parameters

By adding the value with the name, type, and data shown below, the feature is enabled. For obvious reasons, it is not enabled by default.

Table

Value Name:	CrashOnCtrlScroll
Data Type:	REG_DWORD
Value:	1

Enabling the Target System's Debug Client

Both the retail and checked (debug) versions of Windows 2000 include a debugging client that allows the kernel to communicate over a serial line with the WinDbg debugger. The debug client must be enabled on the target system during the boot process.

To enable the debug client, select the OS for the target machine with cursor keys while the system boot screen is displayed. Press F8 and select (with cursor keys) the option for Debugging Mode. Press Enter twice to boot the system.

By default, the debug client uses the COM2 port, configured for 19200 baud. These are the same defaults used by WinDbg. To use a different port or baud rate, the BOOT.INI file located on the boot partition root must be modified.

Modifying BOOT.INI

The BOOT.INI file is a read-only, hidden file located on the boot partition's root directory. To edit it, these attributes must be removed. Windows Explorer works well, or the ATTRIB utility of the command prompt can be used.

```
attrib -s -h -r BOOT.INI
```

Then the file is modified using any familiar text editor, such as Notepad. The file is similar in structure to an INI file and contains a section labeled **[operating systems]**. Each entry (line) within this section is displayed on the system boot screen as a choice for user selection. (A timeout value located in the **[boot loader]** section specifies how long the user has to make this choice.)

The format of each line of the **[operating systems]** section includes an ARC-style path to a specific drive, partition, and directory, which holds kernel files. An "Description" value specifies the text shown to the user on the boot screen for the selection.

An example of a BOOT.INI file is

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="W2K Pro" /fastdetect
C:\="Microsoft Windows"
```

To specify a boot option for debugging, switches are added to an existing or new **[operating systems]** entry. The relevant switches for debugging are listed in [Table A.3](#).

Table A.3. Debugging Options for BOOT.INI

BOOT.INI Options	
Option	Description
/DEBUG	Enables kernel debug client
/NODEBUG	Disables debug client (default)
/DEBUGPORT= <i>PortName</i>	Specifies COM <i>n</i> port to use
/BAUDRATE= <i>BaudRate</i>	Specifies baud rate for COM port
/CRASHDEBUG	Causes debugger to activate only upon system bugcheck
/SOS	Displays list of modules loaded while system starts
/MAXMEM= <i>MBSize</i>	Specifies maximum memory available to this boot (for testing with smaller memory configurations)

An example entry to allow the user to select a debug client on COM port 4, with a baud rate of 38.4 KBaud, would be:

```
multi(0)disk(0)rdisk(0)partition(2)\WINNT="W2K Debug" /debugport=COM4 /baudrate=38400
```

If two different kernels are installed on the same system, they must reside in different directories. The **[operating systems]** entry would specify the appropriate directory instead of WINNT.

Appendix B. Bugcheck Codes

System Stop messages, also known as bugchecks, display as codes, which designate the reason for the system panic. Depending upon the bugcheck code, up to four parameters provide additional information about the cause and location of the problem. The purpose of this appendix is to provide a list of the more common bugcheck codes that a driver author may encounter. The meanings and common causes of the codes, along with the meanings of the additional parameters, are listed.

Table

Bugcheck 0x0A	IRQL_NOT_LESS_OR_EQUAL
The driver accessed paged memory at DISPATCH_LEVEL or above.	
Parameter	Description
1 align="center"	Memory referenced
2 align="center"	IRQL at time of reference
3 align="center"	0: Read operation
	1: Write operation
4 align="center"	Address that referenced memory

Table

Bugcheck 0x1E	KMODE_EXCEPTION_NOT_HANDLED
A kernel-mode program generated an unhandled exception.	
Parameter	Description
1 align="center"	Exception Code
2 align="center"	Address where exception occurred
3 align="center"	Parameter 0 of exception
4 align="center"	Parameter 1 of exception

Table

Bugcheck 0x24	NTFS_FILE_SYSTEM
A problem occurred in ntfs. sys.	
Parameter	Description
1	Source file and line number
2	Address of the exception record (optional)
3	Address of the context record (optional)
4	Address where the original exception occurred (optional)

Table

Bugcheck 0x2E	DATA_BUS_ERROR
Typically indicates that a parity error in system memory has been detected; usually a hardware problem.	
Parameter	Description
1	Virtual address that caused the fault
2	Physical address that caused the fault
3	Processor status register (PSR)
4	Faulting instruction register (FIR)

Table

Bugcheck 0x35	NO_MORE_IRP_STACK_LOCATIONS
The IoCallDriver packet has no remaining stack locations.	
Parameter	Description
1	Address of the IRP
2–4	Reserved

Table

Bugcheck 0x3F	NO_MORE_SYSTEM_PTES
A fragmented system page table exists	

Appendix C. Building Drivers

CHAPTER OBJECTIVES

-

[The Build Utility](#)

-

[Using Visual Studio to Build Drivers](#)

There is nothing magical about the build process for a driver. Like any piece of "real" (i.e., noninterpreted) code, it must be compiled and linked into a binary file before it can be used. The issues that arise when building drivers concern notifying the build tools (compiler and linker) that the environment for the code being generated is that of kernel mode.

Although it sometimes may appear otherwise, Microsoft supplies a single compiler and linker. Whether the tools are invoked from a command-line prompt or from an elaborate Integrated Development Environment (IDE) such as Visual Studio, the resulting binary files are the same. In many ways, Visual Studio is merely a GUI interface for the build and editing tools. Clicking on the menu option for Project, then Settings, provides a convenient dialog-based interface to the tool switches.

The DDK provides a utility, BUILD, that is convenient for batch and production builds of device drivers (as well as other production-level binaries). With proper environment settings, however, the driver author may successfully use Visual Studio. Indeed, it may be appropriate to use Visual Studio throughout the development phase and introduce BUILD during the later release stages of the project.

The purpose of this appendix is to provide background material on the BUILD utility, as well as provide the Visual Studio settings necessary for successful driver builds.

The Build Utility

The DDK-supplied BUILD utility is ideal for production releases because it works from a single "makefile" concept to batch-build a variety of targeted binaries. For example, a single set of source files can be used to generate driver binaries for both Intel x86 platforms and Alpha platforms. Similarly, both the retail (free) and checked (debug) versions can be built from a single procedure.

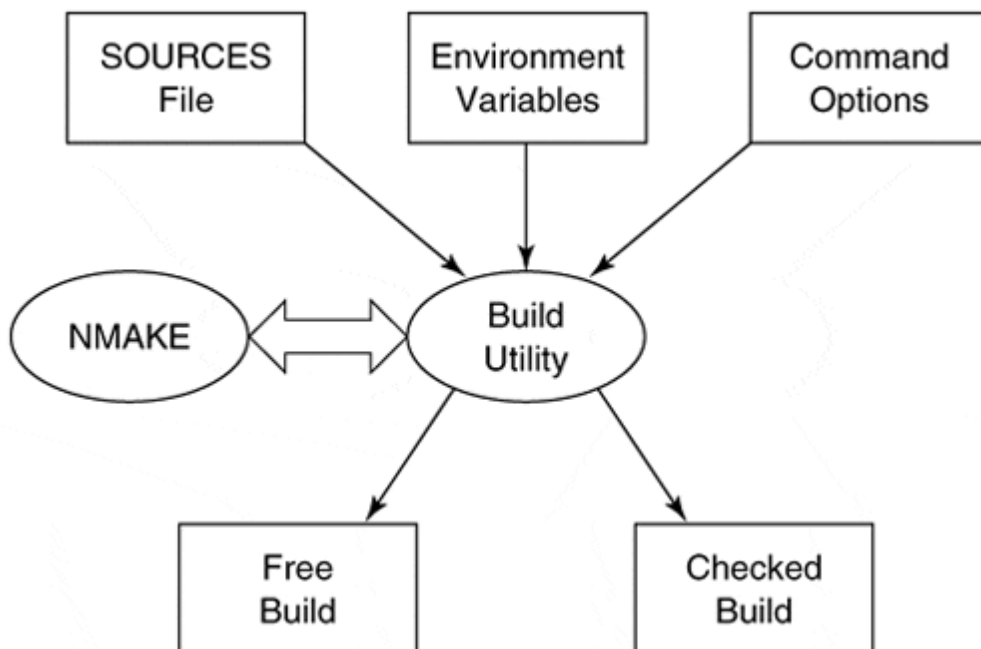
What BUILD Does

The BUILD utility is just an elaborate wrapper around NMAKE. (NMAKE is Microsoft's attempt at the Unix *make* utility.) A batch file of instructions provides a recipe for the construction of one or more target configurations (e.g., checked and free).

Using a set of keywords, a source file list, and miscellaneous switches, BUILD constructs an appropriate *makefile* and then invokes NMAKE to actually perform the build to produce one or more BUILD *products*. [Figure C.1](#) shows how this process works.

BUILD itself is actually a rather simple-minded program. Most of the build process is controlled by a set of standard command files that BUILD passes to NMAKE. These files contain all the platform-specific rules and option settings needed to create a BUILD project. By separating the rules from the utilities, there is less version interdependency.

Figure c.1. The BUILD utility process.



BUILD uses several command files (located in the DDK's binary directory, \BIN:).

-
- MAKEFILES.DEF is the master control file. It references other files to perform its work.
-
- MAKEFILE.PLT selects the platform for a build operation.
-
- i386MK.INC, ALPHAMK.INC, and ia64mk.inc contain platform-specific compiler and linker switches for Intel, Alpha, and 64-bit Intel Itanium processors, respectively.

BUILD helps manage multiplatform projects by separating binary files according to their platform type. To do this, it uses different directories for different platforms. So long as cross-hosted compilers and linkers are available, a single BUILD process can generate binaries for all the supported platforms. The directory structure for the output files takes the form shown in [Figure C.2](#).

Notice that BUILD uses separate directories for the checked and free versions of the generated binaries. The **DBG** symbol is defined as 1 when performing the checked build.

How to Build a Driver with BUILD

Once the source code is ready for compilation, the use of the BUILD utility requires that the following steps be followed:

- 1.

Using Visual Studio to Build Drivers

Although Visual Studio is quite obviously promoted as an application development tool for C++ (and several other languages), it is nonetheless an IDE for the standard compiler and linker. Therefore, using the proper compile and link switch settings, the tool can be used to build kernel-mode device drivers.

The DDAppWiz Tool

The CD that accompanies this book includes an App Wizard that conveniently provides the necessary compile and link switch settings for Visual Studio to build a device driver. The file, **DDAppWiz.AWX**, must be copied into the Visual Studio path: ...\\Microsoft Visual Studio\\Common\\MSDev98\\Template. Thereafter, a new Project type appears in the New Project dialog box: W2K Device Driver.

The use of the App Wizard is straightforward and described in [Chapter 6](#). This section lists the Visual Studio environment settings modified or added by the wizard.

Preprocessor Symbols Modification

The table below shows the compiler preprocessor symbols removed from the standard Win32 project, normally generated by Visual Studio.

Table

Preprocessor Symbol	Modification	Reason
WIN32	Deleted	Driver is not a Win32 application
_WINDOWS	Deleted	Driver is not a Windows application
_MBCS	Deleted	Driver uses Unicode, not multibyte character set
_DEBUG	Deleted	Driver is not Win32 debug target
DBG	Added	Driver's standard debug symbol definition (1=checked, 0=free)
X86	Added	DDAppWiz produces driver for Intel platform
WIN32_WINNT=0x500	Added	Version of Windows 2000

Compiler Switch Modifications

Several compile switches are required for driver builds, as shown in the following table:

Table

Compiler Switch	Modification	Reason
/GX and -GX	Deleted	Do not enable synchronous exception handling
/GZ	Deleted	Do not attempt to catch release build errors
-Gz	Added	Use __stdcall calling convention

Compiler Include Directories

The Device Driver App Wizard adds two include directories to the set of compiler options (-I switch). The paths are listed below.

\\NTDDK\\inc and
\\NTDDK\\inc\\ddk

These may require modification, depending upon choices specified when the DDK is installed.

Link Modifications

All of the standard Win32 libraries are removed from linker input and replaced with the following list of libraries:

Jerry Lozano, Art Baker

ART BAKER, founder of Cydonix Corporation, has spent 30 years writing device-control software. For 15 years, he has trained professional developers to write device drivers.

JERRY LOZANO is an Electrical Engineer and has been writing low-level device and OS code since 1976. For several years, he has been training and consulting on Microsoft system tools and development.

Bibliography

AndersonDon . *FireWire System Architecture*. Reading, Masschusetts, Addison-Wessley, 1999. ISBN 0-201-48535-4

SolomonDavid . *Inside Windows NT—Second Edition*. Redmond, Washington, 1998. ISBN 1-57231-677-2

RichterJeffrey . *Advanced Windows*. Redmond, Washington, Microsoft Press, 1997. ISBN 1-57231-548-2

BrainMarshall . *Win32 System Services*. Upper Saddle, New Jersey, Prentice Hall PTR, 1995. ISBN 0-13-324732-5

AndersonDon Mindshare Inc. *Universal Serial Bus System Architecture*. Reading, Massachusetts, Addison Wesley Longman 1997. ISBN 0-201-46137-4

IEEE 1394 Specifications and white papers. (<http://www.1394ta.org>)

USB Specifications and white papers (<http://www.usb.org>)

Windows 2000 DDK (<http://www.microsoft.com/ddk>)

Windows 2000 Platform SDK (<http://msdn.microsoft.com/library/psdk/portals/mainport.htm>)

Windows 2000 IFS Kit (<http://www.microsoft.com/ddk/IFSKit>)