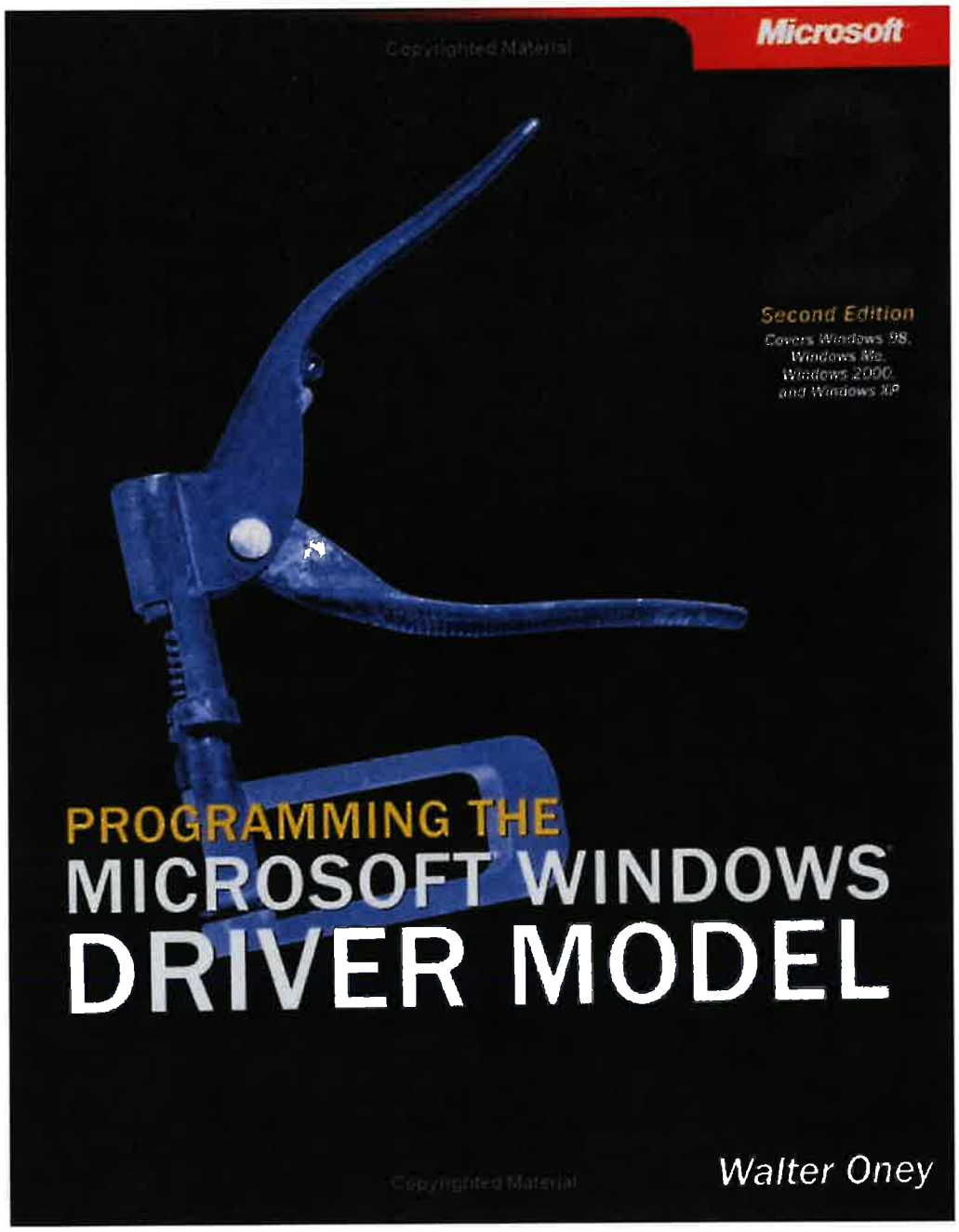

Microsoft
Copyrighted Material
Second Edition
Covers Windows 98, Windows Me, Windows 2000, and Windows XP
PROGRAMMING THE
MICROSOFT WINDOWS
DRIVER MODEL
Copyrighted Material
Walter Oney

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2003 by Walter Oney

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Oney, Walter.
Programming the Microsoft Windows Driver Model / Walter Oney -- 2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1803-8
1. Microsoft Windows NT device drivers (Computer programs) 2. Computer programming. I. Title.

QA76.76.D49 O54 2002
005.7'126--dc21 2002038650

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 8 7 6 5 4 3

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Klingon font Copyright 2002, Klingon Language Institute. Active Directory, DirectX, Microsoft, MSDN, MS-DOS, Visual C++, Visual Studio, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: Juliana Aldous
Project Editor: Dick Brown
Technical Editor: Jim Fuchs

Table of Contents

Acknowledgments IX

Introduction X

1 Beginning a Driver Project - 1 -

1.1 A Brief History of Device Drivers - 1 -

1.2 An Overview of the Operating Systems - 3 -

1.2.1 Windows XP Overview - 3 -

1.2.2 Windows 98/Windows Me Overview - 4 -

1.3 What Kind of Driver Do I Need? - 6 -

1.3.1 WDM Drivers - 7 -

1.3.2 Other Types of Drivers - 8 -

1.3.3 Management Overview and Checklist - 9 -

2 Basic Structure of a WDM Driver - 11 -

2.1 How Drivers Work - 11 -

2.1.1 How Applications Work - 11 -

2.1.2 Device Drivers - 12 -

2.2 How the System Finds and Loads Drivers - 14 -

2.2.1 Device and Driver Layering - 14 -

2.2.2 Plug and Play Devices - 15 -

2.2.3 Legacy Devices - 17 -

2.2.4 Recursive Enumeration - 18 -

2.2.5 Order of Driver Loading - 19 -

2.2.6 IRP Routing - 20 -

2.3 The Two Basic Data Structures - 23 -

2.3.1 Driver Objects - 23 -

2.3.2 Device Objects - 25 -

2.4 The *DriverEntry* Routine - 27 -

2.4.1 Overview of *DriverEntry* - 28 -

2.4.2 DriverUnload - 29 -

2.5 The *AddDevice* Routine - 29 -

2.5.1 Creating a Device Object - 30 -

2.5.2 Naming Devices - 31 -

2.5.3 Other Global Device Initialization - 39 -

2.5.4 Putting the Pieces Together - 42 -

2.6 Windows 98/Me Compatibility Notes - 43 -

2.6.1 Differences in *DriverEntry* Call - 43 -

2.6.2 DriverUnload - 43 -

2.6.3 The \GLOBAL?? Directory - 43 -

2.6.4 Unimplemented Device Types - 43 -

3 Basic Programming Techniques - 45 -

3.1 The Kernel-Mode Programming Environment - 45 -

3.1.1 Using Standard Run-Time Library Functions - 46 -

3.1.2 A Caution About Side Effects - 46 -

IPR2024-00292
KASPERSKY EX 1008 Page 3

3.2 Error Handling - 46 -
3.2.1 Status Codes - 47 -
3.2.2 Structured Exception Handling - 48 -
3.2.3 Bug Checks - 54 -
3.3 Memory Management - 55 -
3.3.1 User-Mode and Kernel-Mode Address Spaces - 55 -
3.3.2 Heap Allocator - 60 -
3.3.3 Linked Lists - 64 -
3.3.4 Lookaside Lists - 67 -
3.4 String Handling - 69 -
3.5 Miscellaneous Programming Techniques - 71 -
3.5.1 Accessing the Registry - 71 -
3.5.2 Accessing Files - 76 -
3.5.3 Floating-Point Calculations - 78 -
3.5.4 Making Debugging Easier - 79 -
3.6 Windows 98/Me Compatibility Notes - 82 -
3.6.1 File I/O - 82 -
3.6.2 Floating Point - 82 -
4 Synchronization - 83 -
4.1 An Archetypal Synchronization Problem - 83 -
4.2 Interrupt Request Level - 85 -
4.2.1 *IRQL* in Operation - 86 -
4.2.2 IRQL Compared with Thread Priorities - 86 -
4.2.3 IRQL and Paging - 87 -
4.2.4 Implicitly Controlling IRQL - 87 -
4.2.5 Explicitly Controlling IRQL - 88 -
4.3 Spin Locks - 88 -
4.3.1 Some Facts About Spin Locks - 89 -
4.3.2 Working with Spin Locks - 89 -
4.3.3 Queued Spin Locks - 90 -
4.4 Kernel Dispatcher Objects - 91 -
4.4.1 How and When You Can Block - 91 -
4.4.2 Waiting on a Single Dispatcher Object - 92 -
4.4.3 Waiting on Multiple Dispatcher Objects - 93 -
4.4.4 Kernel Events - 94 -
4.4.5 Kernel Semaphores - 96 -
4.4.6 Kernel Mutexes - 97 -
4.4.7 Kernel Timers - 98 -
4.4.8 Using Threads for Synchronization - 101 -
4.4.9 Thread Alerts and APCs - 102 -
4.5 Other Kernel-Mode Synchronization Primitives - 104 -
4.5.1 Fast Mutex Objects - 104 -
4.5.2 Interlocked Arithmetic - 106 -
4.5.3 Interlocked List Access - 108 -
4.5.4 Windows 98/Me Compatibility Notes - 110 -
5 The I/O Request Packet - 111 -
5.1 Data Structures - 111 -
5.1.1 Structure of an IRP - 111 -

IPR2024-00292
KASPERSKY EX 1008 Page 4

5.1.2 The I/O Stack - 113 -

5.2 The "Standard Model" for IRP Processing - 114 -

5.2.1 Creating an IRP - 114 -

5.2.2 Forwarding to a Dispatch Routine - 116 -

5.2.3 Duties of a Dispatch Routine - 119 -

5.2.4 The StartIo Routine - 123 -

5.2.5 The Interrupt Service Routine - 124 -

5.2.6 Deferred Procedure Call Routine - 124 -

5.3 Completion Routines - 125 -

5.4 Queuing I/O Requests - 132 -

5.4.1 Using the DEVQUEUE Object - 134 -

5.4.2 Using Cancel-Safe Queues - 136 -

5.5 Cancelling I/O Requests - 140 -

5.5.1 If It Weren't for Multitasking... - 140 -

5.5.2 Synchronizing Cancellation - 140 -

5.5.3 Some Details of IRP Cancellation - 141 -

5.5.4 How the DEVQUEUE Handles Cancellation - 142 -

5.5.5 Cancelling IRPs You Create or Handle - 146 -

5.5.6 Handling *IRP_MJ_CLEANUP* - 151 -

5.5.7 Cleanup with a *DEVQUEUE* - 152 -

5.5.8 Cleanup with a Cancel-Safe Queue - 153 -

5.6 Summary—Eight IRP-Handling Scenarios - 153 -

5.6.1 Scenario 1—Pass Down with Completion Routine - 154 -

5.6.2 Scenario 2—Pass Down Without Completion Routine - 154 -

5.6.3 Scenario 3—Complete in the Dispatch Routine - 155 -

5.6.4 Scenario 4—Queue for Later Processing - 156 -

5.6.5 Scenario 5—Your Own Asynchronous IRP - 157 -

5.6.6 Scenario 6—Your Own Synchronous IRP - 158 -

5.6.7 Scenario 7—Synchronous Pass Down - 159 -

5.6.8 Scenario 8—Asynchronous IRP Handled Synchronously - 160 -

6 Plug and Play for Function Drivers - 163 -

6.1 *IRP_MJ_PNP* Dispatch Function - 164 -

6.2 Starting and Stopping Your Device - 165 -

6.2.1 IRP_MN_START_DEVICE - 166 -

6.2.2 IRP_MN_STOP_DEVICE - 167 -

6.2.3 IRP_MN_REMOVE_DEVICE - 168 -

6.2.4 IRP_MN_SURPRISE_REMOVAL - 169 -

6.3 Managing PnP State Transitions - 169 -

6.3.1 Starting the Device - 171 -

6.3.2 Is It OK to Stop the Device? - 171 -

6.3.3 While the Device Is Stopped - 172 -

6.3.4 Is It OK to Remove the Device? - 173 -

6.3.5 Synchronizing Removal - 174 -

6.3.6 Why Do I Need This @#$! Remove Lock, Anyway? - 176 -

6.3.7 How the *DEVQUEUE* Works with PnP - 178 -

6.4 Other Configuration Functionality - 181 -

6.4.1 Filtering Resource Requirements - 181 -

6.4.2 Device Usage Notifications - 182 -

IPR2024-00292
KASPERSKY EX 1008 Page 5

6.4.3 PnP Notifications - 184 -
6.5 Windows 98/Me Compatibility Notes - 191 -
6.5.1 Surprise Removal - 191 -
6.5.2 PnP Notifications - 191 -
6.5.3 The Remove Lock - 191 -
7 Reading and Writing Data - 193 -
7.1 Configuring Your Device - 193 -
7.2 Addressing a Data Buffer - 195 -
7.2.1 Specifying a Buffering Method - 195 -
7.3 Ports and Registers - 198 -
7.3.1 Port Resources - 200 -
7.3.2 Memory Resources - 201 -
7.4 Servicing an Interrupt - 201 -
7.4.1 Configuring an Interrupt - 202 -
7.4.2 Handling Interrupts - 203 -
7.4.3 Deferred Procedure Calls - 204 -
7.4.4 A Simple Interrupt-Driven Device - 207 -
7.5 Direct Memory Access - 211 -
7.5.1 Transfer Strategies - 212 -
7.5.2 Performing DMA Transfers - 213 -
7.5.3 Using a Common Buffer - 222 -
7.5.4 A Simple Bus-Master Device - 224 -
7.6 Windows 98/Me Compatibility Notes - 225 -
8 Power Management - 227 -
8.1 The WDM Power Model - 227 -
8.1.1 The Roles of WDM Drivers - 227 -
8.1.2 Device Power and System Power States - 227 -
8.1.3 Power State Transitions - 228 -
8.1.4 Handling *IRP_MJ_POWER* Requests - 229 -
8.2 Managing Power Transitions - 231 -
8.2.1 Required Infrastructure - 232 -
8.2.2 Initial Triage - 233 -
8.2.3 System Power IRPs That Increase Power - 233 -
8.2.4 System Power IRPs That Decrease Power - 238 -
8.2.5 Device Power IRPs - 238 -
8.2.6 Flags to Set in *AddDevice* - 244 -
8.3 Additional Power-Management Details - 245 -
8.3.1 Device Wake-Up Features - 245 -
8.3.2 Powering Off When Idle - 250 -
8.3.3 Using Sequence Numbers to Optimize State Changes - 252 -
8.4 Windows 98/Me Compatibility Notes - 253 -
8.4.1 The Importance of *DO_POWER_PAGABLE* - 253 -
8.4.2 Completing Power IRPs - 253 -
8.4.3 Requesting Device Power IRPs - 253 -
8.4.4 PoCallDriver - 253 -
8.4.5 Other Differences - 253 -
9 I/O Control Operations - 255 -

IPR2024-00292
KASPERSKY EX 1008 Page 6

9.1 The *DeviceIoControl* API - 255 -
9.1.1 Synchronous and Asynchronous Calls to *DeviceIoControl* - 256 -
9.1.2 Defining I/O Control Codes - 257 -
9.2 Handling IRP_MJ_DEVICE_CONTROL - 258 -
9.3 METHOD_BUFFERED - 259 -
9.3.1 The *DIRECT* Buffering Methods - 260 -
9.3.2 METHOD_NEITHER - 261 -
9.3.3 Designing a Safe and Secure IOCTL Interface - 262 -
9.4 Internal I/O Control Operations - 263 -
9.5 Notifying Applications of Interesting Events - 264 -
9.5.1 Using a Shared Event for Notification - 265 -
9.5.2 Using a Pending IOCTL for Notification - 266 -
9.6 Windows 98/Me Compatibility Notes - 269 -
10 Windows Management Instrumentation - 271 -
10.1 WMI Concepts - 271 -
10.1.1 A Sample Schema - 272 -
10.1.2 Mapping WMI Classes to C Structures - 273 -
10.2 WDM Drivers and WMI - 274 -
10.2.1 Delegating IRPs to WMILIB - 275 -
10.2.2 Advanced Features - 280 -
10.3 Windows 98/Me Compatibility Notes - 285 -
11 Controller and Multifunction Devices - 287 -
11.1 Overall Architecture - 287 -
11.1.1 Child Device Objects - 287 -
11.2 Handling PnP Requests - 289 -
11.2.1 Telling the PnP Manager About Our Children - 290 -
11.2.2 PDO Handling of PnP Requests - 290 -
11.2.3 Handling Device Removal - 293 -
11.2.4 Handling *IRP_MN_QUERY_ID* - 293 -
11.2.5 Handling *IRP_MN_QUERY_DEVICE_RELATIONS* - 294 -
11.2.6 Handling *IRP_MN_QUERY_INTERFACE* - 294 -
11.2.7 Handling *IRP_MN_QUERY_PNP_DEVICE_STATE* - 297 -
11.3 Handling Power Requests - 297 -
11.4 Handling Child Device Resources - 299 -
12 The Universal Serial Bus - 301 -
12.1 Programming Architecture - 301 -
12.1.1 Device Hierarchy - 302 -
12.1.2 What's in a Device? - 303 -
12.1.3 Information Flow - 304 -
12.1.4 Descriptors - 310 -
12.2 Working with the Bus Driver - 315 -
12.2.1 Initiating Requests - 315 -
12.2.2 Configuration - 317 -
12.2.3 Managing Bulk Transfer Pipes - 323 -
12.2.4 Managing Interrupt Pipes - 329 -
12.2.5 Control Requests - 329 -
12.2.6 Managing Isochronous Pipes - 331 -

IPR2024-00292
KASPERSKY EX 1008 Page 7

12.2.7 Idle Power Management for USB Devices - 340 -
13 Human Interface Devices - 343 -
13.1 Drivers for HID Devices - 343 -
13.2 Reports and Report Descriptors - 343 -
13.2.1 Sample Keyboard Descriptor - 343 -
13.2.2 HIDFAKE Descriptor - 345 -
13.3 *HIDCLASS* Minidrivers - 346 -
13.3.1 DriverEntry - 346 -
13.3.2 Driver Callback Routines - 347 -
13.3.3 Internal IOCTL Interface - 351 -
13.3.4 IOCTL_HID_GET_DEVICE_DESCRIPTOR - 353 -
13.4 Windows 98/Me Compatibility Notes - 360 -
13.4.1 Handling *IRP_MN_QUERY_ID* - 360 -
13.4.2 Joysticks - 361 -
14 Specialized Topics - 363 -
14.1 Logging Errors - 363 -
14.1.1 Creating an Error Log Packet - 364 -
14.1.2 Creating a Message File - 365 -
14.2 System Threads - 367 -
14.2.1 Creating and Terminating System Threads - 368 -
14.2.2 Using a System Thread for Device Polling - 369 -
14.3 Work Items - 371 -
14.3.1 Watchdog Timers - 372 -
14.4 Windows 98/Me Compatibility Notes - 375 -
14.4.1 Error Logging - 375 -
14.4.2 Waiting for System Threads to Finish - 375 -
14.4.3 Work Items - 375 -
15 Distributing Device Drivers - 377 -
15.1 The Role of the Registry - 377 -
15.1.1 The Hardware (Instance) Key - 378 -
15.1.2 The Class Key - 379 -
15.1.3 The Driver Key - 380 -
15.1.4 The Service (Software) Key - 380 -
15.1.5 Accessing the Registry from a Program - 381 -
15.1.6 Device Object Properties - 382 -
15.2 The INF File - 383 -
15.2.1 Install Sections - 386 -
15.2.2 Populating the Registry - 388 -
15.2.3 Security Settings - 391 -
15.2.4 Strings and Localization - 392 -
15.2.5 Device Identifiers - 392 -
15.2.6 Driver Ranking - 396 -
15.2.7 Tools for INF Files - 397 -
15.3 Defining a Device Class - 399 -
15.3.1 A Property Page Provider - 400 -
15.4 Customizing Setup - 402 -
15.4.1 Installers and Co-installers - 403 -

IPR2024-00292
KASPERSKY EX 1008 Page 8

15.4.2 Preinstalling Driver Files - 407 -
15.4.3 Value-Added Software - 407 -
15.4.4 Installing a Driver Programmatically - 408 -
15.4.5 The *RunOnce* Key - 408 -
15.4.6 Launching an Application - 409 -
15.5 The Windows Hardware Quality Lab - 409 -
15.5.1 Running the Hardware Compatibility Tests - 409 -
15.5.2 Submitting a Driver Package - 413 -
15.6 Windows 98/Me Compatibility Notes - 417 -
15.6.1 Property Page Providers - 417 -
15.6.2 Installers and Co-installers - 417 -
15.6.3 Preinstalled Driver Packages - 417 -
15.6.4 Digital Signatures - 418 -
15.6.5 Installing Drivers Programmatically - 418 -
15.6.6 CONFIGMG API - 418 -
15.6.7 INF Incompatibilities - 418 -
15.6.8 Registry Usage - 418 -
15.7 Getting Device Properties - 419 -
16 Filter Drivers - 421 -
16.1 The Role of a Filter Driver - 421 -
16.1.1 Upper Filter Drivers - 421 -
16.1.2 Lower Filter Drivers - 423 -
16.2 Mechanics of a Filter Driver - 424 -
16.2.1 The *DriverEntry* Routine - 424 -
16.2.2 The *AddDevice* Routine - 424 -
16.2.3 The *DispatchAny* Function - 426 -
16.2.4 The *DispatchPower* Routine - 426 -
16.2.5 The *DispatchPnp* Routine - 427 -
16.3 Installing a Filter Driver - 428 -
16.3.1 Installing a Class Filter - 428 -
16.3.2 Installing a Device Filter with the Function Driver - 430 -
16.3.3 Installing a Device Filter for an Existing Device - 430 -
16.4 Case Studies - 430 -
16.4.1 Lower Filter for Traffic Monitoring - 430 -
16.4.2 Named Filters - 431 -
16.4.3 Bus Filters - 433 -
16.4.4 Keyboard and Mouse Filters - 433 -
16.4.5 Filtering Other HID Devices - 435 -
16.5 Windows 98/Me Compatibility Notes - 435 -
16.5.1 WDM Filters for VxD Drivers - 435 -
16.5.2 INF Shortcut - 436 -
16.5.3 Class Filter Drivers - 436 -
A Coping with Cross-Platform Incompatibilities - 437 -
Determining the Operating System Version - 437 -
Run-Time Dynamic Linking - 437 -
Checking Platform Compatibility - 438 -
Defining Win98/Me Stubs for Kernel-Mode Routines - 439 -
Version Compatibility - 440 -

IPR2024-00292
KASPERSKY EX 1008 Page 9

Stub Functions - 440 -
Using WDMSTUB - 442 -
Interaction Between WDMSTUB and WDMCHECK - 442 -
Special Licensing Note - 442 -
B Using WDMWIZ.AWX - 443 -
Basic Driver Information - 443 -
***DeviceIoControl* Codes - 444 -**
I/O Resources - 445 -
USB Endpoints - 445 -
WMI Support - 446 -
Parameters for the INF File - 447 -
Now What? - 448 -

IPR2024-00292
KASPERSKY EX 1008 Page 10

Acknowledgments

Many people helped me write this book. At the beginning of the project, Anne Hamilton, Senior Acquisitions Editor at Microsoft Press, had the vision to realize that a revision of this book was needed. Juliana Aldous, the Acquisitions Editor, shepherded the project through to the complete product you're holding in your hands. Her team included Dick Brown, Jim Fuchs, Shawn Peck, Rob Nance, Sally Stickney, Paula Gorelick, Elizabeth Hansford, and Julie Kawabata. That the grammar and diction in the book are correct, that the figures are correctly referenced and intelligible, and that the index accurately correlates with the text are due to all of them.

Marc Reinig and Dr. Lawrence M. Schoen provided valuable assistance with a linguistic and typographical issue.

Mike Tricker of Microsoft deserves special thanks for championing my request for a source code license, as does Brad Carpenter for his overall support of the revision project.

Eliyas Yakub acted as the point man to obtain technical reviews of the content of the book and to facilitate access to all sorts of resources within Microsoft. Among the developers and managers who took time from busy schedules to make sure that this book would be as accurate as possible are—in no particular order—Adrian Oney (no relation, but I'm fond of pointing out his vested interest in a book that has his name on the spine), Allen Marshall, Scott Johnson, Martin Borve, Jean Valentine, Doron Holan, Randy Aull, Jake Oshins, Neill Clift, Narayanan Ganapathy, Fred Bhesania, Gordan Lacey, Alan Warwick, Bob Fruth, and Scott Herrboldt.

Lastly, my wife, Marty, provided encouragement and support throughout the project.

IPR2024-00292
KASPERSKY EX 1008 Page 11

Introduction

This book explains how to write device drivers for the newest members of the Microsoft Windows family of operating systems using the Windows Driver Model (WDM). In this Introduction, I'll explain who should be reading this book, the organization of the book, and how to use the book most effectively. You'll also find a note on errors and a section on other resources you can use to learn about driver programming. Looking ahead, Chapter 1 explains how the two main branches of the Windows family operate internally, what a WDM device driver is, and how it relates to the rest of Windows.

Who Should Read This Book

I've aimed this book at experienced programmers who don't necessarily know anything about writing device drivers for Windows operating systems. This book is for you if you want to learn how to do that. To succeed at driver writing, you will need to understand the C programming language very well because WDM drivers are written in C. You'll also need to be exceptionally able to tolerate ambiguity and to reverse-engineer portions of the operating system because a good deal of trial and error in the face of incomplete or inaccurate information is required.

Writing a WDM driver is much like writing a kernel-mode driver for Windows NT4.0. It's a bit easier because you don't have to detect and configure your own hardware. Ironically, it's simultaneously harder because correctly handling Plug and Play and power management is fiendishly difficult. If you've written kernel-mode drivers for Windows NT, you'll have no trouble at all reading this book. You'll also be glad to have some code samples that you can cut and paste to deal with the aforementioned fiendishly difficult areas.

Writing a WDM driver is completely unlike writing a virtual device driver (VxD) for Windows 3.0 and its successors, a UNIX driver, or a real-mode driver for MS-DOS. If your experience lies in those areas, expect to work hard learning this new technology. Nonetheless, I think programming WDM drivers is easier than programming those other drivers because you have more rules to follow, leading to fewer choices between confusing alternatives. Of course, you have to learn the rules before you can benefit from that fact.

If you already own a copy of the first edition of this book and are wondering whether you should buy this revised edition, here's a bit of information to help you decide. Windows XP and Windows Me made few changes in the way you develop drivers for Windows 2000 and Windows 98, respectively. The main reason we decided to revise this book is that so many changes had accumulated on my update/errata Web page. This edition does, of course, explain some of the new bells and whistles that Windows XP brings with it. It contains more explicit advice about writing robust, secure drivers. It also, frankly, explains some things much better than the first edition does.

Chapter 1 has some information that will be useful to development managers and others who need to plan hardware projects. It's very embarrassing to be brought up short near the end of a hardware development project by the realization that you need a driver. Sometimes you'll be able to find a generic driver that will handle your hardware. Often, however, such a driver won't exist and you'll need to write one yourself. I hope to convince you managers in the first chapter that writing drivers is pretty hard and deserves your attention earlier rather than later. When you're done reading that chapter, by the way, give the book to the person who's going to carry the oar. And buy lots more copies. (As I told one of my college friends, you can always use the extra copies as dining room chair extenders for a young family.)

Organization of This Book

After teaching driver programming seminars for many years, I've come to understand that people learn things in fundamentally different ways. Some people like to learn a great deal of theory about something and then learn how to apply that theory to practical problems. Other people like to learn practical things first and then learn the general theory. I call the former approach deductive and the latter approach inductive. I personally prefer an inductive approach, and I've organized this book to suit that style of learning.

My aim is to explain how to write device drivers. Broadly speaking, I want to provide the minimum background you'll need to write an actual driver and then move on to more specialized topics. That "minimum background" is pretty extensive, however; it consumes seven chapters. Once past Chapter 7, you'll be reading about topics that are important but not necessarily on the fall line that leads straight downhill to a working driver.

Chapter 1, "Beginning a Driver Project," as I've mentioned, describes WDM device drivers and how they relate to Windows itself. Along the way, I'll relate the story of how we got to where we are today in operating system and driver technology. The chapter also explains how to choose the kind of driver you need, provides an overview and checklist specifically for development managers, and addresses the issue of binary compatibility.

Chapter 2, "Basic Structure of a WDM Driver," explains the basic data structures that Windows 2000 uses to manage I/O devices and the basic way your driver relates to those data structures. I'll discuss the driver object and the device object. I'll also discuss how you write two of the subroutines—the *DriverEntry* and *AddDevice* routines—that every WDM driver package contains.

IPR2024-00292
KASPERSKY EX 1008 Page 12

Chapter 3, "Basic Programming Techniques," describes the most important service functions you can call on to perform mundane programming tasks. In that chapter, I'll discuss error handling, memory management, and a few other miscellaneous tasks.

Chapter 4, "Synchronization," discusses how your driver can synchronize access to shared data in the multitasking, multiprocessor world of Windows XP. You'll learn the details about interrupt request level (IRQL) and about various synchronization primitives that the operating system offers for your use.

Chapter 5, "The I/O Request Packet," introduces the subject of input/output programming, which of course is the real reason for this book. I'll explain where I/O request packets come from, and I'll give an overview of what drivers do with them when they follow what I call the "standard model" for IRP processing. I'll also discuss the knotty subject of IRP queuing and cancellation, wherein accurate reasoning about synchronization problems becomes crucial.

Chapter 6, "Plug and Play for Function Drivers," concerns just one type of I/O request packet, namely IRP_MJ_PNP. The Plug and Play Manager component of the operating system sends you this IRP to give you details about your device's configuration and to notify you of important events in the life of your device.

Chapter 7, "Reading and Writing Data," is where we finally get to write driver code that performs I/O operations. I'll discuss how you obtain configuration information from the PnP Manager and how you use that information to prepare your driver for "substantive" IRPs that read and write data. I'll present two simple driver sample programs as well: one for dealing with a PIO device and one for dealing with a bus-mastering DMA device.

Chapter 8, "Power Management," describes how your driver participates in power management. I think you'll find, as I did, that power management is pretty complicated. Unfortunately, you have to participate in the system's power management protocols, or else the system as a whole won't work right. Luckily, the community of driver writers already has a grand tradition of cutting and pasting, and that will save you.

Chapter 9, "I/O Control Operations," contains a discussion of this important way for applications and other drivers to communicate "out of band" with your driver.

Chapter 10, "Windows Management Instrumentation," concerns a scheme for enterprisewide computer management in which your driver can and should participate. I'll explain how you can provide statistical and performance data for use by monitoring applications, how you can respond to standard WMI controls, and how you can alert controlling applications of important events when they occur.

Chapter 11, "Controller and Multifunction Devices," discusses how to write a driver for a device that embodies multiple functions, or multiple instances of the same function, in one physical device.

Chapter 12, "The Universal Serial Bus," describes how to write drivers for USB devices.

Chapter 13, "Human Interface Devices," explains how to write a driver for this important class of devices.

Chapter 14, "Specialized Topics," describes system threads, work items, error logging, and other special programming topics.

Chapter 15, "Distributing Device Drivers," tells you how to arrange for your driver to get installed on end user systems. You'll learn the basics of writing an INF file to control installation, and you'll also learn some interesting and useful things to do with the system registry. This is where to look for information about WHQL submissions too.

Chapter 16, "Filter Drivers," discusses when you can use filter drivers to your advantage and how to build and install them.

Appendix A, "Coping with Cross-Platform Incompatibilities," explains how to determine which version of the operating system is in control and how to craft a binary-compatible driver.

Appendix B, "Using WDMWIZ.AWX," describes how to use my Visual C++ application wizard to build a driver. WDMWIZ.AWX is not intended to take the place of a commercial toolkit. Among other things, that means that it's not easy enough to use that you can dispense with documentation.

Driver Security and Reliability

Software security and reliability is everybody's job. Those of us who write drivers have a special responsibility because our code runs in the trusted kernel. When our code crashes, it usually takes the whole system with it. When our code has a trap door, a hacker can squeeze through to take over the whole system and, perhaps, the enterprise it serves. It behooves all of us to take these issues seriously. If we don't, real people can suffer economic and physical injury.



Because of the seriousness of security issues in driver programming, this edition uses a special icon to highlight areas that are especially important to driver reliability and security.



The Driver Verifier component of the operating system performs a variety of checks on a driver—if we ask it to. The Windows Hardware Quality Laboratory (WHQL) will run your driver with all sorts of Driver Verifier tests enabled, so you might as well beat them to it by enabling Driver Verifier as soon as your driver is minimally functional. We'll use this icon to mark discussions of how the Driver Verifier can help you debug your driver.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 13

Sample Files

You can find sample files for this book at the Microsoft Press Web site at http://www.microsoft.com/mspress/books/6262.asp. Clicking the Companion Content link takes you to a page from which you can download the sa mples. You can also find the files on the book's companion CD.

This book's companion content contains a great many sample drivers and test programs. I crafted each sample with a view toward illustrating a particular issue or technique that the text discusses. Each of the samples is, therefore, a "toy" that you can't just ship after changing a few lines of code. I wrote the samples this way on purpose. Over the years, I've observed that programmer-authors tend to build samples that illustrate their prowess at overcoming complexity rather than samples that teach beginners how to solve basic problems, so I won't do that to you. Chapter 7 and Chapter 12 have some drivers that work with "real" hardware, namely development boards from the makers of a PCI chip set and a USB chip set. Apart from that, however, all the drivers are for nonexistent hardware.

In nearly every case, I built a simple user-mode test program that you can use to explore the operation of the sample driver. These test programs are truly tiny: they contain just a few lines of code and are concerned with only whatever point the driver sample attempts to illustrate. Once again, I think it's better to give you a simple way to exercise the driver code that I assume you're really interested in instead of trying to show off every MFC programming trick I've ever learned.

You're free to use all the sample code in this book in your own projects without paying me or anyone else a royalty. (Of course, you must consult the detailed license agreement at the end of this book—this paraphrase is not intended to override that agreement in any way.) Please don't ship GENERIC.SYS to your customers, and please don't ship a driver that calls functions from GENERIC.SYS. The GENERIC.CHM help file in the companion content contains instructions on how to rename GENERIC to something less, well, generic. I intend readers to ship WDMSTUB.SYS and the AutoLaunch.exe modules, but I'll ask you to execute a royalty-free license agreement before doing so. Simply e-mail me at waltoney@oneysoft.com, and I'll tell you what to do. The license agreement basically obligates you to ship only the latest version of these components with an installation program that will prevent end users from ending up with stale copies.

About the Companion CD

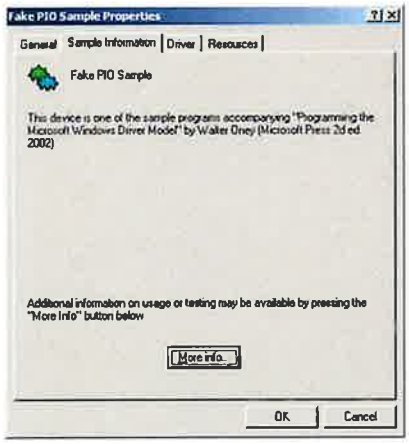
The CD that comes with this book contains the complete source code and an executable copy of each sample. To access those files, insert the companion CD in your computer's CD-ROM drive, and make a selection from the menu that appears. If the AutoRun feature isn't enabled on your system (if a menu doesn't appear when you insert the disc in your computer's CD-ROM drive), run StartCD.exe in the root folder of the companion CD. Installing the sample files on your hard disk requires approximately 50 MB of disk space.

The companion CD also contains a few utility programs that you might find useful in your own work. Open the file WDMBOOK.HTM in your Web browser for an index to the samples and an explanation of how to use these tools.

The setup program on the CD gives you the option to install all the samples on your own disk or to leave them on the CD. However, setup will not actually install any kernel-mode components on your system. Setup will ask your permission to add some environment variables to your system. The build procedure for the samples relies on these environment variables. They will be correctly set immediately on Windows XP and the next time you reboot Windows 98/Windows Me.

If your computer runs both Windows XP and Windows 98/Windows Me, I recommend performing a full install under both operating systems so that the registry and the environment are correctly set up in both places. Run the setup program from the installed sample directory the second time too, to avoid useless file copying. It isn't necessary or desirable to specify different target directories for the two installations.

Each sample includes an HTML file that explains (very briefly) what the sample does, how to build it, and how to test it. I recommend that you read the file before trying to install the sample because some of the samples have unusual installation requirements. Once you've installed a sample driver, you'll find that the Device Manager has an extra property page from which you can view the same HTML file, as shown here:



How the Samples Were Created

There's a good reason why my sample drivers look as though they all came out of a cookie cutter: they did. Faced with so many samples to write, I decided to write a custom application wizard. The wizard functionality in Microsoft Visual C++ version 6.0 is almost up to snuff for building a WDM driver project, so I elected to depend on it. The wizard is named WDMWIZ.AWX, and you'll find it in the companion content. I've documented how to use it in Appendix B. Use it, if you want, to construct the skeletons for your own drivers. But be aware that this wizard is not of product grade—it's intended to help you learn about writing drivers rather than to replace or compete with a commercial toolkit. Be aware too that you need to change a few project settings by hand because the wizard support is only almost what's needed. Refer to the WDMBOOK.HTM in the root directory of the companion CD for more information.

IPR2024-00292
KASPERSKY EX 1008 Page 14

Building the Samples

I vastly prefer using the Microsoft Visual Studio 6.0 integrated development environment for driver projects. If you share this preference, you can follow suit when you work with my samples. The WDMBOOK.HTM file in the companion content contains detailed instructions about how to set up the development environment. I'm deliberately not repeating those instructions here because they may change in the future. Each sample also includes a standard SOURCES file for use with the Driver Development Kit (DDK) build environments, in case your preference lies in that direction.

Updates to the Samples

At my Web site,http://www.oneysoft.com, you'll find a page concerning service packs for the sample drivers. In the three years since the first edition was printed, I issued about a dozen service packs. Service packs fix bugs and offer new samples. If you install my sample drivers, I recommend that you also install each new service pack as it comes out.

If you want to find out when a new service pack is available, you can fill out a simple online form to be added to my mailing list. First edition subscribers needn't reregister, by the way: you're all grandfathered in.

GENERIC.SYS

A WDM driver contains a great deal of code that you could call boilerplate for handling Plug and Play and power management. This code is long. It's boring. It's easy to get wrong. My samples all rely on what amounts to a kernel-mode DLL named GENERIC.SYS. WDMWIZ.AWX will build a project that uses GENERIC.SYS or that doesn't, as you specify. GENERIC.CHM in the companion content details the support functions that GENERIC.SYS exports, in case you want to use them yourself.

The downside to my using GENERIC all over the place is that I managed to obscure how some crucial things occur in the driver. The drivers that use GENERIC delegate all of the IRP_MJ_PNP (see Chapter 6) and IRP_MJ_POWER (see Chapter 8) handling to GENERIC, which then calls back to driver-specific routines to handle details. The following table describes the important callback functions.

IRP Type	Callback Function	Purpose
IRP_MJ_PNP	*StartDevice*	Start the device (map memory registers, connect interrupt, and so on).
	StopDevice	Halt device and release I/O resources (unmap memory registers, disconnect interrupt, and so on).
	RemoveDevice	Undo steps performed in *AddDevice* (disconnect from lower device object, delete device object, and so on).
	OkayToStop	(Optional) Is it OK to stop this device now (used while processing *IRP_MN_QUERY_STOP_DEVICE*)?
	OkayToRemove	(Optional) Is it OK to remove this device now (used while processing *IRP_MN_QUERY_REMOVE_DEVICE*)?
	FlushPendingIo	(Optional) Take any required action to force pending operations to finish in the near future.
IRP_MJ_POWER	*QueryPower*	(Optional) Is a proposed change in device power OK (used while processing *IRP_MN_QUERY_POWER*)?
	SaveDeviceContext	(Optional) Save any device context that will be lost during a period of low power.
	RestoreDeviceContext	(Optional) Restore device context after a period of low power.
	GetDevicePowerState	(Optional) Get device power state corresponding to a given system power state.

System Requirements

To run the sample programs in the companion content, you'll need a computer running Windows 98 Second Edition, Windows Me, Windows 2000, Windows XP, or any later version of Windows. Some of the samples require a USB port and an EZ-USB development kit from Cypress Semiconductors. Two of the samples require an ISA expansion slot and an S5933-DK development board (or equivalent) from Applied Micro Circuits Corporation.

To build the sample programs, you'll need a set of software tools that will change over time whenever I issue service packs. The file WDMBOOK.HTM describes the requirements and will be updated when requirements change. At the time this book is published, you'll need the following:

- The Microsoft Windows .NET DDK.
- Microsoft Visual Studio 6.0. Any edition will do, and it doesn't matter whether you've installed any of the service packs. When you're building the driver samples, you'll be using just the integrated development environment provided by Visual Studio. The compiler and other build tools will be coming from the DDK.
- For one of the samples only (PNPMON), the Windows 98 DDK.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 15

If you have to use Windows 98 or Windows Me as your only build and test environment, you'll also need to obtain a copy of the Windows DDK for a pre-.NET platform. Microsoft denied me permission to distribute a version of the resource compiler that would work on Windows 98/Windows Me or a cross-platform-compatible version of USBD.LIB. Grab these from wherever you can find them before Microsoft stops supporting earlier versions of the DDK. Bear in mind that drivers built on Windows 98/Windows Me might not run on Windows 2000 and later platforms due to an error in checksum computation in the image helper DLL.

Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion content. Microsoft Press provides corrections for books through the World Wide Web at the following address:

http://www.microsoft.com/mspress/support

To connect directly to the Microsoft Press Knowledge Base and enter a query regarding a question or an issue that you might have, go to:

http://www.microsoft.com/mspress/support/search.asp

If you have comments, questions, or ideas regarding this book or the companion content, or questions that aren't answered by querying the Knowledge Base, please send them to Microsoft Press by e-mail to:

mspinput@microsoft.com

Or by postal mail to:

Microsoft Press
Attn:
Programming Microsoft SQL Server 2000 with Microsoft Visual Basic .NET
Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that product support is not offered through the preceding mail address. For product support information, please visit the Microsoft Support Web site at:

http://support.microsoft.com

Note on Errors

Despite heroic attention to detail, I and the editors at Microsoft Press let a few errors slip by from my original manuscript to the finished first edition of this book. I overlooked a few technical things, slipped up on some others, and learned about still others after the book was in print. My personal favorite was the "Special Sauce" layer in Figure 3-1, which was a typically lame attempt to introduce humor into the editorial process that went awry when the original draft of the figure made it into the finished book. At any rate, my errata/update Web page has grown to about 30 printed pages, and my desire to start over at zero was one of the main reasons for this edition.

But, sigh, there will still be corrections and updates to be made to this edition too. I'll continue to publish updates and errata at http://www.oneysoft.com for at least the next couple of years. I recommend you go there first and often to stay up-to-date. And please send me your comments and questions so that I can correct as many errors as possible.

Other Resources

This book shouldn't be the only source of information you use to learn about driver programming. It emphasizes the features that I think are important, but you might need information I don't provide, or you might have a different way of learning than I do. I don't explain how the operating system works except insofar as it bears on what I think you need to know to effectively write drivers. If you're a deductive learner, or if you simply want more theoretical background, you might want to consult one of the additional resources listed next. If you're standing in a bookstore right now trying to decide which book to buy, my advice is to buy all of them: a wise craftsperson never skimps on his or her tools. Besides, you can never tell when a young dinner guest may need help reaching the table.

Books Specifically About Driver Development

Art Baker and Jerry Lozano, *The Windows 2000 Device Driver Book: A Guide for Programmers*, 2nd edition (Prentice Hall, 2001). Quite readable. Some errors survive from the first edition.

Edward N. Dekker and Joseph M. Newcomer, *Developing Windows NT Device Drivers: A Programmer's Handbook* (Addison-Wesley, 1999). A fine book with a fine sense of humor. Written just before WDM came out, so not much coverage of that.

Rajeev Nagar, *Windows NT File System Internals: A Developer's Guide* (O'Reilly & Associates, 1997). Nothing at all to do

IPR2024-00292
KASPERSKY EX 1008 Page 16

with WDM, but the only book that attempts to explain the internals of the Windows NT file system.

Peter G. Viscarola and W. Anthony Mason, *Windows NT Device Driver Development* (Macmillan, 1998). Technical and authoritative. A WDM edition is supposedly coming someday.

Other Useful Books

Michael Howard and David LeBlanc, *Writing Secure Code* (Microsoft Press, 2001). Exceptionally detailed and readable discussion of security issues in applications. I'll be reiterating many of *Writing Secure Code*'s lessons throughout this book.

Gary Nebbett, *Windows NT/2000 Native API Reference* (MacMillan, 2000). Detailed exposition of the underdocumented native API.

David A. Solomon and Mark E. Russinovich, *Inside Windows 2000*, Third Edition (Microsoft Press, 2000). All about the operating system. How come they got *their* pictures on the cover, inquiring minds would like to know?

Magazines

Old editions of *Microsoft Systems Journal* and *Windows Developer Journal* contain many articles about driver programming. Both of the magazines have gone to that Great Publishers Clearinghouse in the sky, however, and I can't speak for how well or often their successors cover driver issues.

Online Resources

The *comp.os.ms-windows.programmer.nt.kernel-mode* newsgroup provides a forum for technical discussion on kernel-mode programming issues. On the msnews.microsoft.com server, you can subscribe to microsoft.public.development.device.drivers. You can find mailing list servers for file system and driver programming issues by going to http://www.osr.com.

Roedy Green, "How to Write Unmaintainable Code" (2002), which I found at http://www.mindprod.com/unmain.html.

Seminars and Development Services

I conduct public and on-site seminars on WDM programming. Visit my Web site at http://www.oneysoft.com for more information and schedules. I also develop custom drivers for hardware manufacturers all over the world. I promise this is the only commercial in the book. (Not counting the back cover of the book, that is, which is full of statements aimed at getting you to buy the book and whose correspondence, if any, to reality will become susceptible to evaluation only if you succumb and actually read the book.)

About the Author

Walter Oney has 35 years of experience in systems-level programming and has been teaching Windows device driver classes for 10 years. He was a contributing editor to *Microsoft Systems Journal* during its heyday and is a Microsoft MVP. He has written several books, including *Systems Programming for Windows 95* and the first edition of *Programming the Microsoft Windows Driver Model*. In his free time, he's a committed jogger, a fan of classical dance, and an amateur oboist. He and his wife, Marty, live in Boston, Massachusetts.

IPR2024-00292
KASPERSKY EX 1008 Page 17

Chapter 1

Beginning a Driver Project

In this chapter, I'll present an overview of the driver writing process. My own personal involvement with personal computing dates from the mid-1980s, when IBM introduced its personal computer (PC) with MS-DOS as the operating system. Decisions made by IBM and Microsoft that long ago are still being felt today. Consequently, a bit of historical perspective will help you understand how to program device drivers.

Windows Driver Model (WDM) drivers run in two radically different operating system environments, and I'll provide an overview of the architecture of these environments in this chapter. Windows XP, like Windows 2000 and earlier versions of Windows NT, provides a formal framework in which drivers play well-defined roles in carrying out I/O operations on behalf of applications and other drivers. Windows Me, like Windows 9x and Windows 3.x before it, is a more freewheeling sort of system in which drivers play many roles.

The first step in any driver project is to decide what kind of driver you need to write—if indeed you need to write one at all. I'll describe many different classes of device in this chapter with a view toward helping you make this decision.

Finally I'll round out the chapter with a management checklist to help you understand the scope of the project.

1.1 A Brief History of Device Drivers

The earliest PCs ran on an Intel processor chip that provided addressability for 640 KB of "real" memory—so called because the memory was really there in the form of memory chips that the processor addressed directly by means of a 20-bit physical address. The processor itself offered just one mode of operation, the so-called *real mode*, wherein the processor combined information from two 16-bit registers to form a 20-bit memory address for every instruction that referenced memory. The computer architecture included the concept of expansion slots that brave users could populate with cards purchased separately from the computer itself. The cards themselves usually came with instructions about how to set DIP switches (later, jumpers between pins) in order to make slight changes in I/O configuration. You had to keep a map of all the I/O and interrupt assignments for your PC in order to do this correctly. MS-DOS incorporated a scheme based on the CONFIG.SYS file whereby the operating system could load real-mode device drivers for original equipment and for add-on cards. Inevitably, these drivers were programmed in assembly language and relied to a greater or lesser extent on the INT instruction to talk to the BIOS and to system services within MS-DOS itself. End users perforce learned how to invoke applications via commands. Application programmers perforce learned how to program the video display, keyboard, and mouse directly because neither MS-DOS nor the system BIOS did so adequately.

Later on, IBM introduced the AT class of personal computers based on the Intel 80286 processor. The 286 processor added a *protected mode* of operation wherein programs could address up to 16 MB of main and extended memory using a 24-bit segment address (specified indirectly via a segment *selector* in a 16-bit segment register) and a 16-bit offset. MS-DOS itself remained a real-mode operating system, so several software vendors built *DOS extender* products to allow programmers to migrate their real-mode applications to *protected mode* and gain access to all the memory that was becoming available on the market. Since MS-DOS was still in charge of the computer, driver technology didn't advance at this point.

The watershed change in PC technology occurred—in my view, anyway—when Intel released the 80386 processor chip. The 386 allowed programs to access up to 4 GB of virtual memory addressed indirectly via page tables, and it allowed programs to easily use 32-bit quantities for arithmetic and addressing. There was a flurry of activity in the software tools market as compiler vendors and *DOS extender* companies raced to capture the ever-growing volume of large applications hungry for memory and processor speed. Device drivers were *still* 16-bit real-mode programs written in assembly language and installed via CONFIG.SYS, and end users *still* needed to manually configure cards.

Subsequent advances in processor chips have been mainly in the area of performance and capacity. As I write this chapter, computers operating faster than 1 GHz with 50-GB hard drives and 512 MB (or more) of memory are commonplace and easily affordable by large segments of the population.

In parallel with the evolution of the platform, another evolution was occurring with operating system technology. Most people, even including programmers of system software, prefer graphics-based ways of interacting with computers to character-based ways. Microsoft was late to the graphical operating system party—Apple beat them with the first Macintosh—but has come to dominate it with the Windows family of operating systems. In the beginning, Windows was just a graphical shell for real-mode MS-DOS. Over time, a collection of Windows drivers for common hardware, including the display, keyboard, and mouse, came into existence. These drivers were executable files with a .DRV extension, and they were written primarily in assembly language.

With the advent of the AT class of computer, Microsoft added a protected-mode version of Windows. Microsoft ported the real-mode .DRV drivers to *protected mode* as well. Hardware other than the standard Windows devices (the display, keyboard, and mouse) continued to be handled by real-mode MS-DOS drivers.

IPR2024-00292
KASPERSKY EX 1008 Page 19

Finally, some time after PCs with 386 processors became widely available, Microsoft released Windows 3.0, whose "enhanced" mode of operation took full advantage of the virtual memory capabilities. Even so, it was still true that every new piece of hardware needed a real-mode driver. But now there was a big problem. To support multitasking of MS-DOS applications (a requirement for end user acceptance of Windows), Microsoft had built a virtual-machine operating system. Each MS-DOS application ran in its own virtual machine, as did the Windows graphical environment. But all those MS-DOS applications were trying to talk directly to hardware by issuing IN and OUT instructions, reading and writing device memory, and handling interrupts from the hardware. Furthermore, two or more applications sharing processor time could be issuing conflicting instructions to the hardware. They would certainly conflict over use of the display, keyboard, and mouse, of course.

To allow multiple applications to share physical hardware, Microsoft introduced the concept of a virtual device driver, whose broad purpose is to "virtualize" a hardware device. Such drivers were generically called *VxDs* because most of them had filenames fitting the pattern VxD.386, where x indicated the type of device they managed. Using this concept, Windows 3.0 created the appearance of virtual machines outfitted with separate instances of many hardware devices. But the devices themselves continued, in most cases, to be driven by real-mode MS-DOS drivers. A VxD's role was to mediate application access to hardware by first intercepting the application's attempts to touch the hardware and briefly switching the processor to a sort of *real mode* called *virtual 8086 mode* to run the MS-DOS driver.

Not to put too fine a face on it, mode switching to run real-mode drivers was a hack whose only virtue was that it allowed for a reasonably smooth growth in the hardware platform and operating system. Windows 3.0 had many bugs whose root cause was that very feature of the architecture. Microsoft's answer was to be OS/2, which it was developing in harmony (using a twentieth-century definition of harmony, that is) with IBM.

Microsoft's version of OS/2 became Windows NT, whose first public release was in the early 1990s, shortly after Windows 3.1. Microsoft built Windows NT from the ground up with the intention of making it a durable and secure platform on which to run Windows. Drivers for Windows NT used a brand-new kernel-mode technology that shared practically nothing with the other two driver technologies then in vogue. Windows NT drivers used the C programming language almost exclusively so that they could be recompiled for new CPU architectures without requiring any source changes.

Another thing happened along about the Windows 3.0 time frame that has an important ramification for us today. Windows 3.0 formally divided the software world into *user-mode* and *kernel-mode* programs. User-mode programs include all the applications and games that people buy computers to run, but they are not to be trusted to deal robustly (or even honestly) with hardware or with other programs. Kernel-mode programs include the operating system itself and all the device drivers that people like you and me write. Kernel-mode programs are fully trusted and can touch any system resource they please. Although Windows 3.0 segregated programs by their mode of operation, no version of Windows (not even Windows Me) has actually put memory protection in place to yield a secure system. Security is the province of Windows NT and its successors, which do forbid user-mode programs from seeing or changing the resources managed by the kernel.

Computing power didn't really advance to the point where an average PC could run Windows NT well until quite recently. Microsoft therefore had to keep the Windows product line alive. Windows 3.0 grew into 3.1, 3.11, and 95. Starting with Windows 95, if you wanted to write a device driver, you would write something called a VxD that was really just a 32-bit protected-mode driver. Also starting with Windows 95, end users could throw away their I/O maps because the new Plug and Play feature of the operating system identified and configured hardware somewhat automatically. As a hardware maker, though, you might have had to write a real-mode driver to keep happy those of your customers who weren't upgrading from Windows 3.1. Meanwhile, Windows NT grew into 3.5, 4.0. You would have needed a *third* driver to support these systems, and not much of your programming knowledge would have been portable between projects.

Enough was enough. Microsoft designed a new technology for device drivers, the *Windows Driver Model* (WDM), and put it into Windows 98 and Windows Me, the successors to Windows 95. They also put this technology into Windows 2000 and Windows XP, the successors to Windows NT 4.0. By the time of Windows Me, MS-DOS was present only by courtesy and there was finally no need for a hardware maker to worry about real-mode device drivers. Because WDM was, at least by original intention, practically the same on all platforms, it became possible to write just one driver.

To summarize, we stand today in the shadow of the original PC architecture and of the first versions of MS-DOS. End users still occasionally have to open the skin of their PCs to install expansion cards, but we use a different and more powerful bus nowadays than we did originally. Plug and Play and the Peripheral Component Interconnect (PCI) bus have largely removed the need for end users to keep track of I/O, memory, and interrupt request usage. There is still a BIOS in place, but its job nowadays is mostly to boot the system and to inform the real operating system (Windows XP or Windows Me) about configuration details discovered along the way. And WDM drivers still have the file extension .SYS, just as the first real-mode drivers did.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 20

1.2 An Overview of the Operating Systems

The Windows Driver Model provides a framework for device drivers that operate in two operating systems—Windows 98/Windows Me and Windows 2000/Windows XP. As discussed in the preceding historical summary, these two pairs of operating systems are the products of two lines of parallel evolution. In fact, I'll refer to the former pair of systems with the abbreviation "98/Me" to emphasize their common heritage and to the latter pair simply as XP. Although to the end user these two pairs of systems are similar, they work quite differently on the inside. In this section, I'll present a brief overview of the two systems.

1.2.1 Windows XP Overview

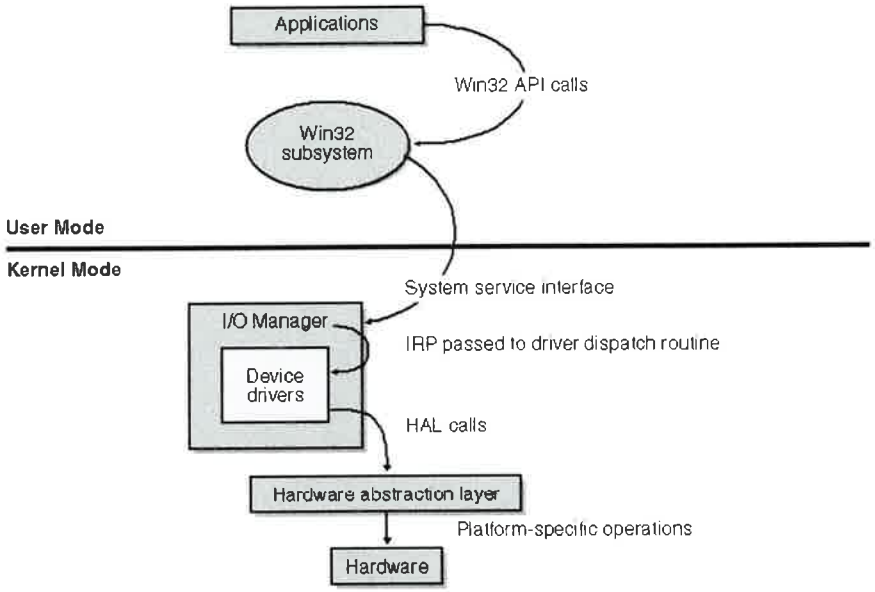


Figure 1-1. *Windows XP architecture.*

Figure 1-1 is a highly abbreviated functional diagram of the Windows XP operating system, wherein I emphasize the features that are important to people who write device drivers. Every platform where Windows XP runs supports two modes of execution. Software executes either in user mode or in kernel mode. A user-mode program that wants to, say, read some data from a device would call an application programming interface (API) such as *ReadFile*. A subsystem module such as KERNEL32.DLL implements this API by invoking a *native API* function such as Nt*ReadFile*. Refer to the sidebar for more information about the native API.

We often say that *NtReadFile* is part of a system component called the I/O Manager. The term I/O Manager is perhaps a little misleading because there isn't any single executable module with that name. We need a name to use when discussing the "cloud" of operating system services that surrounds our own driver, though, and this name is the one we usually pick.

Many routines serve a purpose similar to Nt*ReadFile*. They operate in kernel mode in order to service an application's request to interact with a device in some way. They all validate their parameters, thereby ensuring that they don't inadvertently allow a security breach by performing an operation, or accessing some data, that the user-mode program wouldn't have been able to perform or access by itself. They then create a data structure called an I/O request packet (IRP) that they pass to an entry point in some device driver. In the case of an original *ReadFile* call, Nt*ReadFile* would create an IRP with the major function code IRP_MJ_READ (a constant in a DDK [Driver Development Kit] header file). Processing details at this point can differ, but a likely scenario is for a routine such as Nt*ReadFile* to return to the user-mode caller with an indication that the operation described by the IRP hasn't finished yet. The user-mode program might continue about its business and then wait for the operation to finish, or it might wait immediately. Either way, the device driver proceeds independently of the application to service the request.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 21

The Native API
NtReadFile is part of the so-called native API of Windows XP. The reason there is a native API is historical. The original Windows NT operating system contained a number of subsystems to implement the semantics of several new and existing operating systems. There was an OS/2 subsystem, a POSIX subsystem, and a Win32 subsystem. The subsystems were implemented by making user-mode calls to the native API, which was itself implemented in kernel mode.

A user-mode DLL named (rather redundantly, I've always thought) NTDLL.DLL implements the native API for Win32 callers. Each entry in this DLL is a thin wrapper around a call to a kernel-mode function that actually carries out the function. The call uses a platform-dependent system service interface to transfer control across the user-mode/kernel-mode boundary. On newer Intel processors, this system service interface uses the SYSENTER instruction. On older Intel processors, the interface uses the INT instruction with the function code 0x2E. On other processors, still other mechanisms are employed. You and I don't need to understand the details of the mechanism to write drivers, though. All we need to understand is that the mechanism allows a program running in user mode to call a subroutine that executes in kernel mode and that will eventually return to its user-mode caller. No thread context switching occurs during the process: *all* that changes is the privilege level of the executing code (along with a few other details that only assembly language programmers would ever notice or care about).

The Win32 subsystem is the one most application programmers are familiar with because it implements the functions one commonly associates with the Windows graphical user interface. The other subsystems have fallen by the wayside over time. The native API remains, however, and the Win32 subsystem still relies on it in the way I'm describing by example in the text.

A device driver may eventually need to actually access its hardware to perform an IRP. In the case of an IRP_MJ_READ to a programmed I/O (PIO) sort of device, the access might take the form of a read operation directed to an I/O port or a memory register implemented by the device. Drivers, even though they execute in kernel mode and can therefore talk directly to their hardware, use facilities provided by the hardware abstraction layer (HAL) to access hardware. A read operation might involve calling READ_PORT_UCHAR to read a single data byte from an I/O port. The HAL routine uses a platform-dependent method to actually perform the operation. On an x86 computer, the HAL would use the IN instruction; on some other future Windows XP platform, it might perform a memory fetch.

After a driver has finished with an I/O operation, it completes the IRP by calling a particular kernel-mode service routine. Completion is the last act in processing an IRP, and it allows the waiting application to resume execution.

1.2.2 Windows 98/Windows Me Overview

Figure 1-2 shows one way of thinking about Windows 98/Windows Me. The operating system kernel is called the Virtual Machine Manager (VMM) because its main job is to create one or more virtual machines that share the hardware of a single physical machine. The original purpose of a virtual device driver in Windows 3.0 was to virtualize a specific device in order to help the VMM create the fiction that each virtual machine has a full complement of hardware. The same VMM architecture introduced with Windows 3.0 is in place today in Windows 98/Me, but with a bunch of accretions to handle new hardware and 32-bit applications.

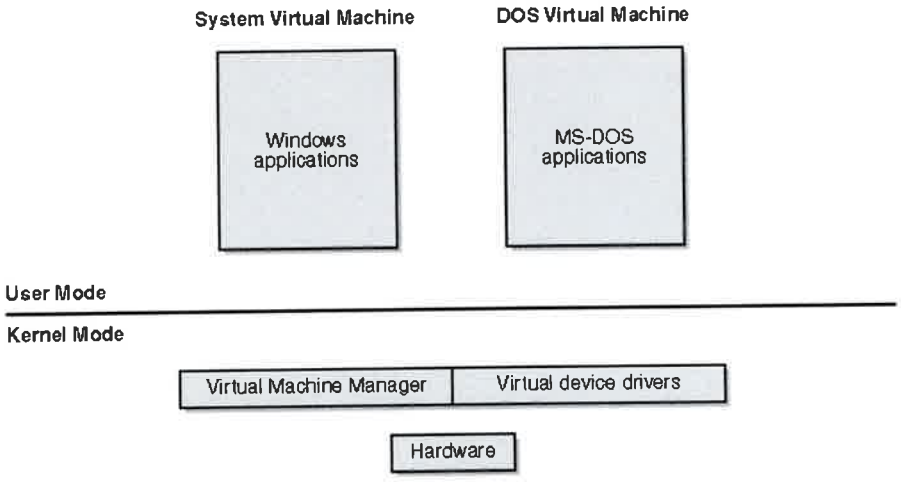


Figure 1-2. *Windows 98/Me architecture.*

Windows 98/Me doesn't handle I/O operations in quite as orderly a way as Windows XP. There are major differences in the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 22

way Windows 98/Me handles operations directed to disks, to communication ports, to keyboards, and so on. There are also fundamental differences between how Windows services 32-bit and 16-bit applications. See Figure 1-3.

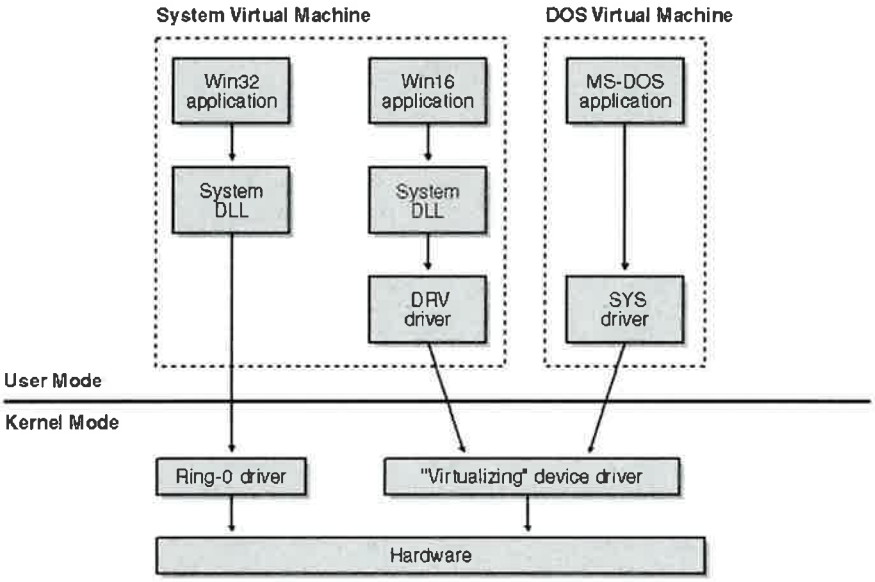


Figure 1-3. *I/O requests in Windows 98/Me*

The left column of Figure 1-3 shows how 32-bit applications get I/O done for them. An application calls a Win32 API such as *ReadFile*, which a system DLL such as KERNEL32.DLL services. But applications can use *ReadFile* only for reading disk files, communication ports, and devices that have WDM drivers. For any other kind of device, an application must use some ad hoc mechanism based on *DeviceIoControl*. The system DLL contains different code than its Windows XP counterpart too. The user-mode implementation of *ReadFile*, for example, validates parameters—a step done in kernel mode on Windows XP—and uses one or another special mechanism to reach a kernel-mode driver. There's one special mechanism for disk files, another for serial ports, another for WDM devices, and so on. The mechanisms all use software interrupt 30h to transition from user mode to kernel mode, but they're otherwise completely different.

The middle column of Figure 1-3 shows how 16-bit Windows-based applications (Win16 applications) perform I/O. The right column illustrates the control flow for MS-DOS-based applications. In both cases, the user-mode program calls directly or indirectly on the services of a user-mode driver that, in principle, could stand alone by itself on a bare machine. Win16 programs perform serial port I/O by indirectly calling a 16-bit DLL named COMM.DRV, for example. (Up until Windows 95, COMM.DRV was a stand-alone driver that hooked IRQ 3 and 4 and issued IN and OUT instructions to talk directly to the serial chip.) A virtual communications device driver (VCD) intercepts the port I/O operations in order to guard against having two different virtual machines access the same port simultaneously. In a weird way of thinking about the process, these user-mode drivers use an "API" interface based on interception of I/O operations. "Virtualizing" drivers like VCD service these pseudo-API calls by simulating the operation of hardware.

Whereas all kernel-mode I/O operations in Windows XP use a common data structure (the IRP), no such uniformity exists in Windows 98/Me, even after an application's request reaches kernel mode. Drivers of serial ports conform to a port driver function-calling paradigm orchestrated by VCOMM.VXD. Disk drivers, on the other hand, participate in a packet-driven layered architecture implemented by IOS.VXD. Other device classes use still other means.

When it comes to WDM drivers, though, the interior architecture of Windows 98/Me is necessarily very similar to that of Windows XP. A system module (NTKERN.VXD) contains Windows-specific implementations of a great many Windows NT kernel support functions. NTKERN creates IRPs and sends them to WDM drivers in just about the same way as Windows XP. WDM drivers can almost not tell the difference between the two environments, in fact.

Despite the similarities in the WDM environments of Windows XP and Windows 98/Me, however, there are some significant differences. You'll find compatibility notes throughout this book that highlight the differences that matter to most driver programmers. Appendix A outlines a scheme whereby you can use the same binary driver on Windows 2000/XP and Windows 98/Me despite these differences.

IPR2024-00292
KASPERSKY EX 1008 Page 23

1.3 What Kind of Driver Do I Need?

Many kinds of drivers form a complete Windows XP system. Figure 1-4 diagrams several of them.

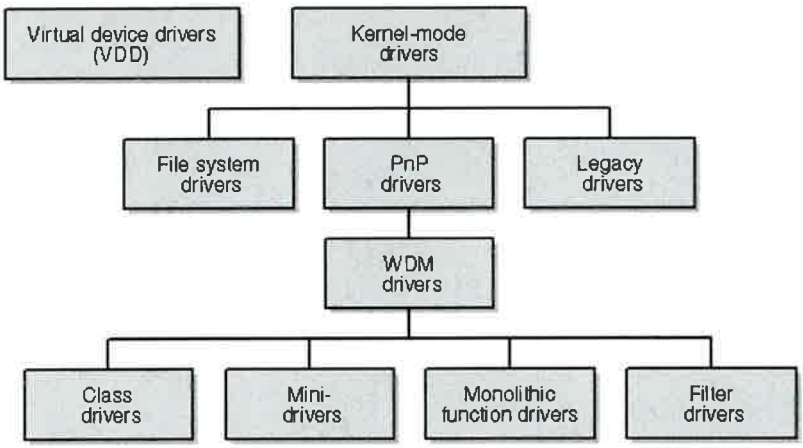


Figure 1-4. *Types of device drivers in Windows XP.*

- A *virtual device driver* (VDD) is a user-mode component that allows MS-DOS-based applications to access hardware on Intel x86 platforms. A VDD relies on the I/O permission mask to trap port access, and it essentially simulates the operation of hardware for the benefit of applications that were originally programmed to talk directly to hardware on a bare machine. Don't confuse a Windows XP VDD with a Windows 98/Me VxD. Both are called virtual device drivers, and they serve the same basic purpose of virtualizing hardware, but they employ completely different software technology.
- The category *kernel-mode drivers* includes many subcategories. A *PnP driver* is a kernel-mode driver that understands the Plug and Play protocols of Windows XP. To be perfectly accurate, this book concerns PnP drivers and nothing else.
- A *WDM driver* is a PnP driver that additionally understands power-management protocols and is source compatible with both Windows 98/Me and Windows 2000/XP. Within the category of WDM driver, you can also distinguish between class drivers (which manage a device belonging to some well-defined class of device) and *minidrivers* (which supply vendor-specific help to a class driver), and between *monolithic function drivers* (which embody all the functionality needed to support a hardware device) and *filter drivers* (which "filter" the I/O operations for a particular device in order to add or modify behavior).
- *File system drivers* implement the standard PC file system model (which includes the concepts of a hierarchical directory structure containing named files) on local hard disks or over network connections. These, too, are kernel-mode drivers.
- *Legacy device drivers* are kernel-mode drivers that directly control a hardware device without help from other drivers. This category essentially includes drivers for earlier versions of Windows NT that are running without change in Windows XP.

Not all the distinctions implied by this classification scheme are important all of the time. As I remarked in my previous book *Systems Programming for Windows 95* (Microsoft Press, 1996), you haven't stumbled into a nest of pedants by buying my book. In particular, I'm not always going to carefully distinguish between WDM and PnP drivers in the rigorous way implied by the preceding taxonomy. The distinction is a phenomenological one based on whether a given driver runs both in Windows 2000/XP and Windows 98/Me. Without necessarily using the technically exact term, I'll be very careful to discuss system dependencies when they come up hereafter.

Faced with all these categories of driver, a new driver writer or manager would understandably be confused about what sort of driver he or she needs for a given piece of hardware. For some devices, you don't need to write any driver at all because Microsoft already ships a generic driver that will work with your device. Here are some examples:

- SCSI-compatible or ATAPI-compatible mass storage device
- Any device connected to a USB port that is fully compatible with an approved specification, provided you're happy with any limitations in the standard Microsoft driver
- Standard serial or PS/2 mouse
- Standard keyboard

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 24

- Video adapter without acceleration or other special features
- Standard parallel or serial port
- Standard floppy disk drive

1.3.1 WDM Drivers

For most devices that Microsoft doesn't directly support, you need to write a WDM driver. You will decide first whether to write a monolithic function driver, a filter driver, or just a minidriver. You'll probably never need to write a class driver because Microsoft would like to reserve that specialty to itself in order to serve the broadest range of hardware makers.

WDM Minidrivers

The basic rule of thumb is that if Microsoft has written a class driver for the type of device you're trying to support, you should write a minidriver to work with that class driver. Your minidriver is nominally in charge of the device, but you'll call subroutines in the class driver that basically take over the management of the hardware and call back to you to do various device-dependent things. The amount of work you need to do in a minidriver varies tremendously from one class of device to another.

Here are some examples of device classes for which you should plan to write a minidriver:

- Non-USB human input devices (HID), including mice, keyboards, joysticks, steering wheels, and so on. If you have a USB device for which the generic behavior of HIDUSB.SYS (the Microsoft driver for USB HID devices) is insufficient, you would write a HIDCLASS minidriver too. The main characteristic of these devices is that they report user input by means of *reports* that can be described by a descriptor data structure. For such devices, HIDCLASS.SYS serves as the class driver and performs many functions that Direct-Input and other higher layers of software depend on, so you're pretty much stuck with using HIDCLASS.SYS. This is hard enough that I've devoted considerable space to it later in this book. As an aside, HIDUSB.SYS is itself a HIDCLASS minidriver.
- Windows Image Acquisition (WIA) devices, including scanners and cameras. You will write a WIA minidriver that essentially implements some COM-style interfaces to support vendor-specific aspects of your hardware.
- Streaming devices, such as audio, DVD, and video devices, and software-only filters for multimedia data streams. You will write a stream minidriver.
- Network interface devices on nontraditional buses, such as USB or 1394. For such a device, you will write a Network Driver Interface Specification (NDIS) miniport driver "with a WDM lower edge," to use the same phrase as the DDK documentation on this subject. Such a driver is unlikely to be portable between operating systems, so you should plan on writing several of them with minor differences to cope with platform dependencies.
- Video cards. These devices require a video minidriver that works with the video port class driver.
- Printers, which require user-mode DLLs instead of kernel-mode drivers.
- Batteries, for which Microsoft supplies a generic class driver. You would write a minidriver (which the DDK calls a miniclass driver, but it's the same thing) to work with BATTC.SYS.

WDM Filter Drivers

You may have a device that operates so closely to a recognized standard that a generic Microsoft driver is almost adequate. In some situations, you may be able to write a filter driver that modifies the behavior of the generic driver just enough to make your hardware work. This doesn't happen very frequently, by the way, because it's often not easy to change the way a generic driver accesses the hardware. I'll discuss filter drivers in great detail in Chapter 16.

Monolithic WDM Function Drivers

With some exceptions to be noted in the next section, most other types of device require what I've called here a monolithic WDM function driver. Such a driver essentially stands alone and handles all the details of controlling your hardware.

When this style of driver is appropriate, I recommend the following approach so that you can end up with a single binary that will work on Intel x86 platforms in all operating systems. First, build with the most recent DDK—I used a beta version of the .NET DDK for the samples in the companion content. You can use *IoIsWdmVersionAvailable* to decide which operating system you happen to be using. If you happen to be running in Windows 2000 or Windows XP, you can call *MmGetSystemRoutineAddress* to get a pointer to a Windows XP-only function. I also suggest shipping WDMSTUB.SYS, which is discussed in Appendix A, to define *MmGetSystemRoutineAddress* and other critical kernel functions in Windows 98/Me; otherwise, your driver simply won't load in Windows 98/Me because of undefined imports.

IPR2024-00292
KASPERSKY EX 1008 Page 25

Here are some examples of devices for which you might write a monolithic WDM function driver:

- Any kind of SmartCard reader except one attached to a serial port
- Digital-to-analog converter
- ISA card supporting proprietary identification tag read/write transducer

More About Binary Compatibility
Originally, WDM was to have been binary portable across all versions of Windows. Because of release schedules and second (and higher-order) thoughts, every release since Windows 98 has included support for more and more kernel functions that are useful, and sometimes even essential, for robust and convenient programming. An example is the *IoXxxWorkItem* family of functions, discussed in Chapter 14, which was added to Windows 2000 and which must be used instead of the similar but less robust *ExXxxWorkItem* family. Unless you do something extra, a driver that calls *IoXxxWorkItem* functions simply won't load in Windows 98/Me because the operating system doesn't export the functions. *MmGetSystemRoutineAddress* is another function that didn't make it into Windows 98/Me, unfortunately, so you can't even make a run-time decision regarding which work item functions to call. As if this weren't enough, the WHQL tests for all drivers flag calls to the *ExXxxWorkItem* functions.

In Windows 98/Me, a VxD named NTKERN implements the WDM subset of kernel support functions. As discussed in more detail in Appendix A, NTKERN relies on defining new export symbols for use by the run-time loader. You can also define your own export symbols, which is how WDMSTUB manages to define missing symbols for use by the kind of binary-portable driver I'm advocating you build.

The companion content for this book includes the WDMCHECK utility, which you can run on a Windows 98/Me system to check a driver for missing imports. If you've developed a driver that works perfectly in Windows XP, I suggest copying the driver to a Windows 98/Me system and running WDMCHECK first thing. If WDMCHECK shows that your driver calls some unsupported functions, the next thing to check is whether WDMSTUB supports those functions. If so, just add WDMSTUB to your driver package as shown in Appendix A. If not, either modify your driver or send me an e-mail asking me to modify WDMSTUB. Either way, you'll eventually end up with a binary-compatible driver.

1.3.2 Other Types of Drivers

A few situations exist in which a monolithic WDM function driver won't suffice because of architectural differences between Windows 98/Me and Windows 2000/XP. In the following cases, you would need to write two drivers: a WDM driver for Windows 2000/XP and a VxD driver for Windows 98/Me:

- A driver for a serial port. The Windows 98/Me driver is a VxD that offers the VCOMM port driver interface at its upper edge, whereas the Windows 2000/XP driver is a WDM driver that offers a rich and rigidly specified IOCTL interface at its upper edge. The two upper-edge specifications have nothing in common.
- A driver for a device connected to a serial port. The Windows 98/Me driver is a VxD that calls VCOMM in order to talk to the port. The Windows 2000/XP driver is a WDM driver that talks to SERIAL.SYS or some other serial port driver that implements the same IOCTL interface.
- A driver for a nonstandard USB mass storage device. For Windows 98/Me, you'll write a VxD that fits into the I/O Supervisor hierarchy of layered drivers. For Windows 2000/XP, you'll write a monolithic WDM function driver that accepts SCSI Request Blocks at its upper edge and communicates with the USB device at its lower edge.
- For two classes of device, Microsoft defined a portable driver architecture long before WDM:
- Small Computer System Interface (SCSI) adapters use a "SCSI miniport" driver, which doesn't use any of the standard kernel support functions and relies instead on a special API exported by SCSIPORT.SYS or SCSIPORT.VXD, as the case may be. The miniport is portable between systems.
- Network interface cards use an "NDIS miniport driver," which relies exclusively on a special API exported by NDIS.SYS or NDIS.VXD, as the case may be. At one time, NDIS miniport drivers were portable between systems, but portability has pretty much been lost by now. Network protocol drivers and so-called "intermediate" drivers that provide filtering functionality also orbit around NDIS.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 26

1.3.3 Management Overview and Checklist

If you're a development manager, or if you're otherwise responsible for delivering a hardware device to market, there are a few things you need to know about device drivers. You need to decide, first of all, whether you need a customized driver and, if so, what kind. The preceding section should help you with that decision, but you might want to hire an expert consultant for the limited purpose of advising you on that score.

If your evaluation leads you to believe you need a custom driver, you then need to locate an appropriate programmer. The sad truth is that WDM driver programming is pretty hard, and only experienced (and expensive!) programmers are capable of doing it well. Some companies have cadres of driver programmers, but most can't afford to. If you're in the latter situation, your basic choice is between training someone who's already on your staff, hiring a programmer who already has the necessary skills, engaging a consultant or contract programmer, or outsourcing the development to a company that specializes in driver programming. All of these alternatives have pluses and minuses, and you will have to weigh them based on your own unique needs.

Driver programming should start as soon as there is a reasonably firm specification for how the hardware will work. You should expect to modify the specification in light of unpleasant discoveries during driver development, and you should also expect to iterate your hardware/firmware and driver design several times. Flexibility and a willingness to start over will really help you.

You should also expect driver programming to last longer and cost more than you initially imagine. All software is subject to time and cost overruns. Additional overruns in this kind of programming stem from communication difficulties between the hardware and software people, from ambiguity in specifications and in the DDK documentation, from bugs in all the components, and from delays in engineering and production.

In most cases, you'll want to submit your hardware and software to the Windows Hardware Quality Lab (WHQL) in order to obtain a digital certificate that will streamline installation and provide an entrée to one of Microsoft's logo programs. You'll do most of the testing yourself, and you'll need specific computer setups to do it, so find out early what the testing requirements are for your class of device to avoid being caught short at the end of your project. (Just as an example, testing a USB device requires you to have a variety of *audio* hardware in a specific topology, even if your device has nothing to do with audio or any other kind of streaming media.)

Also prepare your business infrastructure for working with WHQL. At a minimum, this will require obtaining a Data Universal Numbering System (DUNS) number from Dun and Bradstreet (or providing equivalent proof of business organization) and a digital signature certificate from Verisign. As of this writing, the DUNS number was free, but the Verisign certificate was not. And working through all the processes of multiple companies will take time.

Pay attention early on to how end users will install the driver software. Most vendors of add-on hardware prefer to ship a custom installation program on a CD-ROM, and writing the installer is a lengthy process that can consume an experienced programmer for several weeks. Web-based driver repositories are quite common and require special attention to installation issues.

Drivers can provide statistical and other management information in two ways. The Windows Management Instrumentation (WMI) subsystem provides a language- and transport-independent pathway for various sorts of binary data. Microsoft has established standard WMI classes for certain types of device, and your own industry subgroup may have established other standards to which your driver should conform. Chapter 10 contains information on how to conform to the Microsoft standards, but finding out how to fit in with the rest of the industry may be a job for your company's trade group representatives.

The second way of providing management information is by means of the system event log, which has been part of Windows NT since the beginning and which gives administrators a quick way of learning about exceptional conditions that have arisen in the recent past. Your driver should report events that an administrator would be interested in and can do something about. Whoever programs your driver should consult with an experienced system administrator to decide which events to log, so as to avoid cluttering the log with routine and unexceptional information. Your driver executable file will also probably include the text of messages in a special multilingual message resource, and it would be a good idea to have a trained writer compose that text. (I'm not saying your driver programmer *can't do* this, but he or she may not be the best choice.)

In addition to a driver, you may need control panel or other configuration software. The driver programmer and a specialist in user interaction should work together to build these components. Since they'll be installed along with the driver, they'll be part of the package that WHQL digitally signs, so they need to be finished at the same time as the driver.

Finally, don't treat your drivers as unimportant details. Having a good driver with a smooth installation is at least as important as the exterior appearance of the product. To put it simply, if your driver crashes the operating system, reviewers will alert the public, and anyone who doesn't read the reviews will be irately returning your product to the stores. You won't have any repeat business from people whose systems have crashed, even once, because of your driver. So a myopic decision to short-fund driver development could easily have a dramatic, negative effect on your bottom line for years to come. This advice is especially important for hardware manufacturers in developing countries, where managers have a tendency to look for every possible way to cut costs. I suggest that driver development is one place where cost-based decision making is inappropriate.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 27

To summarize, plan your project with the following milestones in mind:

- Evaluation of required driver and selection of programming talent
- Programming specification for hardware complete enough for driver work to begin
- Prototype hardware available for driver testing
- Driver and hardware/firmware working together as originally intended
- Installation (INF) file tested on all operating systems
- Control panels and other ancillary software done
- WMI and event log functionality done and tested
- WHQL self-tests passed and submission made
- Custom installation program done (not part of WHQL submission)
- Ready to burn discs and ship product!

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 28

Chapter 2

Basic Structure of a WDM Driver

In the first chapter, I described the basic architecture of the Microsoft Windows XP and Microsoft Windows 98/Me operating systems. I explained that the purpose of a device driver is to manage a piece of hardware on behalf of the system, and I discussed how to decide what kind of driver your hardware will need. In this chapter, I'll describe more specifically what program code goes into a WDM driver and how different kinds of drivers work together to manage hardware. I'll also present an overview of how the system finds and loads drivers.

2.1 How Drivers Work

A useful way to think of a complete driver is as a container for a collection of subroutines that the operating system calls to perform various operations that relate to your hardware. Figure 2-1 illustrates this concept. Some routines, such as the *DriverEntry* and *AddDevice* routines, as well as dispatch functions for a few types of I/O Request Packet (IRP), will be present in every such container. Drivers that need to queue requests might have a *StartIo* routine. Drivers that perform direct memory access (DMA) transfers will have an *AdapterControl* routine. Drivers for devices that generate hardware interrupts will have an interrupt service routine (ISR) and a deferred procedure call (DPC) routine. Most drivers will have dispatch functions for several types of IRP besides the three that are shown in Figure 2-1. One of your jobs as the author of a WDM driver, therefore, is to select the functions that need to be included in your particular container.

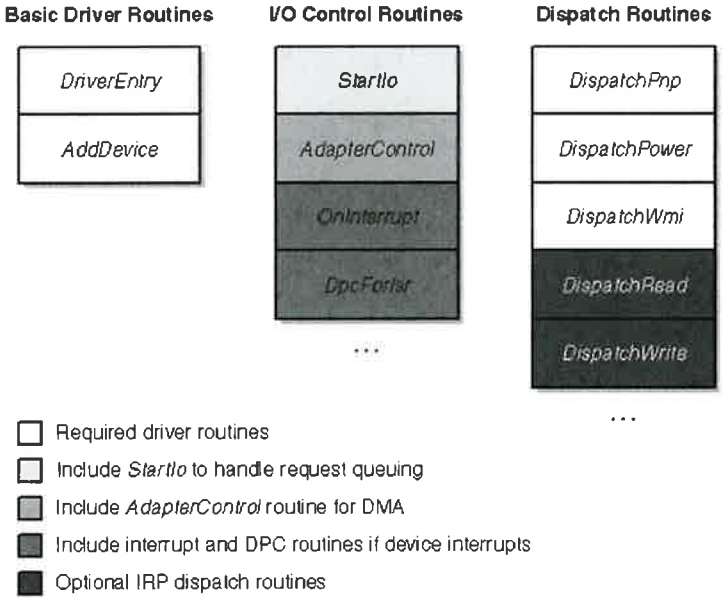


Figure 2-1. A driver considered as a package of subroutines.

I'll show you in this chapter how to write the *DriverEntry* and *AddDevice* routines for a monolithic function driver, one of the types of WDM driver this book discusses. As you'll learn in later chapters, filter drivers also have *DriverEntry* and *AddDevice* routines that are similar to what you'll see here. As you'll also learn, *minidrivers* have very different *DriverEntry* routines and may or may not have *AddDevice* routines, all depending on how the author of the associated class driver designed the class driver interface.

2.1.1 How Applications Work

It's worth a moment to reflect on the implications of the "package of subroutines" model for a driver by contrasting it with the "main program and helpers" model that applies to an application. Consider this program, which is among the first that many of us learn to write:

IPR2024-00292
KASPERSKY EX 1008 Page 29

```
int main(int argc, char* argv[])
  {
  printf("Hello, world!\n");
  return 0;
  }
```

This program consists of a main program named *main* and a library of helper routines, most of which we don't explicitly call. One of the helper routines, *printf*, prints a message to the standard output file. After compiling the source module containing the main program and linking it with a runtime library containing *printf* and the other helper routines needed by the main program, you would end up with an executable module that you might name HELLO.EXE. I'll go so far as to call this module by the grandiose name *application* because it's identical in principle to every other application now in existence or hereafter written. You could invoke this application from a command prompt this way:

```
C:\>hello
Hello, world!

C:\>
```

Here are some other common facts about applications:

- Some of the helper routines an application uses come from a static library, from which the linkage editor extracts them as part of the build process. *printf* is one of these functions.

  Other helper routines are dynamically linked from system dynamic-link libraries (DLLs). For these routines, the linkage editor places special *import* references in the executable file, and the runtime loader fixes up these references to point to the actual system code. As a matter of fact, the entire Win32 API used by application programs is dynamically linked, so you can see that dynamic linking is a very important concept in Windows programming.

- Executable files can contain symbolic information that allows debuggers to associate runtime addresses with the original source code.

- Executable files can also contain resource data, such as dialog box templates, text strings, and version identification. Placing this sort of data within the file is better than using separate auxiliary files because it avoids the problem of having mismatched files.

The interesting thing about HELLO.EXE is that once the operating system gives it control, it doesn't return until it's completely done with the task it performs. That's a characteristic of every application you'll ever use in Windows, actually. In a console-mode application such as HELLO, the operating system initially transfers control to an initialization function that's part of the compiler's runtime library. The initialization function eventually calls *main* to do the application's work.

Graphical applications in Windows work in much the same way except that the main program is named *WinMain* instead of main. *WinMain* operates a *message pump* to receive and dispatch messages to window procedures. It returns to the operating system when the user closes the main window. If the only Windows applications you ever build use Microsoft Foundation Classes (MFC), the *WinMain* procedure is buried in the library where you might never spot it, but rest assured it's there.

More than one application can appear to be running simultaneously on a computer, even a computer that has just one central processing unit. The operating system kernel contains a scheduler that gives short blocks of time, called *time slices*, to all the threads that are currently eligible to run. An application begins life with a single thread and can create more if it wants. Each thread has a priority, given to it by the system and subject to adjustment up and down for various reasons. At each decision point, the scheduler picks the highest-priority eligible thread and gives it control by loading a set of saved register images, including an instruction pointer, into the processor registers. A processor interrupt accompanies expiration of the thread's time slice. As part of handling the interrupt, the system saves the current register images, which can be restored the next time the system decides to redispatch the same thread.

Instead of just waiting for its time slice to expire, a thread can block each time it initiates a time-consuming activity in another thread until the activity finishes. This is better than spinning in a polling loop waiting for completion because it allows other threads to run sooner than they would if the system had to rely solely on expiration of a time slice to turn its attention to some other thread.

Now, I know you already knew what I just said. I just wanted to focus attention on the fact that an application is, at bottom, a selfish thread that grabs the CPU and tries to hold on until it exits and that the operating system scheduler acts like a playground monitor to make a bunch of selfish threads play well together.

2.1.2 Device Drivers

Like HELLO.EXE, a driver is also an executable file. It has the file extension .SYS, but structurally the disk file looks exactly like any 32-bit Windows or console-mode application. Also like HELLO.EXE, a driver uses a number of helper routines, many of which are dynamically linked from the operating system kernel or from a class driver or other supporting library. A driver file can have symbolic debugging information and resource data too.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 30

The System Is in Charge

Unlike HELLO.EXE, however, a driver doesn't contain a main program. Instead, it contains a collection of subroutines that the system can call when the system thinks it's time to. To be sure, these subroutines can use helper subroutines in the driver, in static libraries, and in the operating system, but the driver isn't in charge of anything except its own hardware: the system is in charge of everything else, including the decisions about when to run your driver code.

Here's a brief snapshot of how the operating system might call subroutines in your driver:

1. The user plugs in your device, so the system loads your driver executable into virtual memory and calls your *DriverEntry* routine. *DriverEntry* does a few things and returns.
2. The Plug and Play Manager (PnP Manager) calls your *AddDevice* routine, which does a few things and returns.
3. The PnP Manager sends you a few IRPs. Your dispatch function processes each IRP in turn and returns.
4. An application opens a handle to your device, whereupon the system sends you another IRP. Your dispatch routine does a little work and returns.
5. The application tries to read some data, whereupon the system sends you an IRP. Your dispatch routine puts the IRP in a queue and returns.
6. A previous I/O operation finishes by signaling a hardware interrupt to which your driver is connected. Your interrupt routine does a little bit of work, schedules a DPC, and returns.
7. Your DPC routine runs. Among other things, it removes the IRP you queued at step 5 and programs your hardware to read the data. Then the DPC routine returns to the system.
8. Time passes, during which the system makes many other brief calls into your subroutines.
9. Eventually, the end user unplugs your device. The PnP Manager sends you some IRPs, which you process and return. The operating system calls your *DriverUnload* routine, which usually just does a tiny amount of work and returns. Then the system removes your driver code from virtual memory.

At each step of this process, the system decided that your driver needed to do something, be it initializing, processing an IRP, handling an interrupt, or whatever. So the system selected the appropriate subroutine within your driver. Your routine did what it was supposed to do and returned to the system.

Threads and Driver Code

Another way in which drivers are dissimilar to applications is that the system doesn't create a special thread in which to run the driver code. Instead, a driver subroutine executes in the context of whatever thread happens to be currently active at the time the system decides to call that subroutine.

It's not possible to predict which thread will be current at the time a hardware interrupt occurs. As an analogy, imagine that you're watching a carousel at an amusement park. The horses on the carousel are like threads in a running system. Call the horse that's nearest to you the "current" horse. Now suppose you decide to take a picture with your camera the next time you overhear someone say the phrase, "That's awesome, dude." (In my experience at amusement parks, this does not entail a long wait.) You wouldn't expect to be able to predict which horse would be "current" in your snapshot. Which of all the eligible threads that happens to be executing at the time of hardware interrupt is likewise not predictable. We call this an *arbitrary thread*, and we speak of running in an arbitrary thread context.

The system is often running in an arbitrary thread context when it decides to call a subroutine in your driver. The thread context would be arbitrary—for example, when your interrupt service routine gets control. If you schedule a DPC, the thread in which your DPC routine runs will be arbitrary. If you queue IRPs, your *StartIo* routine will be called in an arbitrary thread. In fact, if some driver outside your own stack sends you an IRP, you have to assume that the thread context is arbitrary. Such would normally be the case for a storage driver since a file system driver will be the agent ultimately responsible for doing reads and writes.

The system doesn't always execute driver code in an arbitrary thread context. A driver can create its own system threads by calling *PsCreateSystemThread*. A driver can also ask the system to call it back in the context of a system thread by scheduling a work item. In these situations, we consider the thread context to be nonarbitrary. I'll discuss the mechanics of system threads and work items in Chapter 14.

Another situation in which the thread context is not arbitrary occurs when an application issues an API call that causes the I/O Manager to send an IRP directly to a driver. You can know when you write a driver whether or not this will be the case with respect to each type of IRP you handle.

You care whether the thread context is arbitrary for two reasons. First, a driver shouldn't block an arbitrary thread: it would be unfair to halt one thread while you carry out activities that benefit some other thread.

The second reason applies when a driver creates an IRP to send to some other driver. As I'll discuss more fully in Chapter 5, you need to create one kind of IRP (an asynchronous IRP) in an arbitrary thread, but you might create a different kind of IRP (a

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 31

synchronous IRP) in a nonarbitrary thread. The I/O Manager ties the synchronous kind of IRP to the thread within which you create the IRP. It will cancel the IRP automatically if that thread terminates. The I/O Manager doesn't tie an asynchronous IRP to any particular thread, though. The thread in which you create an asynchronous IRP may have nothing to do with the I/O operation you're trying to perform, and it would be incorrect for the system to cancel the IRP just because that thread happens to terminate. So it doesn't.

Symmetric Multiprocessing

Windows XP uses a so-called *symmetric* model for managing computers with multiple central processors. In this model, each CPU is treated exactly like every other CPU with respect to thread scheduling. Each CPU has its own current thread. It's perfectly possible for the I/O Manager, executing in the context of the threads running on two or more CPUs, to call subroutines in your driver simultaneously. I'm not talking about the sort of fake simultaneity with which threads execute on a single CPU—on the time scale of the computer, the threads are really taking turns. Rather, on a multiprocessor machine, different threads really do execute at the same time. As you can imagine, simultaneous execution leads to exacting requirements for drivers to synchronize access to shared data. In Chapter 4, I'll discuss the various synchronization methods that you can use for this purpose.

2.2 How the System Finds and Loads Drivers

I emphasized in the preceding section how the operating system is in overall charge of the computer and calls on device drivers to do small amounts of work with respect to hardware. Drivers have a similarly passive role in the process that causes them to be loaded in the first place. It will help you understand the rest of the book if you understand right away how the system detects hardware, determines which driver to load, and then configures the driver to manage the hardware. The system uses two slightly different methods, depending on whether the hardware is Plug and Play compatible:

- A Plug and Play device has an electronic signature that the system can detect. For Plug and Play devices, a system bus driver detects the existence of the hardware and reads the signature to determine what kind of hardware it is. Thereafter, an automatic process based on the registry and INF files allows the system to load the right driver.
- A legacy device does not have any electronic signature, so the system can't detect it automatically. The end user must therefore initiate the "detection" process by invoking the Add New Hardware Wizard, which ends with the system knowing that a certain new piece of hardware exists. Thereafter, the system uses the same automatic registry-and-INF-file process that's used for Plug and Play devices to load the right driver.

Whichever method the system uses to detect hardware and load a driver, the driver itself will be a WDM driver that reacts passively to calls from the operating system. On this point, WDM drivers contrast sharply with kernel-mode drivers for earlier versions of Windows NT and with VxD drivers prior to Windows 95. In those environments, you had to somehow arrange for the system to load your driver. Your driver would then scan hardware buses looking for its own hardware and decide whether to stay resident or not. In addition, your driver had to determine which I/O resources to use and take steps to prevent other drivers from taking the same resources.

2.2.1 Device and Driver Layering

Before I can make sense of the hardware detection and driver loading processes, I need to explain the concept of driver layering illustrated in Figure 2-2. In the figure, the left column represents an upwardly linked stack of kernel DEVICE_OBJECT structures, all of which relate to how the system manages a single piece of hardware. The middle column represents the collection of device drivers that have roles to play in the management. The right column illustrates the flow of an IRP through the drivers.

In the Windows Driver Model, each hardware device has at least two device drivers. One of these drivers, which we call the function driver, is what you've always thought of as being the device driver. It understands all the details about how to make the hardware work. It's responsible for initiating I/O operations, for handling the interrupts that occur when those operations finish, and for providing a way for the end user to exercise any control over the device that might be appropriate.

NOTE
A monolithic WDM function driver is a single executable file with dynamic links to NTOSKRNL.EXE, which contains the kernel of the operating system, and HAL.DLL, which contains the hardware abstraction layer (HAL). A function driver can dynamically link to other kernel-mode DLLs too. In situations in which Microsoft has provided a class driver for your type of hardware, your minidriver will dynamically link to the class driver. The combination of minidriver and class driver adds up to a single function driver. You may see pictures in which things called class drivers appear to be above or below a minidriver. I prefer to think of those so-called "class" drivers as free-standing filter drivers and to use the term *class driver* solely for drivers that are next to minidrivers, reached by means of explicit imports, and acting as partners that the minidriver willingly brings into play.

IPR2024-00292
KASPERSKY EX 1008 Page 32

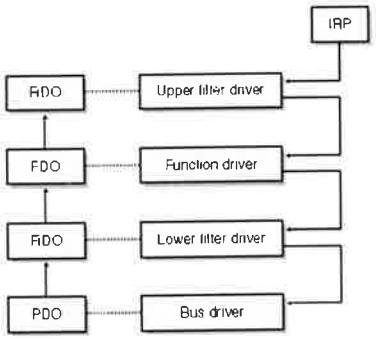


Figure 2-2. *Layering of device objects and drivers in the Windows Driver Model.*

We call the other of the two drivers that every device has the *bus driver*. It's responsible for managing the connection between the hardware and the computer. For example, the bus driver for the Peripheral Component Interconnect (PCI) bus is the software component that actually detects that your card is plugged into a PCI slot and determines the requirements your card has for I/O-mapped or memory-mapped connections with the host. It's also the software that turns on or off the flow of electrical current to your card's slot.

Some devices have more than two drivers. We use the generic term *filter driver* to describe these other drivers. Some filter drivers simply watch as the function driver performs I/O. More often, a software or hardware vendor supplies a filter driver to modify the behavior of an existing function driver in some way. *Upper filter* drivers see IRPs before the function driver, and they have the chance to support additional features that the function driver doesn't know about. Sometimes an upper filter can perform a workaround for a bug or other deficiency in the function driver or the hardware. *Lower filter* drivers see IRPs that the function driver is trying to send to the bus driver. (A lower filter is below the function driver in the stack but still above the bus driver.) In some cases, such as when the device is attached to a universal serial bus (USB), a lower filter can modify the stream of bus operations that the function driver is trying to perform.

Referring once again to Figure 2-2, notice that each of the four drivers shown for a hypothetical device has a connection to one of the *DEVICE_OBJECT* structures in the left column. The acronyms used in the structures are these:

- PDO stands for physical device object. The bus driver uses this object to represent the connection between the device and the bus.
- FDO stands for function device object. The function driver uses this object to manage the functionality of the device.
- FiDO stands for filter device object. A filter driver uses this object as a place to store the information it needs to keep about the hardware and its filtering activities. (The early beta releases of the Windows 2000 DDK used the term FiDO, and I adopted it then. The DDK no longer uses this term because, I guess, it was considered too frivolous.)

What Is a Bus?
I've already used the terms *bus* and *bus driver* pretty freely without explaining what they mean. For purposes of the WDM, a bus is anything that you can plug a device into, either physically or metaphorically.

This is a pretty broad definition. Not only does it include items such as the PCI bus, but it also includes a Small Computer System Interface (SCSI) adapter, a parallel port, a serial port, a USB hub, and so on—anything, in fact, that can have another device plugged into it.

The definition also includes a notional *root bus* that exists only as a figment of our imagination. Think of the root bus as being the bus into which all legacy devices plug. Thus, the root bus is the parent of a non-PnP Industry Standard Architecture (ISA) card or of a SmartCard reader that connects to the serial port but doesn't answer with a standard identification string to the serial port enumeration signals. We also consider the root bus to be the parent of the PCI bus—this is because the PCI bus does not itself announce its presence electronically, and the operating system therefore has to treat it like a legacy device.

2.2.2 Plug and Play Devices

To repeat what I said earlier, a Plug and Play device is one that has an electronic signature that a bus driver can interrogate to learn the identity of a device. Here are some examples of these signatures:

- A PCI card has a configuration space that the PCI bus driver can read via dedicated memory or I/O port addresses. The configuration space contains vendor and product identification information.
- A USB device returns a device descriptor in response to a standardized control-pipe transaction. The device descriptor contains vendor and product identification information.

IPR2024-00292
KASPERSKY EX 1008 Page 33

- A Personal Computer Memory Card International Association (PCMCIA) device has attribute memory that the PCMCIA bus driver can read in order to determine the identity of the card.

A bus driver for a Plug and Play bus has the ability to enumerate its bus by scanning all possible slots at start-up time. Drivers for buses that support hot plugging of devices during a session (as do USB and PCMCIA) also monitor some sort of hardware signal that indicates arrival of a new device, whereupon the driver reenumerates its bus. The end result of the enumeration or reenumeration process is a collection of PDOs. See point 1 in Figure 2-3.

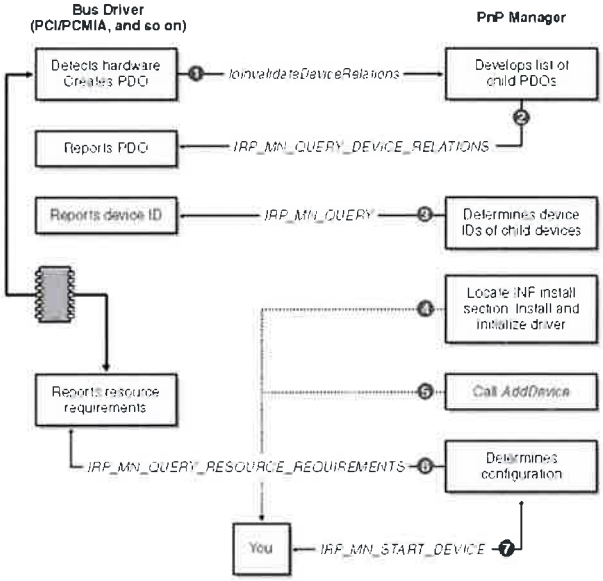


Figure 2-3. *Installing a Plug and Play device.*

When a bus driver detects the insertion or removal of hardware, it calls *IoInvalidateDeviceRelations* to notify the PnP Manager that the bus's population of child devices has changed. To obtain an updated list of the PDOs for the child devices, the PnP Manager sends an IRP to the bus driver. The major function code for this IRP is *IRP_MJ_PNP*, and the minor function code is *IRP_MN_QUERY_DEVICE_RELATIONS*, with a code indicating that the PnP Manager is looking for the so-called "bus" relations. This is point 2 in Figure 2-3.

NOTE
Each IRP has a major and a minor function code. The major function code indicates what sort of request the IRP contains. *IRP_MJ_PNP* is the major function code for requests that the PnP Manager makes. With some of the major function codes, including *IRP_MJ_PNP*, the minor function code is required to further specify the operation.In response to the bus relations query, the bus driver returns its list of PDOs. The PnP Manager can easily determine which of the PDOs represent devices that it hasn't yet initialized. Let's focus on the PDO for your hardware for the time being and see what happens next.

The PnP Manager will send another IRP to the bus driver, this time with the minor function code *IRP_MN_QUERY_ID*. This is point 3 in Figure 2-3. In fact, the PnP Manager sends several such IRPs, each with an operand that instructs the bus driver to return a particular type of identifier. One of the identifiers, the device identifier, uniquely specifies the type of device. A device identifier is just a string, and it might look like one of these examples:

```
PCI\VEN_102C&DEV_00E0&SUBSYS_00000000
USB\VID_0547&PID_2125&REV_0002
PCMCIA\MEGAHERTZ-CC10BT/2-BF05
```

NOTE
Each bus driver has its own scheme for formatting the electronic signature information it gathers into an identifier string. I'll discuss the identifier strings used by common bus drivers in Chapter 15. That chapter is also the place to look for information about INF files and about where the various registry keys described in the text are in the registry hierarchy and what sorts of information are kept in the keys.The PnP Manager uses the device identifier to locate a hardware key in the system registry. For the moment, let's assume that this is the first time your particular device has been plugged into the computer. In that case, there won't yet be a hardware key for your type of device. This is where the setup subsystem steps in to figure out what software is needed to support your device. (See point 4 in Figure 2-3.)

IPR2024-00292
KASPERSKY EX 1008 Page 34

The PnP Manager uses the device identifier to locate a *hardware key* in the system registry. For the moment, let's assume that this is the first time your particular device has been plugged into the computer. In that case, there won't yet be a hardware key for your type of device. This is where the setup subsystem steps in to figure out what software is needed to support your device. (See point 4 in Figure 2-3.)

Installation instructions for all types of hardware exist in files with the extension .INF. Each INF contains one or more model statements that relate particular device identifier strings to install sections within that INF file. Confronted with brand-new hardware, then, the setup subsystem tries to find an INF file containing a model statement that matches the device identifier. It will be your responsibility to provide this file, which is why I labeled the box You that corresponds to this step. I'm being deliberately vague at this point about how the system searches for INF files and ranks the several model statements it's likely to find. I'll burden you with these details in Chapter 15, but it would be a bit much to do that just yet.

When the setup subsystem finds the right model statement, it carries out the instructions you provide in an install section. These instructions probably include copying some files onto the end user's hard drive, defining a new driver service in the registry, and so on. By the end of the process, the setup program will have created the hardware key in the registry and installed all of the software you provided.

Now step back a few paragraphs and suppose that this was not the first time this particular computer had seen an instance of your hardware. For example, maybe we're talking about a USB device that the user introduced to the system long ago and that the user is now reattaching to the system. In that case, the PnP Manager would have found the hardware key and would not have needed to invoke the setup program. So the PnP Manager would skip around all the setup processing to point 5 in Figure 2-3.

At this point, the PnP Manager knows there is a device and that your driver is responsible for it. If your driver isn't already loaded in virtual memory, the PnP Manager calls the Memory Manager to map it in. The system doesn't read the disk file containing your driver directly into memory. Instead, it creates a file mapping that causes the driver code and data to be fetched by paging I/O. The fact that the system uses a file mapping really doesn't affect you much except that it has the side effect of making you be careful later on about when you allow your driver to be unmapped. The Memory Manager then calls your *DriverEntry* routine.

Next the PnP Manager calls your *AddDevice* routine to inform your driver that a new instance of your device has been discovered. (See point 5 in Figure 2-3.) Then the PnP Manager sends an IRP to the bus driver with the minor function code *IRP_MN_QUERY_RESOURCE_REQUIREMENTS*. This IRP is basically asking the bus driver to describe the requirements your device has for an interrupt request line, for I/O port addresses, for I/O memory addresses, and for system DMA channels. The bus driver constructs a list of these resource requirements and reports them back. (See point 6 in Figure 2-3.)

Finally the PnP Manager is ready to configure the hardware. It works with a set of resource arbitrators to assign resources to your device. If that can be done—and it usually can be—the PnP Manager sends an *IRP_MJ_PNP* to your driver with the minor function code *IRP_MN_START_DEVICE*. Your driver handles this IRP by configuring and connecting various kernel resources, following which your hardware is ready to use.

Windows NT Drivers Contrasted

The process described in the text for how Windows XP (and, indeed, Windows 2000, Windows 95, and all successors of Windows 95) finds and loads drivers requires the driver to be relatively passive. Windows NT 4.0 and before worked quite differently. In those systems, you would have provided some sort of setup program to install your driver. Your setup program would have modified the registry to cause your driver to be loaded during the next system restart. At that time, the system would load your driver and call your DriverEntry routine.

Your DriverEntry would have somehow determined which instances of your hardware were actually present. You might have scanned all possible slots of a PCI bus, for example, or assumed that each instance of your device corresponded to a subkey in the registry.

After detecting your own hardware, your DriverEntry routine would go on to assign and reserve I/O resources and then to do the configuration and connection steps that a present-day WDM driver does. As you can see, then, WDM drivers have much less work to do to get started than did drivers for earlier versions of Windows NT.

2.2.3 Legacy Devices

I use the term *legacy device* to describe any device that isn't Plug and Play, meaning that the operating system can't detect its existence automatically. Let's suppose your device fits this category. After purchasing your device, the end user will first invoke the Add New Hardware Wizard and will make a series of dialog selections to lead the setup program to an install section in an INF file. (See Figure 2-4, point 1.)

The setup program follows the instructions in the install section by creating registry entries for use by the root enumerator (point 2 in Figure 2-4). The registry entries might include a logical configuration that lists the I/O resource requirements for the device (point 3).

Finally the setup program instructs the end user to restart the system (point 4). The designers of the setup system expected that the end user would now need to follow the manufacturer's directions to configure the card by setting jumpers or switches and would then need to insert the card into an expansion slot of a powered-down computer.

IPR2024-00292
KASPERSKY EX 1008 Page 35

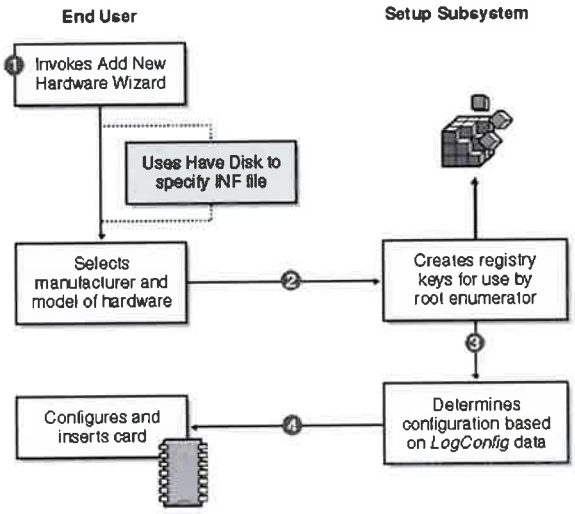


Figure 2-4. *The detection process for a legacy device.*

Following the restart (or following the end user's decision to bypass the restart), the root enumerator will scan the registry and find the newly added device. Thereafter, the process of loading your driver is nearly identical to that for a Plug and Play device. See Figure 2-5.

NOTE
Most of the sample drivers in the companion content are for fake hardware, and you install them as if the (nonexistent) hardware were a legacy device. One or two of the samples work with I/O ports and interrupts. The respective INF files contain LogConfig sections to cause the PnP Manager to assign these resources. If you install one of these drivers via the Add New Hardware Wizard, the system will think that a power-off restart is needed, but you don't really need to restart.

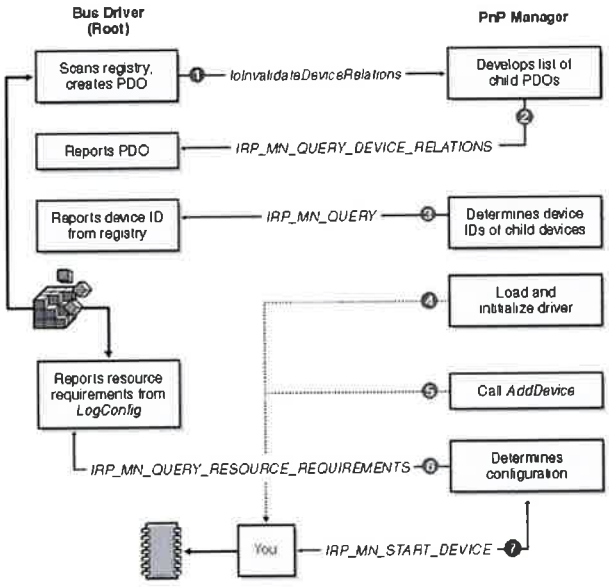


Figure 2-5. *Loading a legacy driver.*

2.2.4 Recursive Enumeration

In the preceding sections, I described how the system loads the correct driver for a single device. That description begs the question of how the system manages to load drivers for all the hardware in the computer. The answer is that it uses a recursive process.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 36

In the first instance, the PnP Manager invokes the root enumerator to find all hardware that can't electronically announce its presence—including the primary hardware bus (such as PCI). The root bus driver gets information about the computer from the registry, which was initialized by the Windows XP Setup program. Setup got the information by running an elaborate hardware detection program and by asking the end user suitable questions. Consequently, the root bus driver knows enough to create a PDO for the primary bus.

The function driver for the primary bus can then enumerate its own hardware electronically. When a bus driver enumerates hardware, it acts in the guise of an ordinary function driver. Having detected a piece of hardware, however, the driver switches roles: it becomes a bus driver and creates a new PDO for the detected hardware. The PnP Manager then loads drivers for this device PDO, as previously discussed. It might happen that the function driver for the device enumerates still more hardware, in which case the whole process repeats recursively. The end result will be a tree like that shown in Figure 2-6, wherein a bus-device stack branches into other device stacks for the hardware attached to that bus. The dark-shaded boxes in the figure illustrate how one driver can wear an "FDO hat" to act as the function driver for its hardware and a "PDO hat" to act as the bus driver for the attached devices.

2.2.5 Order of Driver Loading

I said earlier that devices can have upper and lower filter drivers as well as a function driver. Two registry keys associated with the device contain information about filter drivers. The hardware key, which contains information about an instance of your hardware, can have *UpperFilters* and *LowerFilters* values that specify filter drivers for that instance. There is another registry key for the class to which the device belongs. For example, a mouse belongs to the Mouse class, which you could probably have figured out without me telling you. The class key can also contain *UpperFilters* and *LowerFilters* values. They specify filter drivers that the system will load for every device belonging to the class.

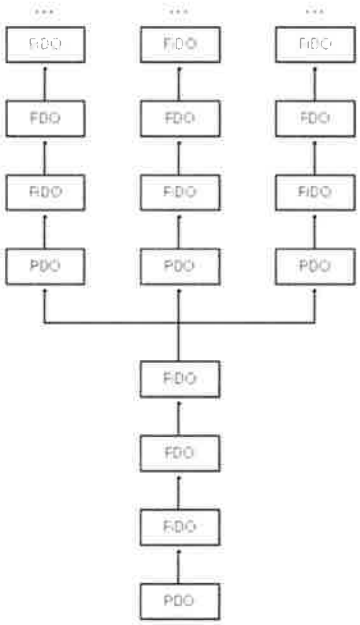


Figure 2-6. *Layering of recursively enumerated devices.*

No matter where it appears, an *UpperFilters* or *LowerFilters* value is of type *REG_MULTI_SZ* and can therefore contain one or more null-terminated Unicode string values.

NOTE
Windows 98/Me doesn't support the *REG_MULTI_SZ* registry type and doesn't fully support Unicode. In Windows 98/Me, the UpperFilters and LowerFilters values are *REG_BINARY* values that contain multiple null-terminated ANSI strings followed by an extra null terminator.

It may be important in some situations to understand in what order the system calls drivers. The actual process of "loading" a driver entails mapping its code image into virtual memory, and the order in which that's done is actually not very interesting. You might be interested, however, in knowing the order of calls to the *AddDevice* functions in the various drivers. (Refer to Figure 2-7.)

1. The system first calls the *AddDevice* functions in any lower filter drivers specified in the device key for the device, in the order in which they appear in the *LowerFilters* value.

2. Then the system calls *AddDevice* in any lower filter drivers specified in the class key. Again, the calls occur in the order

IPR2024-00292
KASPERSKY EX 1008 Page 37

in which the drivers appear in the *LowerFilters* string.

3. The system calls *AddDevice* in the driver specified by the Service value in the device key. This is the function driver.
4. The system calls *AddDevice* for any upper filter drivers specified in the device key, in the order in which they appear in the *UpperFilters* data string.
5. Finally the system calls *AddDevice* for any upper filter drivers specified in the class key, in the order in which they appear in the *UpperFilters* data string.

As I explain later in this chapter, each *AddDevice* function creates a kernel DEVICE_OBJECT and links it into the stack rooted in the PDO. Therefore, the order of calls to *AddDevice* governs the order of device objects in the stack and, ultimately, the order in which drivers see IRPs.

NOTE
You might have noticed that the loading of upper and lower filters belonging to the class and to the device instance isn't neatly nested as you might have expected. Before I knew the facts, I guessed that device-level filters would be closer to the function driver than class-level filters.

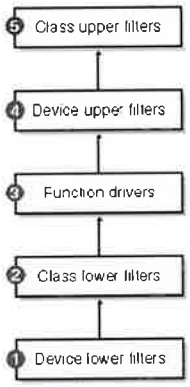


Figure 2-7. *Order of AddDevice calls*

2.2.6 IRP Routing

The formal layering of drivers in the WDM facilitates routing IRPs from one driver to another in a predictable way. Figure 2-2 illustrates the general idea: whenever the system wants to carry out an operation on a device, it sends an IRP to the topmost filter driver in the stack. That driver can decide to process the IRP, to pass the IRP down to the next level, or to do both. Each driver that sees the IRP makes the same decision. Eventually, the IRP might reach the bus driver in its PDO role. The bus driver does not usually pass the IRP any further, despite what Figure 2-6 might seem to imply. Rather, the bus driver usually completes the IRP. In some situations, the bus driver will pass the same IRP to the stack (the parent driver stack) in which it plays the FDO role. In other situations, the bus driver will create a secondary IRP and pass it to the parent driver stack.

How the Device Stack Is Implemented
I'll show you the *DEVICE_OBJECT* data structure a bit later in this chapter. The opaque field *AttachedDevice* links device objects into a vertical stack. Starting with the PDO, each device object points to the object immediately above it. There is no documented downward pointer—drivers must keep track on their own of what's underneath them. (In fact, *IoAttachDeviceToDeviceStack* does set up a downward pointer in a structure for which the DDK doesn't have a complete declaration. It would be unwise to try to reverse-engineer that structure because it's subject to change at any time.)

The *AttachedDevice* field is purposely not documented because its proper use requires synchronization with code that might be deleting device objects from memory. You and I are allowed to call *IoGetAttachedDeviceReference* to find the topmost device object in a given stack. That function also increments a reference count that will prevent that object from being prematurely removed from memory. If you wanted to work your way down to the PDO, you could send your own device an *IRP_MJ_PNP* request with the minor function code *IRP_MN_QUERY_DEVICE_RELATIONS* and the Type parameter *TargetDeviceRelation*. The PDO's driver will answer by returning the address of the PDO. It would be a great deal easier just to remember the PDO address when you first create the device object.

Similarly, to know which device object is immediately underneath you, you need to save a pointer when you first add your object to the stack. Since each of the drivers in a stack will have its own unknowable way of implementing the downward pointers used for IRP dispatching, it's not practical to alter the device stack once the stack has been created.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 38

A few examples should clarify the relationship between FiDOs, FDOs, and PDOs. The first example concerns a read operation directed to a device that happens to be on a secondary PCI bus that itself attaches to the main bus through a PCI-to-PCI bridge chip. To keep things simple, let's suppose there's one FiDO for this device, as illustrated in Figure 2-8. You'll learn in later chapters that a read request turns into an IRP with the major function code IRP_MJ_READ. Such a request would flow first to the upper FiDO and then to the function driver for the device. (That driver is the one for the device object marked FDO_{dev} in the figure.) The function driver calls the HAL directly to perform its work, so none of the other drivers in the figure will see the IRP.

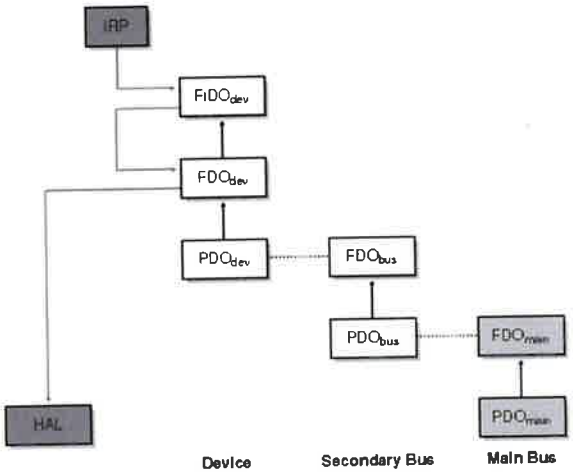


Figure 2-8. *The flow of a read request for a device on a secondary bus.*

A variation on the first example is shown in Figure 2-9. Here we have a read request for a device plugged into a USB hub that itself is plugged into the host controller. The complete device tree therefore contains stacks for the device, for the hub, and for the host controller. The *IRP_MJ_READ* flows through the FiDO to the function driver, which then sends one or more IRPs of a different kind downward to its own PDO. The PDO driver for a USB device is USBHUB.SYS, and it forwards the IRPs to the topmost driver in the host controller device stack, skipping the two-driver stack for the USB hub in the middle of the figure.

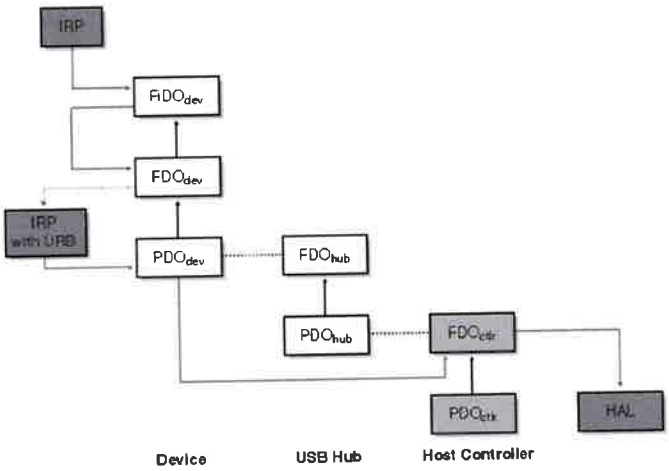


Figure 2-9. *The flow of a read request for a USB device.*

The third example is similar to the first except that the IRP in question is a notification concerning whether a disk drive on a PCI bus will be used as the repository for a system paging file. You'll learn in Chapter 6 that this notification takes the form of an *IRP_MJ_PNP* request with the minor function code *IRP_MN_DEVICE_USAGE_NOTIFICATION*. In this case, the FiDO driver passes the request to the FDO_{dev} driver, which takes note of it and passes it further down the stack to the PDO_{dev} driver. This particular notification has implications about how other I/O requests that concern the PnP system or power management will be handled, so the PDO_{dev} driver sends an identical notification to the stack within which is the FDO_{bus}, as illustrated in Figure 2-10. (Not all bus drivers work this way, but the PCI bus does.)

IPR2024-00292
KASPERSKY EX 1008 Page 39

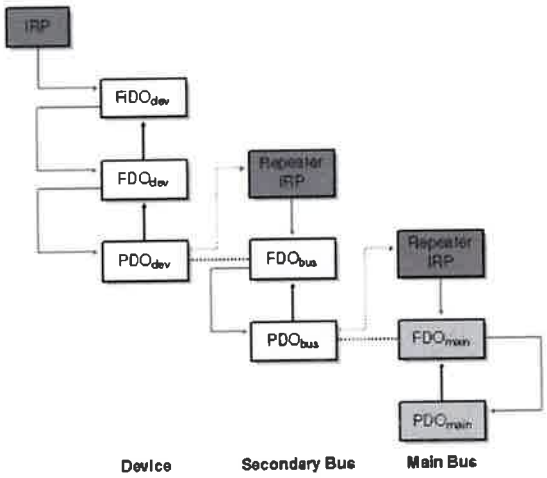


Figure 2-10. *The flow of a device usage notification*

Visualizing the Device Tree
To better visualize the way device objects and drivers are layered, it helps to have a tool. I wrote the DEVVIEW utility, which you'll find in the companion content, for this purpose. With the USB42 sample for Chapter 12 plugged into a secondary USB hub, I ran DEVVIEW and generated the two screen shots shown in Figure 2-11 and Figure 2-12.

This particular device uses only two device objects. The PDO is managed by USBHUB.SYS, whereas the FDO is managed by USB42. In the first of these screen shots, you can see other information about the PDO.

It's worth experimenting with DEVVIEW on your own system to see how various drivers are layered for the hardware you own.

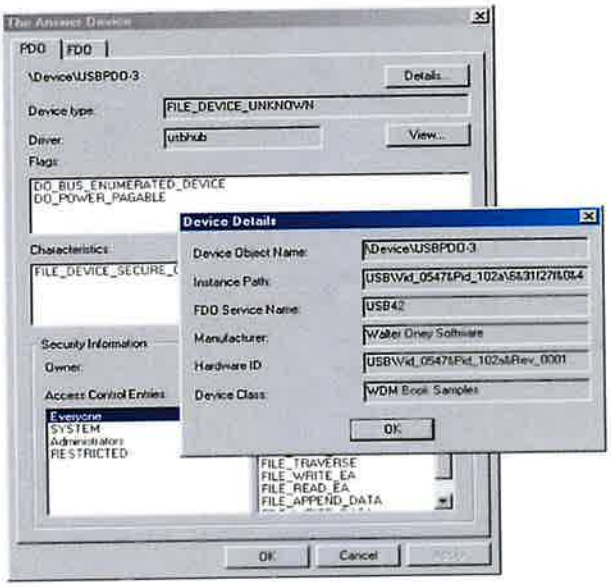


Figure 2-11. *DEVVIEW information about USB42's PDO.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 40

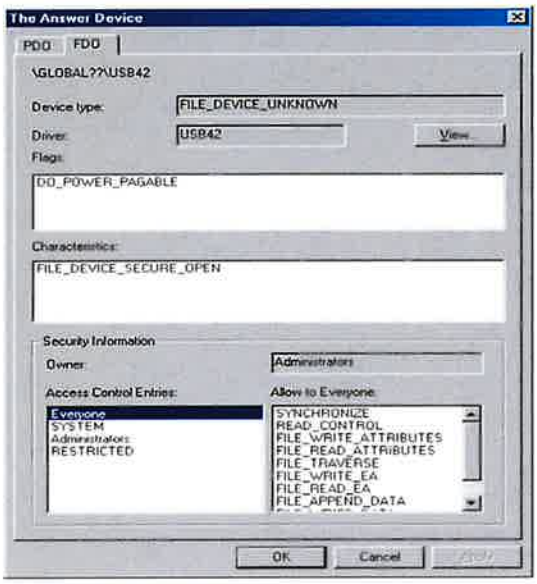


Figure 2-12. DEVVIEW information about USB42's FDO.

2.3 The Two Basic Data Structures

This section describes the two most basic data structures that concern a WDM driver: the driver object and the device object. The driver object represents the driver itself and contains pointers to all the driver subroutines that the system will ever call on its own motion. (For the sake of completeness, you should know that you often provide pointers to other routines within your driver as arguments in various kernel-mode service calls.) The device object represents an instance of hardware and contains data to help you manage that instance.

2.3.1 Driver Objects

The I/O Manager uses a driver object data structure to represent each device driver. (See Figure 2-13.) Like many of the data structures we'll be discussing, the driver object is partially opaque. This means that you and I are supposed to directly access or change only certain fields in the structure, even though the DDK headers declare the entire structure. I've shown the opaque fields of the driver object in the figure with a gray background. These opaque fields are analogous to the private and protected members of a C++ class, and the accessible fields are analogous to public members.

The DDK headers declare the driver object, and all other kernel-mode data structures for that matter, in a stylized way, as this excerpt from WDM.H illustrates:

```
typedef struct  DRIVER OBJECT {
  CSHORT Type;
  CSHORT Size;
  :
  } DRIVER_OBJECT, *PDRIVER_OBJECT;
```

That is, the header declares a structure with the type name DRIVER_OBJECT. It also declares a pointer type (*PDRIVER_OBJECT*) and assigns a structure tag (*_DRIVER_OBJECT*). This declaration pattern appears many places in the DDK, and I won't mention it again. The headers also declare a small set of type names (such as *CSHORT*) to describe the atomic data types used in kernel mode. *CSHORT*, for example, means "signed short integer used as a cardinal number." Table 2-1 lists some of these names.

IPR2024-00292
KASPERSKY EX 1008 Page 41

Type Name	Description
PVOID, PVOID64	Generic pointers (default precision and 64-bit precision)
NTAPI	Used with service function declarations to force use of *__stdcall* calling convention on i86 architectures
VOID	Equivalent to "void"
CHAR, PCHAR	8-bit character, pointer to same (signed or not according to compiler default)
UCHAR, PUCHAR	Unsigned 8-bit character, pointer to same
SCHAR, PSCHAR	Signed 8-bit character, pointer to same
SHORT, PSHORT	Signed 16-bit integer, pointer to same
CSHORT	Signed short integer, used as a cardinal number
USHORT, PUSHORT	Unsigned 16-bit integer, pointer to same
LONG, PLONG	Signed 32-bit integer, pointer to same
ULONG, PULONG	Unsigned 32-bit integer, pointer to same
WCHAR, PWSTR, PWCHAR	Wide (Unicode) character or string
PCWSTR	Pointer to constant Unicode string
NTSTATUS	Status code (typed as signed long integer)
LARGE_INTEGER	Signed 64-bit integer
ULARGE_INTEGER	Unsigned 64-bit integer
PSZ, PCSZ	Pointer to ASCIIZ (single-byte) string or constant string
BOOLEAN, PBOOLEAN	TRUE or FALSE (equivalent to *UCHAR*)

Table 2-1. *Common Type Names for Kernel-Mode Drivers*

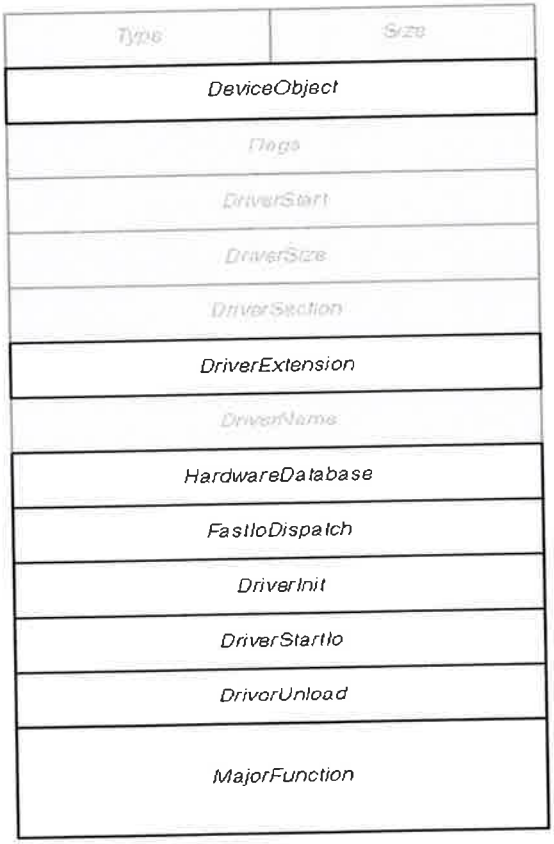


Figure 2-13. *The DRIVER_OBJECT data structure.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 42

NOTE
Note on 64-bit Types: The DDK headers contain type names that will make it relatively painless for driver authors to compile the same source code for either 32-bit or 64-bit Intel platforms. For example, instead of blithely assuming that a long integer and a pointer are the same size, you should declare variables that might be either a LONG_PTR or a ULONG_PTR. Such a variable can hold either a long (or unsigned long) or a pointer to something. Also, for example, use the type SIZE_T to declare an integer that can count as high as a pointer might span—you'll get a 64-bit integer on a 64-bit platform. These and other 32/64 typedefs are in the DDK header file named BASETSD.H.

I'll briefly discuss the accessible fields of the driver object structure now.

DeviceObject (*PDEVICE_OBJECT*) anchors a list of device object data structures, one for each of the devices managed by the driver. The I/O Manager links the device objects together and maintains this field. The *DriverUnload* function of a non-WDM driver would use this field to traverse the list of device objects in order to delete them. A WDM driver probably doesn't have any particular need to use this field.

DriverExtension (*PDRIVER_EXTENSION*) points to a small substructure within which only the *AddDevice* (*PDRIVER_ADD_DEVICE*) member is accessible to the likes of us. (See Figure 2-14.) *AddDevice* is a pointer to a function within the driver that creates device objects; this function is rather a big deal, and I'll discuss it at length in the section "The *AddDevice* Routine" later in this chapter.

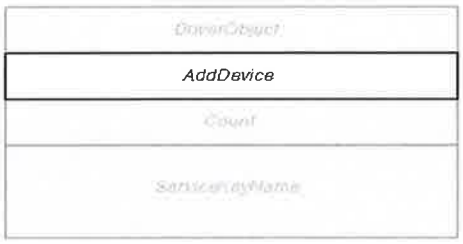


Figure 2-14. *The DRIVER_EXTENSION data structure.*

HardwareDatabase (*PUNICODE_STRING*) describes a string that names a hardware database registry key for the device. This is a name like \Registry\Machine\Hardware\Description\System and names the registry key within which resource allocation information resides. WDM drivers have no need to access the information below this key because the PnP Manager performs resource allocation automatically. The name is stored in Unicode. (In fact, all kernel-mode string data uses Unicode.) I'll discuss the format and the use of the *UNICODE_STRING* data structure in the next chapter.

FastIoDispatch (*PFAST_IO_DISPATCH*) points to a table of function pointers that file system and network drivers export. How these functions are used is beyond the scope of this book. If you're interested in learning more about file system drivers, consult Rajeev Nagar's Windows NT File System Internals: A Developer's Guide (O'Reilly & Associates, 1997).

DriverStartIo (*PDRIVER_STARTIO*) points to a function in your driver that processes I/O requests that the I/O Manager has serialized for you. I'll discuss request queuing in general and the use of this routine in particular in Chapter 5.

DriverUnload (*PDRIVER_UNLOAD*) points to a cleanup function in your driver. I'll discuss this function a bit further on in connection with *DriverEntry*, but you might as well know now that a WDM driver probably doesn't have any significant cleanup to do anyway.

MajorFunction (array of *PDRIVER_DISPATCH*) is a table of pointers to functions in your driver that handle each of the roughly two dozen types of I/O requests. This table is also something of a big deal, as you might guess, because it defines how I/O requests make it into your code.

2.3.2 Device Objects

Figure 2-15 illustrates the format of a device object and uses the same shading convention for opaque fields that I used in the preceding discussion of driver objects. As the author of a WDM driver, you'll create some of these objects by calling IoCreateDevice.

DriverObject (*PDRIVER_OBJECT*) points to the object describing the driver associated with this device object, usually the one that called *IoCreateDevice* to create it.

NextDevice (*PDEVICE_OBJECT*) points to the next device object that belongs to the same driver as this one. This field is the one that links device objects together starting from the driver object's *DeviceObject* member. There's probably no reason for a WDM driver to use this field. That's just as well because proper use of this pointer requires synchronization using an internal system lock that's not exposed for access by device drivers.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 43

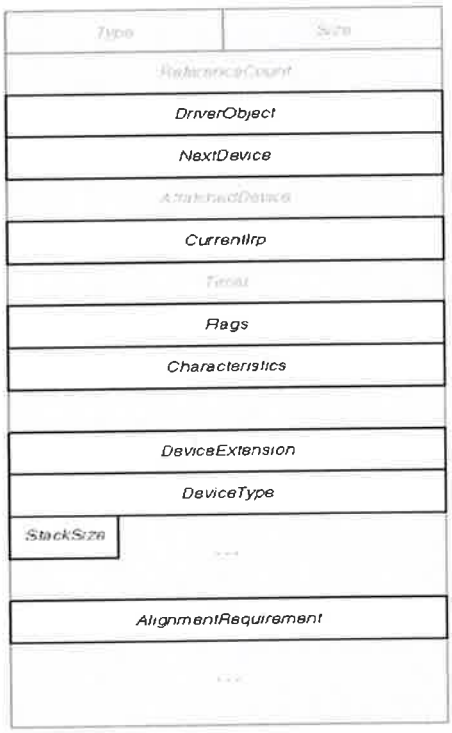


Figure 2-15. *The DEVICE_OBJECT data structure.*

CurrentIrp (*PIRP*) is used by the Microsoft IRP queuing routines *StartPacket* and *StartNextPacket* to record the IRP most recently sent to your *StartIo* routine. WDM drivers should implement their own IRP queues (see Chapter 5) and may have no use for this field.

Flags (*ULONG*) contains a collection of flag bits. Table 2-2 lists the bits that are accessible to driver writers.

Flag	Description
DO_BUFFERED_IO	Reads and writes use the buffered method (system copy buffer) for accessing user-mode data.
DO_EXCLUSIVE	Only one thread at a time is allowed to open a handle.
DO_DIRECT_IO	Reads and writes use the direct method (memory descriptor list) for accessing user-mode data.
DO_DEVICE_INITIALIZING	Device object isn't initialized yet.
DO_POWER_PAGABLE	*IRP_MJ_PNP* must be handled at *PASSIVE_LEVEL*.
DO_POWER_INRUSH	Device requires large inrush of current during power-on.

Table 2-2. *Flags in a DEVICE_OBJECT Data Structure*

Flag	Description
FILE_REMOVABLE_MEDIA	Media can be removed from device.
FILE_READ_ONLY_DEVICE	Media can only be read, not written.
FILE_FLOPPY_DISKETTE	Device is a floppy disk drive.
FILE_WRITE_ONCE_MEDIA	Media can be written once.
FILE_REMOTE_DEVICE	Device accessible through network connection.
FILE_DEVICE_IS_MOUNTED	Physical media is present in device.
FILE_VIRTUAL_VOLUME	This is a virtual volume.
FILE_AUTOGENERATED_DEVICE_NAME	I/O Manager should automatically generate a name for this device.
FILE_DEVICE_SECURE_OPEN	Force security check during open.

Table 2-3. *Characteristics Flags in a DEVICE_OBJECT Data Structure*

Characteristics (*ULONG*) is another collection of flag bits describing various optional characteristics of the device. (See Table 2-3.) The I/O Manager initializes these flags based on an argument to *IoCreateDevice*. Filter drivers propagate some of them

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 44

upward in the device stack. (See the detailed discussion of filter drivers in Chapter 16 for more information about flag propagation.)

DeviceExtension (*PVOID*) points to a data structure you define that will hold per-instance information about the device. The I/O Manager allocates space for the structure, but its name and contents are entirely up to you. A common convention is to declare a structure with the type name *DEVICE_EXTENSION*. To access it given a pointer (for example, fdo) to the device object, use a statement like this one:

```
PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```

It happens to be true (now, anyway) that the device extension immediately follows the device object in memory. It would be a bad idea to rely on this always being true, though, especially when the documented method of following the DeviceExtension pointer will always work.

DeviceType (*DEVICE_TYPE*) is an enumeration constant describing what type of device this is. The I/O Manager initializes this member based on an argument to *IoCreateDevice*. Filter drivers might conceivably need to inspect it. At the date of this writing, there are over 50 possible values for this member. Consult the DDK documentation entry "Specifying Device Types" in the MSDN Library for a list.

StackSize (*CCHAR*) counts the number of device objects starting from this one and descending all the way to the PDO. The purpose of this field is to inform interested parties regarding how many stack locations should be created for an IRP that will be sent first to this device's driver. WDM drivers don't normally need to modify this value, however, because the support routine they use for building the device stack (*IoAttachDeviceToDeviceStack*) does so automatically.

AlignmentRequirement (*ULONG*) specifies the required alignment for data buffers used in read or write requests to this device. WDM.H contains a set of manifest constants ranging from *FILE_BYTE_ALIGNMENT* and *FILE_WORD_ALIGNMENT* up to *FILE_512_BYTE_ALIGNMENT* for these values. The values are just powers of 2 minus 1. For example, the value 0x3F is *FILE_64_BYTE_ALIGNMENT*.

2.4 The *DriverEntry* Routine

In preceding sections, I said that the PnP Manager loads the drivers needed for hardware and calls their *AddDevice* functions. A given driver might be used for more than one piece of similar hardware, and there's some global initialization that the driver needs to perform only once when it's loaded for the first time. That global initialization is the responsibility of the *DriverEntry* routine:

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath)
{

}
```

NOTE
You call the main entry point to a kernel-mode driver "DriverEntry" because the build script—if you use standard procedures—will instruct the linker that *DriverEntry* is the entry point, and it's best to make your code match this assumption (or else change the build script, but why bother?).

Sample Code
You can experiment with the ideas discussed in this chapter using the STUPID sample driver. STUPID implements *DriverEntry* and *AddDevice* but nothing else. It's similar to the very first driver I attempted to write when I was learning.

Before I describe the code you'd write inside *DriverEntry*, I want to mention a few things about the function prototype itself. Unbeknownst to you and me (unless we look carefully at the compiler options used in the build script), kernel-mode functions and the functions in your driver use the *__stdcall* calling convention when compiled for an x86 computer. This shouldn't affect any of your programming, but it's something to bear in mind when you're debugging. I used the extern "C" directive because, as a rule, I package my code in a C++ compilation unit—mostly to gain the freedom to declare variables wherever I please instead of only immediately after left braces. This directive suppresses the normal C++ decoration of the external name so that the linker can find this function. Thus, an x86 compile produces a function whose external name is *_DriverEntry@8*.

Another point about the prototype of *DriverEntry* is those "IN" keywords. **IN** and **OUT** are both noise words that the DDK defines as empty strings. By original intention, they perform a documentation function. That is, when you see an IN parameter, you're supposed to infer that it's purely input to your function. An **OUT** parameter is output by your function, while an **IN OUT** parameter is used for both input and output. As it happens, the DDK headers don't always use these keywords intuitively, and there's not a great deal of point to them. To give you just one example out of many: *DriverEntry* claims that the *DriverObject* pointer is **IN**; indeed, you don't change the pointer, but you will assuredly change the object to which it points.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 45

The last general thing I want you to notice about the prototype is that it declares this function as returning an *NTSTATUS* value. *NTSTATUS* is actually just a long integer, but you want to use the typedef name *NTSTATUS* instead of *LONG* so that people understand your code better. A great many kernel-mode support routines return *NTSTATUS* status codes, and you'll find a list of them in the DDK header NTSTATUS.H. I'll have a bit more to say about status codes in the next chapter; for now, just be aware that your *DriverEntry* function will be returning a status code when it finishes.

2.4.1 Overview of *DriverEntry*

The first argument to *DriverEntry* is a pointer to a barely initialized driver object that represents your driver. A WDM driver's *DriverEntry* function will finish initializing this object and return. Non-WDM drivers have a great deal of extra work to do—they must also detect the hardware for which they're responsible, create device objects to represent the hardware, and do all the configuration and initialization required to make the hardware fully functional. The relatively arduous detection and configuration steps are handled automatically for WDM drivers by the PnP Manager, as I'll discuss in Chapter 6. If you want to know how a non-WDM driver initializes itself, consult Art Baker and Jerry Lozano's The Windows 2000 Device Driver Book (Prentice Hall, 2d ed., 2001) and Viscarola and Mason's Windows NT Device Driver Development (Macmillan, 1998).

The second argument to *DriverEntry* is the name of the service key in the registry. This string is not persistent—you must copy it if you plan to use it later. In a WDM driver, the only use I've ever made of this string is as part of WMI registration. (See Chapter 10.)

A WDM driver's main job in *DriverEntry* is to fill in the various function pointers in the driver object. These pointers indicate to the operating system where to find the subroutines you've decided to place in your driver container. They include these pointer members of the driver object:

- ***DriverUnload***
  Set this to point to whatever cleanup routine you create. The I/O Manager will call this routine just prior to unloading the driver. If there's nothing to clean up, you need to have a DriverUnload function for the system to be able to unload your driver dynamically.

- ***DriverExtension->AddDevice***
  Set this to point to your *AddDevice* function. The PnP Manager will call *AddDevice* once for each hardware instance you're responsible for. Since *AddDevice* is so important to the way WDM drivers work, I've devoted the next main section of this chapter ("The *AddDevice* Routine") to explaining what it does.

- ***DriverStartIo***
  If your driver uses the standard method of queuing I/O requests, you'll set this member of the driver object to point to your *StartIo* routine. Don't worry (yet, that is) if you don't understand what I mean by the "standard" queuing method; all will become clear in Chapter 5, where you'll discover that WDM drivers shouldn't use it.

- ***MajorFunction***
  The I/O Manager initializes this vector of function pointers to point to a dummy dispatch function that fails every request. You're presumably going to be handling certain types of IRPs—otherwise, your driver is basically going to be deaf and inert—so you'll set at least some of these pointers to your own dispatch functions. Chapter 5 discusses IRPs and dispatch functions in detail. For now, all you need to know is that you must handle two kinds of IRPs and that you'll probably be handling several other kinds as well.

A nearly complete *DriverEntry* routine will, then, look like this:

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath)
  {
1
  DriverObject->DriverUnload = DriverUnload;
  DriverObject->DriverExtension->AddDevice = AddDevice;
2
  DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
  DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
  DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchWmi;
3
  .
  .
4
  servkey.Buffer = (PWSTR) ExAllocatePool(PagedPool,
    RegistryPath->Length + sizeof(WCHAR));
  if (!servkey.Buffer)
    return STATUS_INSUFFICIENT_RESOURCES;
  servkey.MaximumLength = RegistryPath->Length + sizeof(WCHAR);
  RtlCopyUnicodeString(&servkey, RegistryPath);
  servkey.Buffer[RegistryPath->Length/sizeof(WCHAR)] = 0;
5
  return STATUS_SUCCESS;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 46

```
}
```

1. These two statements set the function pointers for entry points elsewhere in the driver. I elected to give them simple names indicative of their function: *DriverUnload* and *AddDevice*.

2. Every WDM driver must handle PNP, POWER, and SYSTEM_CONTROL I/O requests, and it should handle SYSTEM_CONTROL I/O requests. This is where you specify your dispatch functions for these requests. What's now *IRP_MJ_SYSTEM_CONTROL* was called *IRP_MJ_WMI* in some early beta releases of the Windows XP DDK, which is why I called my dispatch function *DispatchWmi*.

3. In place of this ellipsis, you'll have code to set several additional *MajorFunction* pointers.

4. If you ever need to access the service registry key elsewhere in your driver, it's a good idea to make a copy of the *RegistryPath* string here. I've assumed that you declared a global variable named servkey as a *UNICODE_STRING* elsewhere. I'll explain the mechanics of working with Unicode strings in the next chapter.

5. Returning *STATUS_SUCCESS* is how you indicate success. If you were to discover something wrong, you'd return an error code chosen from the standard set in NTSTATUS.H or from a set of error codes that you define yourself. *STATUS_SUCCESS* happens to be numerically 0.

Subroutine Naming
Many driver writers give the subroutines in their drivers names that include the name of the driver. For example, instead of defining AddDevice and DriverUnload functions, many programmers would define Stupid_AddDevice and Stupid_DriverUnload. I'm told that earlier versions of Microsoft's WinDbg debugger forced a convention like this onto (possibly unwilling) programmers because it had just one global namespace. Later versions of this debugger don't have that limitation, but you'll observe that the DDK sample drivers still follow the convention.

Now, I'm a great fan of code reuse and an indifferent typist. For me, it has seemed much simpler to have short subroutine names that are exactly the same from one project to the next. That way, I can just lift a body of code from one driver and paste it into another without needing to make a bunch of name changes. I can also compare one driver with another without having extraneous name differences clutter up the comparison results.

2.4.2 *DriverUnload*

The purpose of a WDM driver's *DriverUnload* function is to clean up after any global initialization that *DriverEntry* might have done. There's almost nothing to do. If you made a copy of the *RegistryPath* string in *DriverEntry*, though, *DriverUnload* would be the place to release the memory used for the copy:

```
VOID DriverUnload(PDRIVER OBJECT DriverObject)
  {
  RtlFreeUnicodeString(&servkey);
  }
```

If your *DriverEntry* routine returns a failure status, the system doesn't call your *DriverUnload* routine. Therefore, if *DriverEntry* generates any side effects that need cleaning up prior to returning an error status, *DriverEntry* has to perform the cleanup.

2.5 The *AddDevice* Routine

In the preceding main section, I showed how you initialize a WDM driver when it's first loaded. In general, though, a driver might be called upon to manage more than one actual device. In the WDM architecture, a driver has a special *AddDevice* function that the PnP Manager can call for each such device. The function has the following skeleton:

```
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
  :
  return STATUS SOMETHING;  // e.g., STATUS SUCCESS
  }
```

The *DriverObject* argument points to the same driver object that you initialized in your *DriverEntry* routine. The *pdo* argument is the address of the physical device object at the bottom of the device stack, even if there are already filter drivers below.

The basic responsibility of *AddDevice* in a function driver is to create a device object and link it into the stack rooted in this PDO. The steps involved are as follows:

1. Call IoCreateDevice to create a device object and an instance of your own device extension object.

2. Register one or more device interfaces so that applications know about the existence of your device. Alternatively, give

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 47

the device object a name and then create a symbolic link.

3. Next initialize your device extension and the Flags member of the device object.
4. Call IoAttachDeviceToDeviceStack to put your new device object into the stack.

Now I'll explain these steps in more detail. I'll show a complete example of AddDevice at the very end of this discussion.

NOTE
In the code snippets that follow, I've deliberately left out all the error handling that should be there. That's so I could concentrate on the normal control flow through AddDevice. You mustn't imitate this programming style in a production driver—but of course you already knew that. I'll discuss how to handle errors in the next chapter. Every code sample in the companion content has full error checking in place too.

2.5.1 Creating a Device Object

You create a device object by calling *IoCreateDevice*. For example:

```
PDEVICE_OBJECT fdo;
NTSTATUS status = IoCreateDevice(DriverObject,
  sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,
  FILE_DEVICE_SECURE_OPEN, FALSE, &fdo);
```

The first argument (*DriverObject*) is the same value supplied to *AddDevice* as the first parameter. This argument establishes the connection between your driver and the new device object, thereby allowing the I/O Manager to send you IRPs intended for the device. The second argument is the size of your device extension structure. As I discussed earlier in this chapter, the I/O Manager allocates this much additional memory and sets the *DeviceExtension* pointer in the device object to point to it.

The third argument, which is NULL in this example, can be the address of a *UNICODE_STRING* providing a name for the device object. Deciding whether to name your device object and which name to give it requires some thought, and I'll describe these surprisingly complex considerations a bit further on in the section "Should I Name My Device Object?"

The fourth argument (*FILE_DEVICE_UNKNOWN*) is one of the device types defined in WDM.H. Whatever value you specify here can be overridden by an entry in the device's hardware key or class key. If both keys have an override, the device key has precedence. For devices that fit into one of the established categories, specify the right value in one of these places because some details about the interaction between your driver and the surrounding system depend on it. In fact, the device type is crucial for the correct functioning of a file system driver or a disk or tape driver. Additionally, the default security settings for your device object depend on this device type.

The fifth argument (*FILE_DEVICE_SECURE_OPEN*) provides the Characteristics flag for the device object. (See Table 2-3.) Most of these flags are relevant for mass storage devices. The flag bit *FILE_AUTOGENERATED_DEVICE_NAME* is for use by bus and multifunction drivers when creating PDOs. I'll discuss the importance of *FILE_DEVICE_SECURE_OPEN* later in this chapter in the section "Should I Name My Device Object?" Whatever value you specify here can be overridden by an entry in the device's hardware key or class key. If both keys have an override, the hardware key has precedence.

The sixth argument to *IoCreateDevice* (*FALSE* in my example) indicates whether the device is exclusive. The I/O Manager allows only one handle to be opened by normal means to an exclusive device. Whatever value you specify here can be overridden by an entry in the device's hardware key or class key. If both keys have an override, the hardware key has precedence.

NOTE
The exclusivity attribute matters only for whatever named device object is the target of an open request. If you follow Microsoft's recommended guidelines for WDM drivers, you won't give your device object a name. Open requests will then target the PDO, but the PDO will *not* usually be marked exclusive because the bus driver generally has no way of knowing whether you need your device to be exclusive. The only time the PDO will be marked exclusive is when there's an *Exclusive* override in the device's hardware key or the class key's Properties subkey. You're best advised, therefore, to avoid relying on the exclusive attribute altogether. Instead, make your *IRP_MJ_CREATE* handler reject open requests that would violate whatever restriction you require.

The last argument (*&fdo*) points to a location where *IoCreateDevice* will store the address of the device object it creates.

If *IoCreateDevice* fails for some reason, it returns a status code and doesn't alter the *PDEVICE_OBJECT* described by the last argument. If it succeeds, it returns a successful status code and sets the *PDEVICE_OBJECT* pointer. You can then proceed to initialize your device extension and do the other work associated with creating a new device object. Should you discover an error after this point, you should release the device object and return a status code. The code to accomplish these tasks would be something like this:

```
NTSTATUS status = IoCreateDevice(...);
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 48

```
if (!NT SUCCESS(status))
  return status;
.
.
if (<some other error discovered>)
  {
  IoDeleteDevice(fdo);
  return status;
  }
```

I'll explain the *NTSTATUS* status codes and the *NT_SUCCESS* macro in the next chapter.

2.5.2 Naming Devices

Windows XP uses a centralized Object Manager to manage many of its internal data structures, including the driver and device objects I've been talking about. David Solomon and Mark Russinovich present a fairly complete explanation of the Object Manager and namespace in Chapter 3, "System Mechanisms," of *Inside Windows 2000*, Third Edition (Microsoft Press, 2000). Objects have names, which the Object Manager maintains in a hierarchical namespace. Figure 2-16 is a screen shot of my DEVVIEW application showing the top level of the name hierarchy. The objects displayed as folders in this screen shot are directory objects, which can contain subdirectories and "regular" objects. The objects displayed with other icons are examples of these regular objects. (In this respect, DEVVIEW is similar to the WINOBJ utility that you'll find in the BIN\WINNT directory of the Platform SDK. WINOBJ can't give you information about device objects and drivers, though, which is why I wrote DEVVIEW in the first place.)

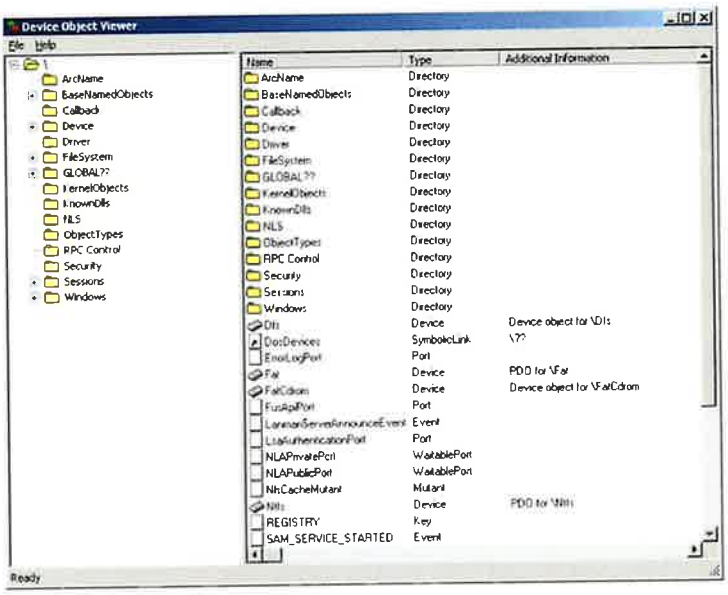


Figure 2-16. *Using DEVVIEW to view the namespace.*

Device objects can have names that conventionally live in the \Device directory. Names for devices serve two purposes in Windows XP. Giving your device object a name allows other kernel-mode components to find it by calling service functions such as *IoGetDeviceObjectPointer*. Having found your device object, they can send you IRPs.

The other purpose of naming a device object is to allow applications to open handles to the device so they can send you IRPs. An application uses the standard *CreateFile* API to open a handle, whereupon it can use *ReadFile*, *WriteFile*, and *DeviceIoControl* to talk to you. The pathname an application uses to open a device handle begins with the prefix \\.\ rather than with a standard Universal Naming Convention (UNC) name such as C:\MYFILE.CPP or \\FRED\C-Drive\HISFILE.CPP. Internally, the I/O Manager converts this prefix to \??\ before commencing a name search. To provide a mechanism for connecting names in the \?? directory to objects whose names are elsewhere (such as in the \Device directory), the Object Manager implements an object called a symbolic link.

The name \?? has a special meaning in Windows XP. Confronted with this name, the Object Manager first searches a portion of the kernel namespace that is local to the current user session. To see how this works, establish two or more sessions and start DEVVIEW in one of them. Expand the \Sessions folder, and you will eventually see folders for each user. Figure 2-18, which appears later in this chapter, provides an example. If the local search isn't successful, the Object Manager then searches the \GLOBAL?? folder.

IPR2024-00292
KASPERSKY EX 1008 Page 49

Symbolic Links

A symbolic link is a little bit like a desktop shortcut in that it points to some other entity that's the real object of attention. One use of symbolic links in Windows XP is to connect the leading portion of MS-DOS-style names to devices. Figure 2-17 shows a portion of the \GLOBAL?? directory, which includes a number of symbolic links. Notice, for example, that C and other drive letters in the MS-DOS file-naming scheme are actually links to objects whose names are in the \Device directory. These links allow the Object Manager to jump somewhere else in the namespace as it parses through a name. So if I call *CreateFile* with the name C:\MYFILE.CPP, the Object Manager will take this path to open the file:

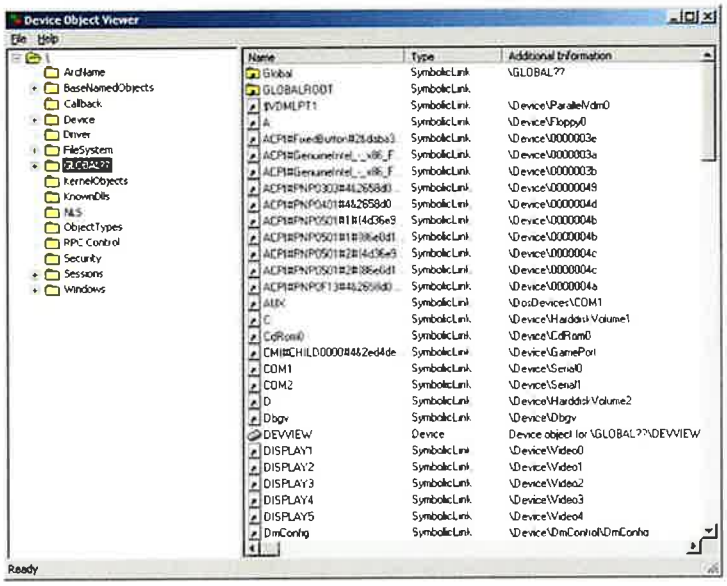


Figure 2-17. *The \GLOBAL?? directory with several symbolic links.*

1. Kernel-mode code initially sees the name \??\C:\MYFILE.CPP. The Object Manager special-cases the "??" name to mean the DosDevices directory for the current session. (In Figure 2-18, this directory is one of the subdirectories of \Sessions\0\DosDevices.)

2. The Object Manager doesn't find "C:" in the session DosDevices directory, so it follows a symbolic link named "Global" to the "GLOBAL??" directory.

3. The Object Manager now looks up "C:" in the \GLOBAL?? directory. It finds a symbolic link by that name, so it forms the new kernel-mode pathname \Device\HarddiskVolume1\MYFILE.CPP and parses that.

4. Working with the new pathname, the Object Manager looks up "Device" in the root directory and finds a directory object.

5. The Object Manager looks up "HarddiskVolume1" in the \Device directory. It finds a device object by that name.

Opening a Disk File

The overall process that occurs when an application opens a disk file is complicated almost beyond belief. To continue the example in the text, the driver for HarddiskVolume1 would be a file system driver such as NTFS.SYS, FASTFAT.SYS, or CDFS.SYS. How a file system realizes that a particular disk volume belongs to it and initializes to handle the volume is itself a saga of Norse proportions. That would have already happened before an application could get far enough to call *CreateFile* with a particular volume letter in the pathname, though, so we can ignore that process.

The file system driver will locate the topmost device object in the storage stack that includes the physical disk drive on which the C volume happens to be mounted. The I/O Manager and file system driver share management of a Volume Parameters Block (VPB) that ties the storage stack and the file system's volume stack together. In principle, the file system driver sends IRPs to the storage driver in order to read directory entries that it can search while parsing the pathname specified by the original *CreateFile* call. In practice, the file system calls the kernel cache manager, which fulfills requests from an in-memory cache if possible and makes recursive calls to the file system driver to fill cache buffers. Deadlock prevention and surprise dismount handling during this process require heroic efforts, including a mechanism whereby a file system driver can stash a pointer to an automatic variable (that is, one allocated on the call/return stack) to be used in deeper layers of recursion within the same thread.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 50

> Luckily, you needn't worry about VPBs and other complications arising from the way file system drivers work in any driver besides one for a storage device. I won't say any more about this in the book.

At this point in the process, the Object Manager will create an IRP that it will send to the driver or drivers for HarddiskVolume1. The IRP will eventually cause some file system driver or another to locate and open a disk file. Describing how a file system driver works is beyond the scope of this book, but the sidebar "Opening a Disk File" will give you a bit of the flavor.

If we were dealing with a device name such as COM1, the driver that ended up receiving the IRP would be the driver for \Device\Serial0. How a device driver handles an open request is definitely within the scope of this book, and I'll be discussing it in this chapter (in the section "Should I Name My Device Object?") and in Chapter 5, when I'll talk about IRP processing in general.

A user-mode program can create a symbolic link in the local (session) namespace by calling DefineDosDevice, as in this example (see Figure 2-18):

```
BOOL okay = DefineDosDevice(DDD RAW TARGET PATH,
  "barf", "\\Device\\Beep");
```

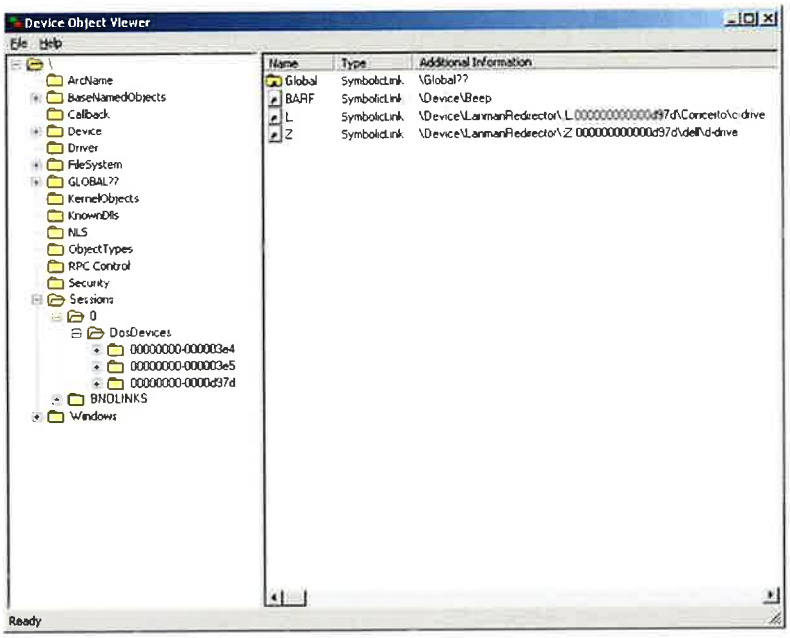


Figure 2-18. *Symbolic link created by DefineDosDevice*

You can create a symbolic link in a WDM driver by calling *IoCreateSymbolicLink*,

```
IoCreateSymbolicLink(linkname, targname);
```

where *linkname* is the name of the symbolic link you want to create and *targname* is the name to which you're linking. Incidentally, the Object Manager doesn't care whether *targname* is the name of any existing object: someone who tries to access an object by using a link that points to an undefined name simply receives an error. If you want to allow user-mode programs to override your link and point it somewhere else, you should call *IoCreateUnprotectedSymbolicLink* instead.

The kernel-mode equivalent of the immediately preceding *DefineDosDevice* call is this:

```
UNICODE STRING linkname;
UNICODE STRING targname;
RtlInitUnicodeString(&linkname, L"\\DosDevices\\barf");
RtlInitUnicodeString(&targname, L"\\Device\\Beep");
IoCreateSymbolicLink(&linkname, &targname);
```


IPR2024-00292
KASPERSKY EX 1008 Page 51

Should I Name My Device Object?



Deciding whether to give your device object a name requires, as I said earlier, a little thought. If you give your object a name, it will be possible for any kernel-mode program to try to open a handle to your device. Furthermore, it will be possible for any kernel-mode or user-mode program to create a symbolic link to your device object and to use the symbolic link to try to open a handle. You might or might not want to allow these actions.

The primary consideration in deciding whether to name your device object is security. When someone opens a handle to a named object, the Object Manager verifies that they have permission to do so. When *IoCreateDevice* creates a device object for you, it assigns a default security descriptor based on the device type you specify as the fourth argument. The I/O Manager uses three basic categories to select a security descriptor:

- Most file system device objects (that is, disk, CD-ROM, file, and tape) receive the "public default unrestricted" access control list (ACL). This list gives just *SYNCHRONIZE*, *READ_CONTROL*, *FILE_READ_ATTRIBUTES*, and *FILE_TRAVERSE* access to everyone except the System account and all administrators. File system device objects, by the way, exist only so that there can be a target for a *CreateFile* call that will open a handle to a file managed by the file system.
- Disk devices and network file system objects receive the same ACL as the file system objects, with some modifications. For example, everyone gets full access to a named floppy disk device object, and administrators get sufficient rights to run ScanDisk. (User-mode network provider DLLs need greater access to the device object for their corresponding file system driver, which is why network file systems are treated differently from other file systems.)
- All other device objects receive the public open unrestricted ACL, which allows anyone with a handle to the device to do pretty much anything.

You can see that anyone will be able to access a nondisk device for both reading and writing if the driver gives the device object a name at the time it calls *IoCreateDevice*. This is because the default security allows nearly full access and because no security check at all is associated with creating a symbolic link—the security checks happen at open time, based on the named object's security descriptor. This is true even if other device objects in the same stack have more restrictive security.

NOTE
IoCreateDeviceSecure, a function in the .NET DDK, allows you to specify a nondefault security descriptor in situations in which no override is in the registry. This function is too new for us to describe it more fully here.

DEVVIEW will show you the security attributes of the device objects it displays. You can see the operation of the default rules I just described by examining a file system, a disk device, and any other random device.

The PDO also receives a default security descriptor, but it's possible to override it with a security descriptor stored in the hardware key or in the Properties subkey of the class key. (The hardware key has precedence if both keys specify a descriptor.) Even lacking a specific security override, if either the hardware key or the class key's Properties subkey overrides the hardware type or characteristics specification, the I/O Manager constructs a new default security descriptor based on the new type. The I/O Manager does not, however, override the security setting for any of the other device objects above the PDO. Consequently, for the overrides (and the administrative actions that set them up) to have any effect, you shouldn't name your device object. Don't despair though—applications can still access your device by means of a registered interface, which I'll discuss soon.



You need to know about one last security concern. As the Object Manager parses its way through an object name, it needs only *FILE_TRAVERSE* access to the intermediate components of the name. It performs a full security check only on the object named by the final component. So suppose you have a device object reachable under the name \Device\Beep or by the symbolic link \??\Barf. A user-mode application that tries to open \\.\Barf for writing will be blocked if the object security has been set up to deny write access. But if the application tries to open a name like \\.\Barf\ExtraStuff that has additional name qualifications, the open request will make it all the way to the device driver (in the form of an *IRP_MJ_CREATE* I/O request) if the user merely has FILE_TRAVERSE permission, which is routinely granted. (In fact, most systems even run with the option to check for traverse permission turned off.) The I/O Manager expects the device driver to deal with the additional name components and to perform any required security checks with regard to them.

To avoid the security concern I just described, you can supply the flag *FILE_DEVICE_SECURE_OPEN* in the device characteristics argument to *IoCreateDevice*. This flag causes Windows XP to verify that someone has the right to open a handle to a device even if additional name components are present.

The Device Name

If you decide to name the device object, you'll normally put the name in the \Device branch of the namespace. To give it a name, you have to create a *UNICODE_STRING* structure to hold the name, and you have to specify that string as an argument to *IoCreateDevice*:

```
UNICODE_STRING devname;
RtlInitUnicodeString(&devname, L"\\Device\\Simple0");
IoCreateDevice(DriverObject, sizeof(DEVICE_EXTENSION), &devname,
...);
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 52

I'll discuss the use of *RtlInitUnicodeString* in the next chapter.

NOTE
Starting in Windows XP, device object names are case insensitive. In Windows 98/Me and in Windows 2000, they are case sensitive. Be sure to spell *\Device* exactly as shown if you want your driver to be portable across all the systems. Note also the spelling of *\DosDevices*, particularly if your mother tongue doesn't inflect the plural form of nouns!

Conventionally, drivers assign their device objects a name by concatenating a string naming their device type ("Simple" in this code fragment) with a 0-based integer denoting an instance of that type. In general, you don't want to hard-code a name as I just did—you want to compose it dynamically using string-manipulation functions like the following:

```
UNICODE_STRING devname;
static LONG lastindex = -1;
LONG devindex = InterlockedIncrement(&lastindex);
WCHAR name[32];
 snwprintf(name, arraysize(name), L"\\Device\\SIMPLE%2.2d",devindex);
RtlInitUnicodeString(&devname, name);
IoCreateDevice(...);
```

I'll explain the various service functions used in this code fragment in the next couple of chapters. The instance number you derive for private device types might as well be a static variable, as shown in the code fragment.

Notes on Device Naming
The \GLOBAL?? directory used to be named \DosDevices. The change was made to move the often-searched directory of user-mode names to the front of the alphabetical list of directories. Windows 98/Me doesn't recognize the name \?? or \GLOBAL??.

Windows 2000 defines a symbolic link named \DosDevices that points to the \?? directory. Windows XP treats \DosDevices differently depending on the process context at the time you create an object. If you create an object, such as a symbolic link, in a system thread, \DosDevices refers to \GLOBAL??, and you end up with a global name. If you create an object in a user thread, \DosDevices refers to \??, and you end up with a session-specific name. In most situations, a device driver creates symbolic links in its *AddDevice* function, which runs in a system thread, and so ends up with globally named objects in all WDM environments simply by putting the symbolic link in \DosDevices. If you create a symbolic link at another time, you should use \GLOBAL?? in Windows XP and \DosDevices in earlier systems. See Appendix A for a discussion of how to distinguish between WDM platforms.

A quick-and-dirty shortcut for testing is to name your device object in the \DosDevices directory, as many of the sample drivers in the companion content do. A production driver should name the device object in \Device, however, to avoid the possibility of creating an object that ought to be global in a session-private namespace.

In previous versions of Windows NT, drivers for certain classes of devices (notably disks, tapes, serial ports, and parallel ports) called *IoGetConfigurationInformation* to obtain a pointer to a global table containing counts of devices in each of these special classes. A driver would use the current value of the counter to compose a name like Harddisk0, Tape1, and so on and would also increment the counter. WDM drivers don't need to use this service function or the table it returns, however. Constructing names for the devices in these classes is now the responsibility of a Microsoft type-specific class driver (such as DISK.SYS).

Device Interfaces

The older method of naming I just discussed—naming your device object and creating a symbolic link name that applications can use—has two major problems. We've already discussed the security implications of giving your device object a name. In addition, the author of an application that wants to access your device has to know the scheme you adopted to name your devices. If you're the only one writing the applications that will be accessing your hardware, that's not much of a problem. But if many different companies will be writing applications for your hardware, and especially if many hardware companies are making similar devices, devising a suitable naming scheme is difficult.

To solve these problems, WDM introduces a new naming scheme for devices that is language-neutral, easily extensible, usable in an environment with many hardware and software vendors, and easily documented. The scheme relies on the concept of a *device interface*, which is basically a specification for how software can access hardware. A device interface is uniquely identified by a 128-bit GUID. You can generate GUIDs by running the Platform SDK utilities UUIDGEN or GUIDGEN—both utilities generate the same kind of number, but they output the result in different formats. The idea is that some industry group gets together to define a standard way of accessing a certain kind of hardware. As part of the standard-making process, someone runs GUIDGEN and publishes the resulting GUID as the identifier that will be forever after associated with that interface standard.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 53

More About GUIDs
The GUIDs used to identify software interfaces are the same kind of unique identifier that's used in the Component Object Model (COM) to identify COM interfaces and in the Open Software Foundation (OSF) Distributed Computing Environment (DCE) to identify the target of a remote procedure call (RPC). For an explanation of how GUIDs are generated so as to be statistically unique, see page 66 of Kraig Brockschmidt's *Inside OLE*, Second Edition (Microsoft Press, 1995), which contains a further reference to the original algorithm specification by the OSF. I found the relevant portion of the OSF specification on line at http://www.opengroup.org/onlinepubs/9629399/apdxa.htm.

The mechanics of creating a GUID for use in a device driver involve running either UUIDGEN or GUIDGEN and then capturing the resulting identifier in a header file. GUIDGEN is easier to use because it allows you to choose to format the GUID for use with the *DEFINE_GUID* macro and to copy the resulting string onto the Clipboard. Figure 2-19 shows the GUIDGEN window. You can paste its output into a header file to end up with this:

```
// {CAF53C68-A94C-11d2-BB4A-00C04FA330A6}
DEFINE_GUID(<<name>>,
  0xcaf53c68, 0xa94c, 0x11d2, 0xbb, 0x4a, 0x0, 0xc0, 0x4f,
  0xa3, 0x30, 0xa6);
```

You then replace *<<name>>* with something more mnemonic like *GUID_DEVINTERFACE_SIMPLE* and include the definition in your driver and applications.

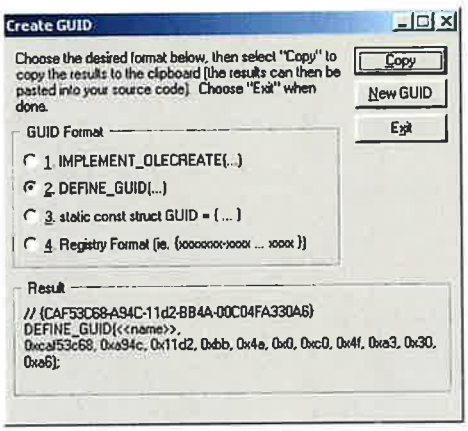


Figure 2-19. *Using GUIDGEN to generate a GUID*

I think of an interface as being analogous to the protein markers that populate the surface of living cells. An application desiring to access a particular kind of device has its own protein markers that fit like a key into the markers exhibited by conforming device drivers. See Figure 2-20.

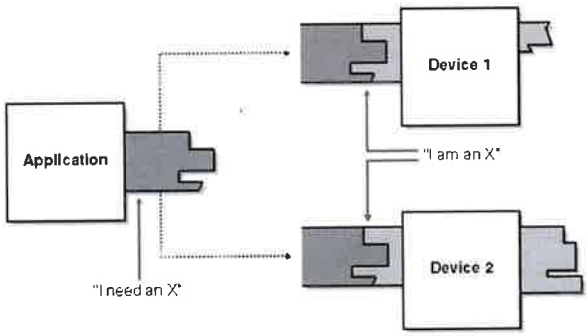


Figure 2-20. *Using device interfaces to match applications and devices*

Registering a Device Interface

A function driver's *AddDevice* function should register one or more device interfaces by calling *IoRegisterDeviceInterface*, as shown here:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 54

```
1
#include <initguid.h>
2
#include "guids.h"
...
NTSTATUS AddDevice(...)
  {
...
3
  IoRegisterDeviceInterface(pdo, &GUID DEVINTERFACE SIMPLE,NULL, &pdx->ifname);
...
  }
```

1. We're about to include a header (GUIDS.H) that contains one or more *DEFINE_GUID* macros. *DEFINE_GUID* normally declares an external variable. Somewhere in the driver, though, we have to actually reserve initialized storage for every GUID we're going to reference. The system header file INITGUID.H works some preprocessor magic to make *DEFINE_GUID* reserve the storage even if the definition of the *DEFINE_GUID* macro happens to be in one of the precompiled header files.

2. I'm assuming here that I put the GUID definitions I want to reference into a separate header file. This would be a good idea, inasmuch as user-mode code will also need to include these definitions and won't want to include a bunch of extraneous kernel-mode declarations relevant only to our driver.

3. The first argument to *IoRegisterDeviceInterface* must be the address of the PDO for your device. The second argument identifies the GUID associated with your interface, and the third argument specifies additional qualified names that further subdivide your interface. Only Microsoft code uses this name subdivision scheme. The last argument is the address of a *UNICODE_STRING* structure that will receive the name of a symbolic link that resolves to this device object.

The return value from *IoRegisterDeviceInterface* is a Unicode string that applications will be able to determine without knowing anything special about how you coded your driver and will then be able to use in opening a handle to the device. The name is pretty ugly, by the way; here's an example that I generated for one of my sample devices: \\?\ROOT#UNKNOWN#0000#{b544b9a2-6995-11d3-81b5-00c04fa330a6}.

All that registration actually does is create the symbolic link name and save it in the registry. Later on, in response to the *IRP_MN_START_DEVICE* Plug and Play request we'll discuss in Chapter 7, you'll make the following call to *IoSetDeviceInterfaceState* to enable the interface:

```
IoSetDeviceInterfaceState(&pdx->ifname, TRUE);
```

In response to this call, the I/O Manager creates an actual symbolic link object pointing to the PDO for your device. You'll make a matching call to disable the interface at a still later time (just call *IoSetDeviceInterfaceState* with a FALSE argument), whereupon the I/O Manager will delete the symbolic link object while preserving the registry entry that contains the name. In other words, the name persists and will always be associated with this particular instance of your device; the symbolic link object comes and goes with the hardware.

Since the interface name ends up pointing to the PDO, the PDO's security descriptor ends up controlling whether people can access your device. That's good because it's the PDO's security that you control in the INF used to install the driver.

Enumerating Device Interfaces

Both kernel-mode and user-mode code can locate all the devices that happen to support an interface in which they're interested. I'm going to explain how to enumerate all the devices for a particular interface in user mode. The enumeration code is so tedious to write that I eventually wrote a C++ class to make my own life simpler. You'll find this code in the DEVICELIST.CPP and DEVICELIST.H files that are part of the HIDFAKE and DEVPROP samples in Chapter 8. These files declare and implement a *CDeviceList* class, which contains an array of *CDeviceListEntry* objects. These two classes have the following declaration:

```
class CDeviceListEntry
{
public:
  CDeviceListEntry(LPCTSTR linkname, LPCTSTR friendlyname);
  CDeviceListEntry(){}
  CString m_linkname;
  CString m_friendlyname;
};

class CDeviceList
{
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 55

```
public:
  CDeviceList(const GUID& guid);
  ~CDeviceList();
  GUID m_guid;
  CArray<CDeviceListEntry, CDeviceListEntry&> m_list;
  int Initialize();
};
```

The classes rely on the *CString* class and *CArray* template class that are part of the Microsoft Foundation Classes (MFC) framework. The constructors for these two classes simply copy their arguments into the obvious data members:

```
CDeviceList::CDeviceList(const GUID& guid)
  {
  m_guid = guid;
  }
CDeviceListEntry::CDeviceListEntry(LPCTSTR linkname,LPCTSTR friendlyname)
  {
  m_linkname = linkname;
  m_friendlyname = friendlyname;
  }
```

All the interesting work occurs in the *CDeviceList::Initialize* function. The executive overview of what it does is this: it will enumerate all of the devices that expose the interface whose GUID was supplied to the constructor. For each such device, it will determine a friendly name that we're willing to show to an unsuspecting end user. Finally it will return the number of devices it found. Here's the code for this function:

```
int CDeviceList::Initialize()
  {
1
  HDEVINFO info = SetupDiGetClassDevs(&m_guid, NULL, NULL,
    DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
  if (info == INVALID_HANDLE_VALUE)
    return 0;
  SP_INTERFACE_DEVICE_DATA ifdata;
  ifdata.cbSize = sizeof(ifdata);
  DWORD devindex;
2
  for (devindex = 0;
    SetupDiEnumDeviceInterfaces(info,NULL, &m_guid, devindex, &ifdata);
    ++devindex)
    {
    DWORD needed;
3
    SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, &needed, NULL);

    PSP_INTERFACE_DEVICE_DETAIL_DATA detail =
      (PSP_INTERFACE_DEVICE_DETAIL_DATA) malloc(needed);
    detail->cbSize = sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
    SP_DEVINFO_DATA did = {sizeof(SP_DEVINFO_DATA)};
    SetupDiGetDeviceInterfaceDetail(info, &ifdata, detail, needed, NULL, &did));

4
    TCHAR fname[256];
    if (!SetupDiGetDeviceRegistryProperty(info, &did,
      SPDRP_FRIENDLYNAME, NULL, (PBYTE) fname,
      sizeof(fname), NULL)
      && !SetupDiGetDeviceRegistryProperty(info, &did,
        SPDRP_DEVICEDESC,
        NULL, (PBYTE) fname, sizeof(fname), NULL))
       _tcsncpy(fname, detail->DevicePath, 256);
      fname[255] = 0;

5
    CDeviceListEntry e(detail->DevicePath, fname);
    free((PVOID) detail);

    m_list.Add(e);
    }
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 56

```
    SetupDiDestroyDeviceInfoList(info);
    return m_list.GetSize();
    }
```

1. This statement opens an enumeration handle that we can use to find all devices that have registered an interface that uses the same GUID.
2. Here we call *SetupDiEnumDeviceInterfaces* in a loop to find each device.
3. The only two items of information we need are the detail information about the interface and information about the device instance. The detail is just the symbolic name for the device. Since it's variable in length, we make two calls to *SetupDiGetDeviceInterfaceDetail*. The first call determines the length. The second call retrieves the name.
4. We obtain a friendly name for the device from the registry by asking for either the *FriendlyName* or the *DeviceDesc*.
5. We create a temporary instance named e of the *CDeviceListEntry* class, using the device's symbolic name as both the link name and the friendly name.

NOTE
You might be wondering how the registry comes to have a FriendlyName for a device. The INF file you use to install your device driver—see Chapter 15—can have an HW section that specifies registry parameters for the device. You can provide a FriendlyName as one of these parameters, but bear in mind that every instance of your hardware will have the *same* name if you do. The MAKENAMES sample describes a DLL-based way of defining a unique friendly name for each instance. You can also write a *CoInstaller* DLL that will define unique friendly names.

If you don't define a FriendlyName, by the way, most system components will use the *DeviceDesc* string in the registry. This string originates in the INF file and will usually describe your device by manufacturer and model.

Sample Code
The DEVINTERFACE sample is a user-mode program that enumerates all instances of all known device interface GUIDs on your system. One way to use this sample is as a way to determine which GUID you need to enumerate to find a particular device.

2.5.3 Other Global Device Initialization

You need to take some other steps during *AddDevice* to initialize your device object. I'm going to describe these steps in the order you should do them, which isn't exactly the same order as their respective logical importance. I want to emphasize that the code snippets in this section are even more fragmented than usual—I'm going to show only enough of the entire *AddDevice* routine to establish the surrounding context for the small pieces I'm trying to illustrate.

Initializing the Device Extension

The content and management of the device extension are entirely up to you. The data members you place in this structure will obviously depend on the details of your hardware and on how you go about programming the device. Most drivers would need a few items placed there, however, as illustrated in the following fragment of a declaration:

```
1
typedef struct  DEVICE_EXTENSION {
2
    PDEVICE_OBJECT DeviceObject;
3
    PDEVICE_OBJECT LowerDeviceObject;
4
    PDEVICE_OBJECT Pdo;
5
    UNICODE_STRING ifname;
6
    IO_REMOVE_LOCK RemoveLock;
7
    DEVSTATE devstate;
    DEVSTATE prevstate;
    DEVICE_POWER_STATE devpower;
    SYSTEM_POWER_STATE syspower;
8
    DEVICE_CAPABILITIES devcaps;
    .
    .
    .
```


IPR2024-00292
KASPERSKY EX 1008 Page 57

```
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

1. I find it easiest to mimic the pattern of structure declaration used in the official DDK, so I declared this device extension as a structure with a tag as well as a type and pointer-to-type name.

2. You already know that you locate your device extension by following the *DeviceExtension* pointer from the device object. It's also useful in several situations to be able to go the other way—to find the device object given a pointer to the extension. The reason is that the logical argument to certain functions is the device extension itself (since that's where all of the per-instance information about your device resides). Hence, I find it useful to have this *DeviceObject* pointer.

3. I'll mention in a few paragraphs that you need to record the address of the device object immediately below yours when you call *IoAttachDeviceToDeviceStack*, and *LowerDeviceObject* is the place to do that.

4. A few service routines require the address of the PDO instead of some higher device object in the same stack. It's very difficult to locate the PDO, so the easiest way to satisfy the requirement of those functions is to record the PDO address in a member of the device extension that you initialize during *AddDevice*.

5. Whichever method (symbolic link or device interface) you use to name your device, you'll want an easy way to remember the name you assign. In this code fragment, I've declared a Unicode string member named *ifname* to record a device interface name. If you were going to use a symbolic link name instead of a device interface, it would make sense to give this member a more mnemonic name, such as *linkname*.

6. I'll discuss in Chapter 6 a synchronization problem affecting how you decide when it's safe to remove this device object by calling *IoDeleteDevice*. The solution to that problem involves using an *IO_REMOVE_LOCK* object that needs to be allocated in your device extension as shown here. *AddDevice* needs to initialize that object.

7. You'll probably need a device extension variable to keep track of the current Plug and Play state and current power states of your device. *DEVSTATE* is an enumeration that I'm assuming you've declared elsewhere in your own header file. I'll discuss the use of all these state variables in later chapters.

8. Another part of power management involves remembering some capability settings that the system initializes by means of an IRP. The *devcaps* structure in the device extension is where I save those settings in my sample drivers.

The initialization statements in *AddDevice* (with emphasis on the parts involving the device extension) would be as follows:

```
NTSTATUS AddDevice(...)
   {
   PDEVICE OBJECT fdo;
   IoCreateDevice(..., sizeof(DEVICE EXTENSION), ..., &fdo);
   PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
   pdx->DeviceObject = fdo;
   pdx->Pdo = pdo;
   IoInitializeRemoveLock(&pdx->RemoveLock, ...);
   pdx->devstate = STOPPED;
   pdx->devpower = PowerDeviceD0;
   pdx->syspower = PowerSystemWorking;
   IoRegisterDeviceInterface(..., &pdx->ifname);
   pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(...);
   }
```

In this code snippet, *STOPPED* and *DEVICE_EXTENSION* are things I defined in one of my own header files.

Initializing the Default DPC Object

Many devices signal completion of operations by means of an interrupt. As you'll learn when I discuss interrupt handling in Chapter 7, there are strict limits on what your interrupt service routine (ISR) can do. In particular, an ISR isn't allowed to call the routine (*IoCompleteRequest*) that signals completion of an IRP, but that's exactly one of the steps you're likely to want to take. You utilize *a deferred procedure call* (DPC) to get around the limitations. Your device object contains a subsidiary DPC object that can be used for scheduling your particular DPC routine, and you need to initialize it shortly after creating the device object:

```
NTSTATUS AddDevice(...)
   {
   IoCreateDevice(...);
   IoInitializeDpcRequest(fdo, DpcForIsr)
   }
```

Setting the Buffer Alignment Mask

Devices that perform DMA transfers work directly with data buffers in memory. The HAL might require that buffers used for

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 58

DMA be aligned to some particular boundary, and your device might require still more stringent alignment. The *AlignmentRequirement* field of the device object expresses the restriction—it's a bit mask equal to 1 less than the required address boundary. You can round an arbitrary address down to this boundary with this statement:

```
PVOID address = ...;
SIZE_T ar = fdo->AlignmentRequirement;
address = (PVOID) ((SIZE_T) address & ~ar);
```

You round an arbitrary address up to the next alignment boundary like this:

```
PVOID address = ...;
SIZE_T ar = fdo->AlignmentRequirement;
address = (PVOID) (((SIZE_T) address + ar) & ~ar);
```

In these two code fragments, I used *SIZE_T* casts to transform the pointer (which may be 32 bits or 64 bits wide, depending on the platform for which you're compiling) into an integer wide enough to span the same range as the pointer.

IoCreateDevice sets the *AlignmentRequirement* field of the new device object equal to whatever the HAL requires. For example, the HAL for Intel x86 chips has no alignment requirement, so *AlignmentRequirement* is 0 initially. If your device requires a more stringent alignment for the data buffers it works with (say, because you have bus-mastering DMA capability with a special alignment requirement), you want to override the default setting. For example:

```
if (MYDEVICE_ALIGNMENT - 1 > fdo->AlignmentRequirement)
  fdo->AlignmentRequirement = MYDEVICE_ALIGNMENT - 1;
```

I've assumed here that elsewhere in your driver is a manifest constant named *MYDEVICE_ALIGNMENT* that equals a power of 2 and represents the required alignment of your device's data buffers.

Miscellaneous Objects

Your device might well use other objects that need to be initialized during *AddDevice*. Such objects might include various synchronization objects, linked list anchors, scatter/gather list buffers, and so on. I'll discuss these objects, and the fact that initialization during *AddDevice* would be appropriate, in various other parts of this book.

Initializing the Device Flags

Two of the flag bits in your device object need to be initialized during *AddDevice* and never changed thereafter: the *DO_BUFFERED_IO* and *DO_DIRECT_IO* flags. You can set one (but only one) of these bits to declare once and for all how you want to handle memory buffers coming from user mode as part of read and write requests. (I'll explain in Chapter 7 what the difference between these two buffering methods is and why you'd want to pick one or the other.) The reason you have to make this important choice during *AddDevice* is that any upper filter drivers that load afterwards will be copying your flag settings, and it's the setting of the bits in the topmost device object that's actually important. Were you to change your mind after the filter drivers loaded, they probably wouldn't know about the change.

Two of the flag bits in the device object pertain to power management. In contrast with the two buffering flags, these two can be changed at any time. I'll discuss them in greater detail in Chapter 8, but here's a preview. *DO_POWER_PAGABLE* means that the Power Manager must send you *IRP_MJ_POWER* requests at interrupt request level (IRQL) *PASSIVE_LEVEL*. (If you don't understand all of the concepts in the preceding sentence, don't worry—I'll completely explain all of them in later chapters.) *DO_POWER_INRUSH* means that your device draws a large amount of current when powering on, so the Power Manager should make sure that no other inrush device is powering up simultaneously.

Building the Device Stack

Each filter and function driver has the responsibility of building up the stack of device objects, starting from the PDO and working upward. You accomplish your part of this work with a call to *IoAttachDeviceToDeviceStack*:

```
NTSTATUS AddDevice(..., PDEVICE_OBJECT pdo)
  {
  PDEVICE_OBJECT fdo;
  IoCreateDevice(..., &fdo);
  pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
  }
```

The first argument to *IoAttachDeviceToDeviceStack* (fdo) is the address of your own newly created device object. The second argument is the address of the PDO. The second parameter to AddDevice is this address. The return value is the address of whatever device object is immediately *underneath* yours, which can be the PDO or the address of some lower filter device object. Figure 2-21 illustrates the situation when there are three lower filter drivers for your device. By the time your *AddDevice* function executes, all three of their *AddDevice* functions have already been called. They have created their

IPR2024-00292
KASPERSKY EX 1008 Page 59

respective FiDOs and linked them into the stack rooted at the PDO. When you call *IoAttachDeviceToDeviceStack*, you get back the address of the topmost FiDO.

IoAttachDeviceToDeviceStack might conceivably fail by returning a NULL pointer. For this to occur, someone would have to remove the physical device from the system at just the point in time when your *AddDevice* function was doing its work, and the PnP Manager would have to process the removal on another CPU. I'm not even sure these conditions are enough to trigger a failure. (Or else the driver under you could have forgotten to clear *DO_DEVICE_INITIALIZING*, I suppose.) You would deal with the failure by cleaning up and returning *STATUS_DEVICE_REMOVED* from your *AddDevice* function.

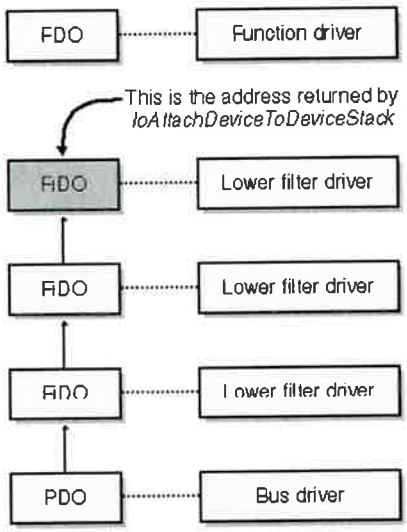


Figure 2-21. *What IoAttachDeviceToDeviceStack returns.*

Clear *DO_DEVICE_INITIALIZING*

Pretty much the last thing you do in *AddDevice* should be to clear the *DO_DEVICE_INITIALIZING* flag in your device object:

```
fdo->Flags &= ~DO_DEVICE_INITIALIZING;
```

While this flag is set, the I/O Manager will refuse to attach other device objects to yours or to open a handle to your device. You have to clear the flag because your device object initially arrives in the world with the flag set. In previous releases of Windows NT, most drivers created all of their device objects during *DriverEntry*. When *DriverEntry* returns, the I/O Manager automatically traverses the list of device objects linked from the driver object and clears this flag. Since you're creating your device object long after *DriverEntry* returns, however, this automatic flag clearing won't occur, and you must do it yourself.

2.5.4 Putting the Pieces Together

Here is a complete *AddDevice* function, presented without error checking or annotations and including all the pieces described in the preceding sections:

```
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
  PDEVICE OBJECT fdo;
  NTSTATUS status = IoCreateDevice(DriverObject,
    sizeof(DEVICE EXTENSION), NULL,
    FILE DEVICE UNKNOWN, FILE DEVICE SECURE OPEN, FALSE, &fdo);

  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION)
    fdo->DeviceExtension;

  IoRegisterDeviceInterface(pdo, &GUID DEVINTERFACE SIMPLE, NULL, &pdx->ifname);

  pdx->DeviceObject = fdo;
  pdx->Pdo = pdo;
  IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 0);
  pdx->devstate = STOPPED;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 60

```
pdx->devpower = PowerDeviceD0;
pdx->syspower = PowerSystemWorking;

IoInitializeDpcRequest(fdo, DpcForIsr);

if (MYDEVICE_ALIGNMENT - 1 > fdo->AlignmentRequirement)
  fdo->AlignmentRequirement = MYDEVICE_ALIGNMENT - 1;

KeInitializeSpinLock(&pdx->SomeSpinLock);
KeInitializeEvent(&pdx->SomeEvent, NotificationEvent, FALSE);
InitializeListHead(&pdx->SomeListAnchor);

fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;

pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);

fdo->Flags &= ~DO_DEVICE_INITIALIZING;
return STATUS_SUCCESS;
}
```

2.6 Windows 98/Me Compatibility Notes

Windows 98/Me handles some of the details surrounding device object creation and driver loading differently than Windows XP. This section explains the differences that might affect your driver. I've already mentioned a few of these, but repetition can't hurt.

2.6.1 Differences in *DriverEntry* Call

As I indicated earlier, the *DriverEntry* routine receives a UNICODE_STRING argument naming the service key for the driver. In Windows XP, the string is a full registry path of the form \Registry\Machine\System\CurrentControlSet\Services\xxx (where xxx is the name of the service entry for your driver). In Windows 98/Me, however, the string is of the form System\CurrentControlSet\Services\<classname >\<instance#> (where <classname> is the class name of your device and <instance#> is an instance number such as 0000 indicating which device of that class you happen to be). You can open the key in either environment by calling *ZwOpenKey*, however.

2.6.2 *DriverUnload*

Windows 98/Me will call *DriverUnload* within a call to *IoDeleteDevice* that occurs within *DriverEntry*. You care about this only if (1) your *DriverEntry* function calls *IoCreateDevice* and then (2) decides to return an error status, whereupon it (3) cleans up by calling *IoDeleteDevice*.

2.6.3 The \GLOBAL?? Directory

Windows 98/Me doesn't understand the directory name \GLOBAL??. Consequently, you need to put symbolic link names in the \DosDevices directory. You can use \DosDevices in Windows XP also because it's a symbolic link to the \?? directory, whose (virtual) contents include \GLOBAL??.

2.6.4 Unimplemented Device Types

Windows 98 didn't support creating device objects for mass storage devices. These are devices with types *FILE_DEVICE_DISK*, *FILE_DEVICE_TAPE*, *FILE_DEVICE_CD_ROM*, and *FILE_DEVICE_VIRTUAL_DISK*. You can call *IoCreateDevice*, and it will even return with a status code of *STATUS_SUCCESS*, but it won't have actually created a device object or modified the PDEVICE_OBJECT variable whose address you gave as the last argument.

IPR2024-00292
KASPERSKY EX 1008 Page 61

Chapter 3

Basic Programming Techniques

Writing a WDM driver is fundamentally an exercise in software engineering. Whatever the requirements of your particular hardware, you will combine various elements to form a program. In the preceding chapter, I described the basic structure of a WDM driver, and I showed you two of its elements—*DriverEntry* and *AddDevice*—in detail. In this chapter, I'll focus on the even more basic topic of how you call upon the large body of kernel-mode support routines that the operating system exposes for your use. I'll discuss error handling, memory and data structure management, registry and file access, and a few other topics. I'll round out the chapter with a short discussion of the steps you can take to help debug your driver.

3.1 The Kernel-Mode Programming Environment

Figure 3-1 illustrates some of the components that make up the Microsoft Windows XP operating system. Each component exports service functions whose names begin with a particular two-letter or three-letter prefix:

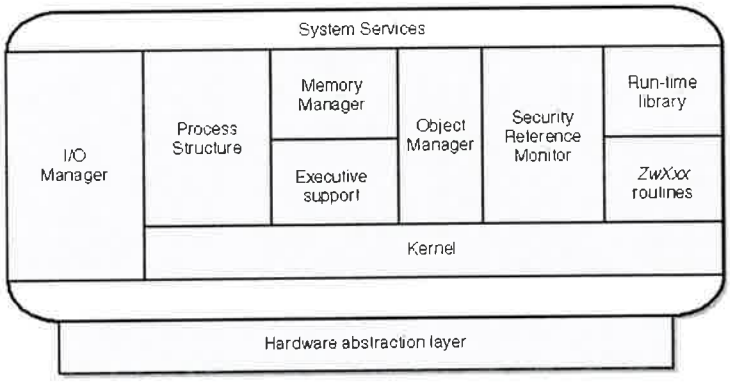


Figure 3-1. *Overview of kernel-mode support routines*

- The I/O Manager (prefix *Io*) contains many service functions that drivers use, and I'll be discussing them all throughout this book.
- The Process Structure module (prefix *Ps*) creates and manages kernel-mode threads. An ordinary WDM driver might use an independent thread to repeatedly poll a device incapable of generating interrupts, and for other purposes.
- The Memory Manager (prefix *Mm*) controls the page tables that define the mapping of virtual addresses onto physical memory.
- The executive (prefix *Ex*) supplies heap management and synchronization services. I'll discuss the heap management service functions in this chapter. The next chapter covers the synchronization services.
- The Object Manager (prefix *Ob*) provides centralized control over the many data objects with which Windows XP works. WDM drivers rely on the Object Manager for keeping a reference count that prevents an object from disappearing while someone is still using it and to convert object handles to pointers to the objects the handles represent.
- The Security Reference Monitor (prefix *Se*) allows file system drivers to perform security checks. Someone else has usually dealt with security concerns by the time an I/O request reaches a WDM driver, so I won't be discussing these functions in this book.
- The so-called run-time library component (prefix *Rtl*) contains utility routines, such as list and string-management routines, that kernel-mode drivers can use instead of regular ANSI-standard library routines. For the most part, the operation of these functions is obvious from their names, and you would pretty much know how to use them in a program if you just were aware of them. I'll describe a few of them in this chapter.
- Windows XP implements the native API for kernel-mode callers using routine names that begin with the prefix Zw. The DDK documents just a few of the *ZwXxx* functions, namely the ones that pertain to registry and file access. I'll discuss those functions in this chapter.

IPR2024-00292
KASPERSKY EX 1008 Page 63

- The Windows XP kernel (prefix *Ke*) is where all the low-level synchronization of activities between threads and processors occurs. I'll discuss the *KeXxx* functions in the next chapter.
- The very bottom layer of the operating system, on which the support sandwich rests, is the hardware abstraction layer (or HAL, prefix *Hal*). All the operating system's knowledge of how the computer is actually wired together reposes in the HAL. The HAL understands how interrupts work on a particular platform, how to address I/O and memory-mapped devices, and so on. Instead of talking directly to their hardware, WDM drivers call functions in the HAL to do it. The driver ends up being platform-independent and bus-independent.

3.1.1 Using Standard Run-Time Library Functions

Historically, the Windows NT architects preferred that drivers not use the run-time libraries supplied by vendors of C compilers. In part, the initial disapproval arose from simple timing. Windows NT was designed at a time when there was no ANSI standard for what functions belonged in a standard library and when many compiler vendors existed, each with its own idea of what might be cool to include and its own unique quality standards. Another factor is that standard run-time library routines sometimes rely on initialization that can happen only in a user-mode application and are sometimes implemented in a thread-unsafe or multiprocessor-unsafe way.



I suggested in the first edition that it would be OK to use a number of "standard" runtime library functions for string processing. That was probably bad advice, though, because most of us (including me!) have a hard time using them safely. It's true (at least at the time I'm writing this paragraph) that the kernel exports standard string functions such as *strcpy*, *wcscmp*, and *strncpy*. Since these functions work with null-terminated strings, though, it's just too easy to make mistakes with them. Are you sure you've provided a large enough target buffer for *strcpy*? Are you sure that both of the strings you're comparing with *wcscmp* have a null terminator before they tail off into a not-present page? Were you aware that *strncpy* can fail to null-terminate the target string if the source string is longer than the target?

Because of all the potential problems with run-time library functions, Microsoft now recommends using the set of "safe" string functions declared in NtStrsafe.h. I'll discuss these functions, and the very few standard string and byte functions that it's safe to use in a driver, later in this chapter.

3.1.2 A Caution About Side Effects

Many of the support "functions" that you use in a driver are defined as macros in the DDK header files. We were all taught to avoid using expressions that have side effects (that is, expressions that alter the state of the computer in some persistent way) as arguments to macros for the obvious reason that the macro can invoke the argument more or less than exactly once. Consider, for example, the following code:

```
int a = 2, b = 42, c;
c = min(a++, b);
```

What's the value of a afterward? (For that matter, what's the value of c?) Take a look at a plausible implementation of min as a macro:

```
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

If you substitute a++ for x, you can see that a will equal 4 because the expression a++ gets executed twice. The value of the "function" *min* will be 3 instead of the expected 2 because the second invocation of a++ delivers the value.

You basically can't tell when the DDK will use a macro and when it will declare a real external function. Sometimes a particular service function will be a macro for some platforms and a function call for other platforms. Furthermore, Microsoft is free to change its mind in the future. Consequently, you should follow this rule when programming a WDM driver:

Never use an expression that has side effects as an argument to a kernel-mode service function.

3.2 Error Handling

To err is human; to recover is part of software engineering. Exceptional conditions are always arising in programs. Some of them start with program bugs, either in our own code or in the user-mode applications that invoke our code. Some of them relate to system load or the instantaneous state of hardware. Whatever the cause, unusual circumstances demand a flexible response from our code. In this section, I'll describe three aspects of error handling: status codes, structured exception handling, and bug checks. In general, kernel-mode support routines report unexpected errors by returning a status code, whereas they report expected variations in normal flow by returning a Boolean or numeric value other than a formal status code. Structured exception handling offers a standardized way to clean up after really unexpected events, such as dereferencing an invalid user-mode pointer, or to avoid the system crash that normally ensues after such events. A bug check is the internal name for a catastrophic failure for which a system shutdown is the only cure.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 64

3.2.1 Status Codes

Kernel-mode support routines (and your code too, for that matter) indicate success or failure by returning a status code to their caller. An *NTSTATUS* value is a 32-bit integer composed of several subfields, as illustrated in Figure 3-2. The high-order 2 bits denote the severity of the condition being reported—success, information, warning, or error. I'll explain the impact of the customer flag shortly. The facility code indicates which system component originated the message and basically serves to decouple development groups from each other when it comes to assigning numbers to codes. The remainder of the status code—16 bits' worth—indicates the exact condition being reported.

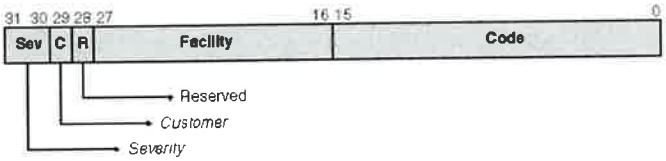


Figure 3-2. *Format of an NTSTATUS code.*

You should always check the status returns from routines that provide them. I'm going to break this rule frequently in some of the code fragments I show you because including all the necessary error handling code often obscures the expository purpose of the fragment. But don't you emulate this sloppy practice!

If the high-order bit of a status code is 0, any number of the remaining bits could be set and the code would still indicate success. Consequently, never just compare status codes with 0 to see whether you're dealing with success—instead, use the *NT_SUCCESS* macro:

```
NTSTATUS status = SomeFunction(...);
if (!NT SUCCESS(status))
  {
. //handle error
:
:
  }
```

Not only do you want to test the status codes you receive from routines you call, but you also want to return status codes to the routines that call you. In the preceding chapter, I dealt with two driver subroutines—*DriverEntry* and *AddDevice*—that are both defined as returning *NTSTATUS* codes. As I discussed, you want to return *STATUS_SUCCESS* as the success indicator from these routines. If something goes wrong, you often want to return an appropriate status code, which is sometimes the same value that a routine returned to you.

As an example, here are some initial steps in the *AddDevice* function, with all the error checking left in:

```
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
  NTSTATUS status;
  PDEVICE OBJECT fdo;
  status = IoCreateDevice(DriverObject, sizeof(DEVICE EXTENSION),
    NULL, FILE DEVICE UNKNOWN, FILE DEVICE SECURE OPEN, FALSE,
    &fdo);
1
  if (!NT SUCCESS(status))
    {
2
    KdPrint(("IoCreateDevice failed - %X\n", status));
    return status;
    }
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  pdx->DeviceObject = fdo;
  pdx->Pdo = pdo;
  pdx->state = STOPPED;
3
  IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 0);
4
  status = IoRegisterDeviceInterface(pdo, &GUID SIMPLE, NULL,
    &pdx->ifname);
  if (!NT SUCCESS(status))
    {
    KdPrint(("IoRegisterDeviceInterface failed - %X\n", status));
    IoDeleteDevice(fdo);
```


IPR2024-00292
KASPERSKY EX 1008 Page 65

```
    return status;
    }
:
:
}
```

1. If *IoCreateDevice* fails, we'll simply return the same status code it gave us. Note the use of the NT_SUCCESS macro as described in the text.

2. It's sometimes a good idea, especially while debugging a driver, to print any error status you discover. I'll discuss the exact usage of *KdPrint* later in this chapter (in the "Making Debugging Easier" section).

3. *IoInitializeRemoveLock*, discussed in Chapter 6, cannot fail. Consequently, there's no need to check a status code. Generally speaking, most functions declared with type VOID are in the same "cannot fail" category. A few VOID functions can fail by raising an exception, but the DDK documents that behavior very clearly.

4. Should *IoRegisterDeviceInterface* fail, we have some cleanup to do before we return to our caller; namely, we must call *IoDeleteDevice* to destroy the device object we just created.

You don't always have to fail calls that lead to errors in the routines you call, of course. Sometimes you can ignore an error. For example, in Chapter 8, I'll tell you about a power management I/O request with the subtype *IRP_MN_POWER_SEQUENCE* that you can use as an optimization to avoid unnecessary state restoration during a power-up operation. Not only is it optional whether you use this request, but it's also optional for the bus driver to implement it. Therefore, if that request should fail, you should just go about your business. Similarly, you can ignore an error from *IoAllocateErrorLogEntry* because the inability to add an entry to the error log isn't at all critical.

Completing an IRP with an error status—driver programmers call this failing the IRP—usually leads to a failure indication in the return from a Win32 API function in an application. The application can call *GetLastError* to determine the cause of the failure. If you fail the IRP with a status code containing the customer flag, *GetLastError* will return exactly that status code. If you fail the IRP with a status code in which the customer flag is 0 (which is the case for every standard status code defined by Microsoft), *GetLastError* returns a value drawn from WINERROR.H in the Platform SDK. Knowledge Base article Q113996, "Mapping NT Status Error Codes to Win32 Error Codes," documents the correspondence between *GetLastError* return values and kernel status codes. Table 3-1 shows the correspondence for the most important status codes.

Kernel-Mode Status Code	User-Mode Error Code
STATUS_SUCCESS	NO_ERROR (0)
STATUS_INVALID_PARAMETER	ERROR_INVALID_PARAMETER
STATUS_NO_SUCH_FILE	ERROR_FILE_NOT_FOUND
STATUS_ACCESS_DENIED	ERROR_ACCESS_DENIED
STATUS_INVALID_DEVICE_REQUEST	ERROR_INVALID_FUNCTION
ERROR_BUFFER_TOO_SMALL	ERROR_INSUFFICIENT_BUFFER
STATUS_DATA_ERROR	ERROR_CRC

Table 3-1. *Correspondence Between Common Kernel-Mode and User-Mode Status Codes*

The difference between an error and a warning can be significant. For example, failing a *METHOD_BUFFERED* control operation (see Chapter 9) with *STATUS_BUFFER_OVERFLOW*—a warning—causes the I/O Manager to copy data to the user-mode buffer. Failing the same operation with *STATUS_BUFFER_TOO_SMALL*—an error—causes the I/O Manager to not copy any data.

3.2.2 Structured Exception Handling

The Windows family of operating systems provides a method of handling exceptional conditions that helps you avoid potential system crashes. Closely integrated with the compiler's code generator, structured exception handling lets you easily place a guard on sections of your code and invoke exception handlers when something goes wrong in the guarded section. Structured exception handling also lets you easily provide cleanup statements that you can be sure will always execute no matter how control leaves a guarded section of code.

Very few of my seminar students have been familiar with structured exceptions, so I'm going to explain some of the basics here. You can write better, more bulletproof code if you use these facilities. In many situations, the parameters that you receive in a WDM driver have been thoroughly vetted by other code and won't cause you to generate inadvertent exceptions. Good taste may, therefore, be the only impetus for you to use the stuff I'm describing in this section. As a general rule, though, you always want to protect direct references to user-mode virtual memory with a structured exception frame. Such references occur when you directly reference memory and when you call *MmProbeAndLockPages*, *ProbeForRead*, and *ProbeForWrite*, and perhaps at other times.

Sample Code
The SEHTEST sample driver illustrates the mechanics of structured exceptions in a WDM driver.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 66

Which Exceptions Can Be Trapped
Gary Nebbett researched the question of which exceptions can be trapped with the structured exception mechanism and reported his results in a newsgroup post several years ago. The SEHTEST sample incorporates what he learned. In summary, the following exceptions will be caught when they occur at IRQL less than or equal to *DISPATCH_LEVEL* (note that some of these are specific to the Intel x86 processor):

- Anything signaled by *ExRaiseStatus* and related functions
- Attempt to dereference invalid pointer to user-mode memory
- Debug or breakpoint exception
- Integer overflow (INTO instruction)
- Invalid *opcode*

Note that a reference to an invalid kernel-mode pointer leads directly to a bug check and can't be trapped. Likewise, a divide-by-zero exception or a BOUND instruction exception leads to a bug check.

Kernel-mode programs use structured exceptions by establishing exception frames on the same stack that's used for argument passing, subroutine calling, and automatic variables. A dedicated processor register points to the current exception frame. Each frame points to the preceding frame. Whenever an exception occurs, the kernel searches the list of exception frames for an exception handler. It will always find one because there is an exception frame at the very top of the stack that will handle any otherwise unhandled exception. Once the kernel locates an exception handler, it unwinds the execution and exception frame stacks in parallel, calling cleanup handlers along the way. Then it gives control to the exception handler.

When you use the Microsoft compiler, you can use Microsoft extensions to the C/C++ language that hide some of the complexities of working with the raw operating system primitives. You use the *__try* statement to designate a compound statement as the guarded body for an exception frame, and you use either the *__finally* statement to establish a termination handler or the *__except* statement to establish an exception handler.

NOTE
It's better to always spell the words *__try*, *__finally*, and *__except* with leading underscores. In C compilation units, the DDK header file WARNING.H defines macros spelled try, finally, and except to be the words with underscores. DDK sample programs use those macro names rather than the underscored names. The problem this can create for you is that in a C++ compilation unit, *try* is a statement verb that pairs with catch to invoke a completely different exception mechanism that's part of the C++ language. C++ exceptions don't work in a driver unless you manage to duplicate some infrastructure from the run-time library. Microsoft would prefer you not do that because of the increased size of your driver and the memory pool overhead associated with handling the *throw* verb.

Try-Finally Blocks

It's easiest to begin explaining structured exception handling by describing the try-finally block, which you can use to provide cleanup code:

```
try
{
<guarded body>
}
finally
{
<termination handler>
}
```

In this fragment of pseudocode, the guarded body is a series of statements and subroutine calls that expresses some main idea in your program. In general, these statements have side effects. If there are no side effects, there's no particular point to using a try-finally block because there's nothing to clean up. The termination handler contains statements that undo some or all of the side effects that the guarded body might leave behind.

Semantically, the try-finally block works as follows: First the computer executes the guarded body. When control leaves the guarded body for any reason, the computer executes the termination handler. See Figure 3-3.

IPR2024-00292
KASPERSKY EX 1008 Page 67

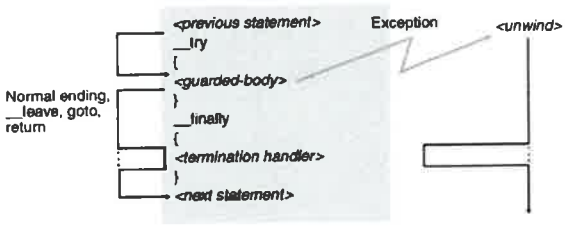


Figure 3-3. *Flow of control in a try-finally block.*

Here's one simple illustration:

```
LONG counter = 0;
  try
  {
  ++counter;
  }
  finally
  {
  --counter;
  }
KdPrint(("%d\n", counter));
```

First the guarded body executes and increments the *counter* variable from 0 to 1. When control "drops through" the right brace at the end of the guarded body, the termination handler executes and decrements *counter* back to 0. The value printed will therefore be 0.

Here's a slightly more complicated variation:

```
VOID RandomFunction(PLONG pcounter)
  {
    try
    {
    ++*pcounter;
    return;
    }
    finally
    {
    --*pcounter;
    }
  }
```

The net result of this function is no change to the integer at the end of the *pcounter* pointer: whenever control leaves the guarded body for any reason, including a *return* statement or a *goto*, the termination handler executes. Here the guarded body increments the counter and performs a return. Next the cleanup code executes and decrements the counter. Then the subroutine actually returns.

One final example should cement the idea of a try-finally block:

```
static LONG counter = 0;
  try
  {
  ++counter;
  BadActor();
  }
  finally
  {
  --counter;
  }
```

Here I'm supposing that we call a function, *BadActor*, that will raise some sort of exception that triggers a stack unwind. As part of the process of unwinding the execution and exception stacks, the operating system will invoke our cleanup code to restore the counter to its previous value. The system then continues unwinding the stack, so whatever code we have after the *__finally* block won't get executed.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 68

Try-Except Blocks

The other way to use structured exception handling involves a try-except block:

```
try
{
<guarded body>
}
except(<filter expression>)
{
<exception handler>
}
```

The guarded body in a try-except block is code that might fail by generating an exception. Perhaps you're going to call a kernel-mode service function such as *MmProbeAndLockPages* that uses pointers derived from user mode without explicit validity checking. Perhaps you have other reasons. In any case, if you manage to get all the way through the guarded body without an error, control continues after the exception handler code. You'll think of this case as being the normal one. If an exception arises in your code or in any of the subroutines you call, however, the operating system will unwind the execution stack, evaluating the filter expressions in *__except* statements. These expressions yield one of the following values:

- *EXCEPTION_EXECUTE_HANDLER* is numerically equal to 1 and tells the operating system to transfer control to your exception handler. If your handler falls through the ending right brace, control continues within your program at the statement immediately following that right brace. (I've seen Platform SDK documentation to the effect that control returns to the point of the exception, but that's not correct.)
- *EXCEPTION_CONTINUE_SEARCH* is numerically equal to 0 and tells the operating system that you can't handle the exception. The system keeps scanning up the stack looking for another handler. If no one has provided a handler for the exception, a system crash will occur.
- *EXCEPTION_CONTINUE_EXECUTION* is numerically equal to -1 and tells the operating system to return to the point where the exception was raised. I'll have a bit more to say about this expression value a little further on.

Take a look at Figure 3-4 for the possible control paths within and around a try-except block.

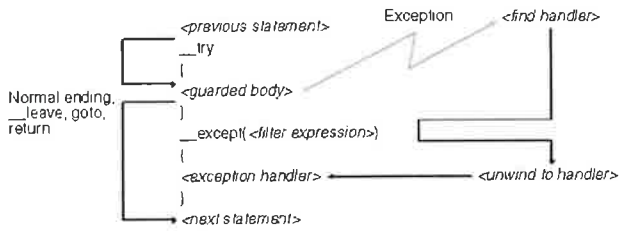


Figure 3-4. *Flow of control in a try-except block.*

For example, you can protect yourself from receiving an invalid pointer by using code like the following. (See the SEHTEST sample in the companion content.)

```
PVOID p = (PVOID) 1;
  try
  {
  KdPrint(("About to generate exception\n"));
  ProbeForWrite(p, 4, 4);
  KdPrint(("You shouldn't see this message\n"));
  }
  except(EXCEPTION EXECUTE HANDLER)
  {
  KdPrint(("Exception was caught\n"));
  }
KdPrint(("Program kept control after exception\n"));
```

ProbeForWrite tests a data area for validity. In this example, it will raise an exception because the pointer argument we supply isn't aligned to a 4-byte boundary. The exception handler gains control. Control then flows to the next statement after the exception handler and continues within your program.

In the preceding example, had you returned the value *EXCEPTION_CONTINUE_SEARCH*, the operating system would have continued unwinding the stack looking for an exception handler. Neither your exception handler code nor the code following it would have been executed: either the system would have crashed or some higher-level handler would have taken over.

IPR2024-00292
KASPERSKY EX 1008 Page 69

You should not return *EXCEPTION_CONTINUE_EXECUTION* in kernel mode because you have no way to alter the conditions that caused the exception in order to allow a retry to occur.

Note that you cannot trap arithmetic exceptions, or page faults due to referencing an invalid kernel-mode pointer, by using structured exceptions. You just have to write your code so as not to generate such exceptions. It's pretty obvious how to avoid dividing by 0—just check, as in this example:

```
ULONG numerator, denominator;  // <== numbers someone gives you
ULONG quotient;
if (!denominator)
  <handle error>else
  quotient = numerator / denominator;
```

But what about a pointer that comes to you from some other part of the kernel? There is no function that you can use to check the validity of a kernel-mode pointer. You just need to follow this rule:

Usually, trust values that a kernel-mode component gives you.

I don't mean by this that you shouldn't liberally sprinkle your code with *ASSERT* statements—you should because you may not initially understand all the ins and outs of how other kernel components work. I just mean that you don't need to burden your own driver with excessive defenses against mistakes in other, well-tested, parts of the system unless you need to work around a bug.

More About *NULL* Pointers

While we're on the subject of invalid pointers, note that a NULL pointer is (a) an invalid user-mode pointer in Windows XP and (b) a perfectly valid pointer in Windows 98/Me. If you use a *NULL* pointer directly, as in *p, or indirectly, as in *p->StructureMember*, you'll be trying to reference something in the first few bytes of virtual memory. Doing so in Windows XP will cause a trappable access violation.

Dereferencing a *NULL* pointer in Windows 98/Me will not, of itself, cause any immediately observable problem. I once spent several days tracking down a bug that resulted from overstoring location 0x0000000C in a Windows 95 system. That location is the real-mode vector for the breakpoint (INT 3) interrupt. The wild store didn't show up until some infrequently used application did an INT 3 that wasn't caught by a debugger. The system reflected the interrupt to real mode. The invalid interrupt vector pointed to memory containing a bunch of technically valid but nonsensical instructions followed by an invalid one. The system halted with an invalid operation exception. As you can see, the eventual symptom was very far removed in space and time from the wild store.

To debug a different problem in Windows 98, I once installed a debugging driver to catch alterations to the first 16 bytes of virtual memory. I had to remove it because so many VxD drivers (including some belonging to Microsoft) were getting caught.

The moral of these anecdotes is that you should always test pointers for *NULL* before using them if there is any possibility that the pointer could be *NULL*. To learn whether the possibility exists, read documentation and specifications very carefully.

Exception Filter Expressions

You might be wondering how to perform any sort of involved error detection or correction when all you're allowed to do is evaluate an expression that yields one of three integer values. You could use the C/C++ comma operator to string expressions together:

```
__except(expr-1, ... EXCEPTION_CONTINUE_SEARCH){}
```

The comma operator basically discards whatever value is on its left side and evaluates its right side. The value that's left over after this computational game of musical chairs (with just one chair!) is the value of the expression.

You could use the C/C++ conditional operator to perform a more involved calculation:

```
except(<some-expr>
  ? EXCEPTION EXECUTE HANDLER
  : EXCEPTION_CONTINUE_SEARCH)
```

If the *some_expr* expression is *TRUE*, you execute your own handler. Otherwise, you tell the operating system to keep looking for another handler above you in the stack.

Finally, it should be obvious that you could just write a subroutine whose return value is one of the *EXCEPTION_Xxx* values:

```
LONG EvaluateException()
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 70

```
{
if (<some-expr>)
  return EXCEPTION EXECUTE HANDLER;
else
  return EXCEPTION CONTINUE SEARCH;
}

except(EvaluateException())
```

For any of these expression formats to do you any good, you need access to more information about the exception. You can call two functions when evaluating an *__except* expression that will supply the information you need. Both functions actually have intrinsic implementations in the Microsoft compiler and can be used only at the specific times indicated:

- *GetExceptionCode()* returns the numeric code for the current exception. This value is an NTSTATUS value that you can compare with manifest constants in ntstatus.h if you want to. This function is available in an __except expression and within the exception handler code that follows the *__except* clause.
- *GetExceptionInformation()* returns the address of an EXCEPTION_POINTERS structure that, in turn, allows you to learn all the details about the exception, such as where it occurred, what the machine registers contained at the time, and so on. This function is available only within an *__except* expression.

NOTE
The scope rules for names that appear in try-except and try-finally blocks are the same as elsewhere in the C/C++ language. In particular, if you declare variables within the scope of the compound statement that follows *__try*, those names aren't visible in a filter expression, an exception handler, or a termination handler. Documentation to the contrary that you might have seen in the Platform SDK or on MSDN is incorrect. For what it's worth, the stack frame containing any local variables declared within the scope of the guarded body still exists at the time the filter expression is evaluated. So if you had a pointer (presumably declared at some outer scope) to a variable declared within the guarded body, you could safely dereference it in a filter expression.

Because of the restrictions on how you can use these two expressions in your program, you'll probably want to use them in a function call to some filter function, like this:

```
LONG EvaluateException(NTSTATUS status, PEXCEPTION POINTERS xp)
{

}

except(EvaluateException(GetExceptionCode(),
GetExceptionInformation()))
```

Raising Exceptions

Program bugs are one way you can (inadvertently) raise exceptions that invoke the structured exception handling mechanism. Application programmers are familiar with the Win32 API function *RaiseException*, which allows you to generate an arbitrary exception on your own. In WDM drivers, you can call the routines listed in Table 3-2. I'm not going to give you a specific example of calling these functions because of the following rule:

Raise an exception only in a nonarbitrary thread context, when you know there's an exception handler above you, and when you really know what you're doing.

In particular, raising exceptions is not a good way to tell your callers information that you discover in the ordinary course of executing. It's far better to return a status code, even though that leads to apparently more unreadable code. You should avoid exceptions because the stack-unwinding mechanism is very expensive. Even the cost of establishing exception frames is significant and something to avoid when you can.

Service Function	Description
ExRaiseStatus	Raise exception with specified status code
ExRaiseAccessViolation	Raise STATUS_ACCESS_VIOLATION
ExRaiseDatatypeMisalignment	Raise *STATUS_DATATYPE_MISALIGNMENT*

Table 3-2. *Service Functions for Raising Exceptions*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 71

Real-World Examples



Notwithstanding the expense of setting up and tearing down exception frames, you have to use structured exception syntax in an ordinary driver in particular situations.

One of the times you must set up an exception handler is when you call *MmProbeAndLockPages* to lock the pages for a memory descriptor list (MDL) you've created:

```
PMDL mdl = MmCreateMdl(...);
  try
  {
  MmProbeAndLockPages(mdl, ...);
  }
  except(EXCEPTION EXECUTE HANDLER)
  {
  NTSTATUS status = GetExceptionCode();
  IoFreeMdl(mdl);
  return CompleteRequest(Irp, status, 0);
  }
```

(*CompleteRequest* is a helper function I use to handle the mechanics of completing I/O requests. Chapter 5 explains all about I/O requests and what it means to complete one.)

Another time to use an exception handler is when you want to access user-mode memory using a pointer from an untrusted source. In the following example, suppose you obtained the pointer p from a user-mode program and believe it points to an integer:

```
PLONG p;        // from user-mode
  try
  {
  ProbeForRead(p, 4, 4);
  LONG x = *p;
  :
  }
  except(EXCEPTION EXECUTE HANDLER)
  {
  NTSTATUS status = GetExceptionCode();
  :
  }
```

3.2.3 Bug Checks

Unrecoverable errors in kernel mode can manifest themselves in the so-called blue screen of death (BSOD) that's all too familiar to driver programmers. Figure 3-5 is an example (hand-painted because no screen capture software is running when one of these occurs!). Internally, these errors are called bug checks, after the service function you use to diagnose their occurrence: *KeBugCheckEx*. The main feature of a bug check is that the system shuts itself down in as orderly a way as possible and presents the BSOD. Once the BSOD appears, the system is dead and must be rebooted.

You call *KeBugCheckEx* like this:

```
KeBugCheckEx(bugcode, info1, info2, info3, info4);
```

where *bugcode* is a numeric value identifying the cause of the error and info1, info2, and so on are integer parameters that will appear in the BSOD display to help a programmer understand the details of the error. This function does not return (!).

As a developer, you don't get much information from the Blue Screen. If you're lucky, the information will include the offset of an instruction within your driver. Later on, you can examine this location in a kernel debugger and, perhaps, deduce a possible cause for the bug check. Microsoft's own bug-check codes appear in bugcodes.h (one of the DDK headers); a fuller explanation of the codes and their various parameters can be found in Knowledge Base article Q103059, "Descriptions of Bug Codes for Windows NT," which is available on MSDN, among other places.

Sample Code
The BUGCHECK sample driver illustrates how to call *KeBugCheckEx*. I used it to generate the screen shot for Figure 3-5.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 72

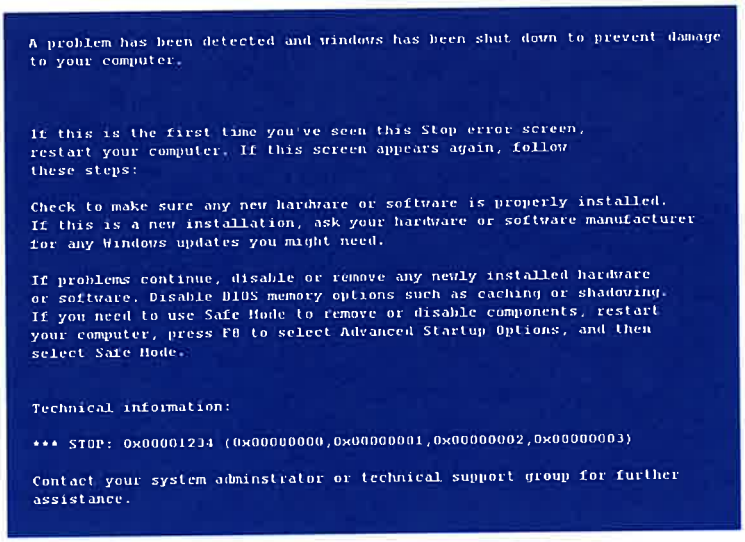


Figure 3-5. *The blue screen of death.*

You can certainly create your own bug-check codes if you want. The Microsoft values are simple integers beginning with 1 (*APC_INDEX_MISMATCH*) and (currently) extending through 0xF6 (*PCI_VERIFIER_DETECTED_VIOLATION*), along with a few others. To create your own bug-check code, define an integer constant as if it were *STATUS_SEVERITY_SUCCESS* status code, but supply either the customer flag or a nonzero facility code. For example:

```
#define MY_BUGCHECK_CODE 0x002A0001
.
.
KeBugCheckEx(MY_BUGCHECK_CODE, 0, 0, 0, 0);
```

You use a nonzero facility code (42 in this example) or the customer flag (which I left 0 in this example) so that you can tell your own codes from the ones Microsoft uses.

Now that I've told you how to generate your own BSOD, let me tell you when to do it: never. Or at most, in the checked build of your driver for use during your own internal debugging. You and I are unlikely to write a driver that will discover an error so serious that taking down the system is the only solution. It would be far better to log the error (using the error-logging facilities I'll describe in Chapter 14) and return a status code.

Note that the end user can configure the behavior of *KeBugCheckEx* in the advanced settings for My Computer. The user can choose to automatically restart the machine or to generate the BSOD. The end user can likewise choose several levels of detail (including none) for a dump file and whether to log an event in the system event log.

3.3 Memory Management

In this section, I'll discuss the topic of memory management. Windows XP divides the available virtual address space in several ways. One division—a very firm one based on security and integrity concerns—is between user-mode addresses and kernel-mode addresses. Another division, which is almost but not quite coextensive with the first, is between paged and nonpaged memory. All user-mode addresses and some kernel-mode addresses reference page frames that the Memory Manager swaps to and from the disk over time, while some kernel-mode addresses always reference page frames in physical memory. Since Windows XP allows portions of drivers to be paged, I'll explain how you control the pageability of your driver at the time you build your driver and at run time.

Windows XP provides several methods for managing memory. I'll describe two basic service functions—*ExAllocatePoolWithTag* and *ExFreePool*—that you use for allocating and releasing randomly sized blocks from a heap. I'll also describe the primitives that you use for organizing memory blocks into linked lists of structures. Finally I'll describe the concept of a lookaside list, which allows you to efficiently allocate and release blocks that are all the same size.

3.3.1 User-Mode and Kernel-Mode Address Spaces

Windows XP and Microsoft Windows 98/Me run on computers that support a virtual address space, wherein virtual addresses are mapped either to physical memory or (conceptually, anyway) to page frames within a swap file on disk. To grossly simplify matters, you can think of the virtual address space as being divided into two parts: a kernel-mode part and a user-mode part. See Figure 3-6.

IPR2024-00292
KASPERSKY EX 1008 Page 73

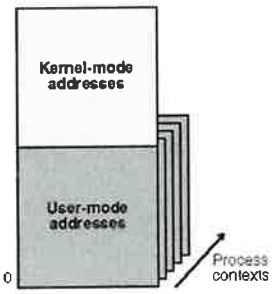


Figure 3-6. *User-mode and kernel-mode portions of the address space.*

Each user-mode process has its own address context, which maps the user-mode virtual addresses to a unique collection of physical page frames. In other words, the meaning of any particular virtual address changes from one moment to the next as the Windows XP scheduler switches from a thread in one process to a thread in another process. Part of the work in switching threads is to change the page tables used by a processor so that they refer to the incoming thread's process context.

It's generally unlikely that a WDM driver will execute in the same thread context as the initiator of the I/O requests it handles. We say that we're running in *arbitrary thread context* if we don't know for sure to which process the current user-mode address context belongs. In arbitrary thread context, we simply can't use a virtual address that belongs to user mode because we can't have any idea to what physical memory it might point. In view of this uncertainty, we generally obey the following rule inside a driver program:

Never (well, hardly ever) directly reference user-mode memory.

In other words, don't take an address that a user-mode application provides and treat that address as a pointer that we can directly dereference. I'll discuss in later chapters a few techniques for accessing data buffers that originate in user mode. All we need to know right now, though, is that we're (nearly) always going to be using kernel-mode virtual addresses whenever we want to access the computer's memory.

How Big Is a Page?

In a virtual memory system, the operating system organizes physical memory and the swap file into like-size page frames. In a WDM driver, you can use the manifest constant *PAGE_SIZE* to tell you how big a page is. In some Windows XP computers, a page is 4096 bytes long; in others, it's 8192 bytes long. A related constant named *PAGE_SHIFT* equals the page size as a power of 2. That is:

```
PAGE_SIZE == 1 << PAGE_SHIFT
```

For your convenience, you can use a few preprocessor macros in your code when you're working with the size of a page:

- *ROUND_TO_PAGES* rounds a size in bytes to the next-higher page boundary. For example, *ROUND_TO_PAGES*(1) is 4096 on a 4-KB-page computer.
- *BYTES_TO_PAGES* determines how many pages are required to hold a given number of bytes beginning at the start of a page. For example, *BYTES_TO_PAGES*(42) would be 1 on all platforms, and *BYTES_TO_PAGES*(5000) would be 2 on some platforms and 1 on others.
- *BYTE_OFFSET* returns the byte offset portion of a virtual address. That is, it calculates the starting offset within some page frame of a given address. On a 4-KB-page computer, *BYTE_OFFSET*(0x12345678) would be 0x678.
- *PAGE_ALIGN* rounds a virtual address down to a page boundary. On a 4-KB-page computer, *PAGE_ALIGN*(0x12345678) would be 0x12345000.
- *ADDRESS_AND_SIZE_TO_SPAN_PAGES* returns the number of page frames occupied by a specified number of bytes beginning at a specified virtual address. For example, the statement *ADDRESS_AND_SIZE_TO_SPAN_PAGES*(0x12345FFF, 2) is 2 on a 4-KB-page machine because the 2 bytes span a page boundary.

Paged and Nonpaged Memory

The whole point of a virtual memory system is that you can have a virtual address space that's much bigger than the amount of physical memory on the computer. To accomplish this feat, the Memory Manager needs to swap page frames in and out of physical memory. Certain parts of the operating system can't be paged, though, because they're needed to support the Memory Manager itself. The most obvious example of something that must always be resident in memory is the code that handles page faults (the exceptions that occur when a page frame isn't physically present when needed) and the data structures used by the page fault handler.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 74

The category of "must be resident" stuff is much broader than just the page fault handlers. Windows XP allows hardware interrupts to occur at nearly any time, including while a page fault is being serviced. If this weren't so, the page fault handler wouldn't be able to read or write pages from a device that uses an interrupt. Thus, every hardware interrupt service routine must be in nonpaged memory. The designers of Windows NT decided to include even more routines in the nonpaged category by using a simple rule:

Code executing at or above interrupt request level (IRQL) DISPATCH_LEVEL cannot cause page faults.

I'll elaborate on this rule in the next chapter.

You can use the *PAGED_CODE* preprocessor macro (declared in wdm.h) to help you discover violations of this rule in the checked build of your driver. For example:

```
NTSTATUS DispatchPower(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PAGED_CODE()
:
  }
```

PAGED_CODE contains conditional compilation. In the checked-build environment, it prints a message and generates an assertion failure if the current IRQL is too high. In the free-build environment, it doesn't do anything. To understand why PAGED_CODE is useful, imagine that *DispatchPower* needs for some reason to be in nonpaged memory but that you have misplaced it in paged memory. If the system happens to call *DispatchPower* at a time when the page containing it isn't present, a page fault will occur, followed by a bug check. The bug check code will be pretty uninformative (*IRQL_NOT_LESS_OR_EQUAL* or *DRIVER_IRQL_NOT_LESS_OR_EQUAL*), but at least you'll find out that you have a problem. If you test your driver in a situation in which the page containing *DispatchPower* happens fortuitously to be in memory, though, there won't be a page fault. *PAGED_CODE* will detect the problem even so.

☑ Setting the Driver Verifier "Force IRQL Checking" option will greatly increase the chances of discovering that you've broken the rule about paging and IRQL. The option forces pageable pages out of memory whenever verified drivers raise the IRQL to *DISPATCH_LEVEL* or beyond.

Compile-Time Control of Pageability

Given that some parts of your driver must always be resident and some parts can be paged, you need a way to control the assignment of your code and data to the paged and nonpaged pools. You accomplish part of this job by instructing the compiler how to apportion your code and data among various sections. The run-time loader uses the names of the sections to put parts of your driver in the places you intend. You can also accomplish parts of this job at run time by calling various Memory Manager routines that I'll discuss in the next section.

NOTE
Win32 executable files, including kernel-mode drivers, are internally composed of one or more sections. A section can contain code or data and, generally speaking, has additional attributes such as being readable, writable, shareable, executable, and so on. A section is also the smallest unit that you can designate when you're specifying pageability. When loading a driver image, the system puts sections whose literal names begin with *PAGE* or *.EDA* (the start of *.EDATA*) into the paged pool unless the *DisablePagingExecutive* value in the HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management key happens to be set (in which case no driver paging occurs). Note that these names are case sensitive! In one of the little twists of fate that affect us all from time to time, running Soft-Ice/W on Windows XP requires you to disable kernel paging in this way. This certainly makes it harder to find bugs caused by misplacement of driver code or data into the paged pool! If you use this debugger, I recommend that you religiously use the *PAGED_CODE* macro and the Driver Verifier.

The traditional way of telling the compiler to put code into a particular section is to use the alloc_text pragma. Since not every compiler will necessarily support the pragma, the DDK headers either define or don't define the constant ALLOC_PRAGMA to tell you whether to use the pragma. You can then invoke the pragma to specify the section placement of individual subroutines in your driver, as follows:

```
#ifdef ALLOC_PRAGMA
  #pragma alloc_text(PAGE, AddDevice)
  #pragma alloc_text(PAGE, DispatchPnp)
:
#endif
```

These statements serve to place the *AddDevice* and *DispatchPnp* functions into the paged pool.

The Microsoft C/C++ compiler places two annoying restrictions on using *alloc_text*:

- The pragma must follow the declaration of a function but precede the definition. One way to obey this rule is to declare all the functions in your driver in a standard header file and invoke alloc_text at the start of the source file that contains a

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 75

given function but after you include that header.

- The pragma can be used only with functions that have C-linkage. In other words, it won't work for class member functions or for functions in a C++ source file that you didn't declare using extern "C".

To control the placement of data variables, you use a different pragma under the control of a different preprocessor macro symbol:

```
#ifdef ALLOC_DATA_PRAGMA
  #pragma data_seg("PAGEDATA")
#endif
```

The *data_seg* pragma causes all static data variables declared in a source module after the appearance of the pragma to go into the paged pool. You'll notice that this pragma differs in a fundamental way from *alloc_text*. A pageable section starts where *#pragma data_seg("PAGEDATA")* appears and ends where a countervailing *#pragma data_seg()* appears. *Alloc_text*, on the other hand, applies to a specific function.

More About Section Placement

In general, I find it more convenient to specify the section placement of whole blocks of code by using the Microsoft *code_seg* pragma, which works the same way as *data_seg*, only for code. That is, you can tell the Microsoft compiler to start putting functions into the paged pool like this:

```
#pragma code_seg("PAGE")
NTSTATUS AddDevice(...){...}
NTSTATUS DispatchPnp(...){...}
```

The *AddDevice* and *DispatchPnp* functions would both end up in the paged pool. You can check to see whether you're compiling with the Microsoft compiler by testing the existence of the predefined preprocessor macro *_MSC_VER*.

To revert to the default code section, just code *#pragma code_seg* with no argument:

```
#pragma code_seg()
```

Similarly, to revert to the regular nonpaged data section, code *#pragma data_seg* with no argument:

```
#pragma data_seg()
```

This sidebar is also the logical place to mention that you can also direct code into the INIT section if it's not needed once your driver finishes initializing. For example:

```
#pragma alloc_text(INIT, DriverEntry)
```

This statement forces the *DriverEntry* function into the INIT section. The system will release the memory it occupies when it returns. This small savings isn't very important in the grand scheme of things because a WDM driver's *DriverEntry* function isn't very big. Previous Windows NT drivers had large *DriverEntry* functions that had to create device objects, locate resources, configure devices, and so on. For them, using this feature offered significant memory savings.

Notwithstanding the low utility of putting *DriverEntry* in the INIT section in a WDM driver, I was in the habit of doing so until quite recently. Because of a bug in Windows 98/Me, I had a situation in which a WDM driver wasn't being completely removed from memory after I unplugged my hardware. One part of the system didn't understand this and tried to call *DriverEntry* when I replugged the hardware. The memory that had originally contained *DriverEntry* had long since been overwritten by INIT code belonging to other drivers, and a crash resulted. This was very difficult to debug! I now prefer to place *DriverEntry* in a paged section.

You can use the DUMPBIN utility that comes with Microsoft Visual C++ .NET to easily see how much of your driver is initially pageable. Your marketing department might even want to crow about how much less nonpaged memory you use than your competitors.

Run-Time Control of Pageability

Table 3-3 lists the service functions you can use at run time to fine-tune the pageability of your driver in various situations. The purpose of these routines is to let you release the physical memory that would otherwise be tied up by your code and data during periods when it won't be needed. In Chapter 8, for example, I'll discuss how you can put your device into a low power state during periods of inactivity. Powering down might be a good time to release your locked pages.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 76

Service Function	Description
MmLockPagableCodeSection	Locks a code section given an address inside it
MmLockPagableDataSection	Locks a data section given an address inside it
MmLockPagableSectionByHandle	Locks a code section by using a handle from a previous *MmLockPagableCodeSection* call (Windows 2000 and Windows XP only)
MmPageEntireDriver	Unlocks all pages belonging to driver
MmResetDriverPaging	Restores compile-time pageability attributes for entire driver
MmUnlockPagableImageSection	Unlocks a locked code or data section

Table 3-3. *Routines for Dynamically Locking and Unlocking Driver Pages*

I'm going to describe one way to use these functions to control the pageability of code in your driver. You might want to read the DDK descriptions to learn about other ways to use them. First distribute subroutines in your driver into separately named code sections, like this:

```
#pragma alloc_text(PAGEIDLE, DispatchRead)
#pragma alloc_text(PAGEIDLE, DispatchWrite)
:
```

That is, define a section name beginning with *PAGE* and ending in any four-character suffix you please. Then use the *alloc_text* pragma to place some group of your own routines in that special section. You can have as many special pageable sections as you want, but your logistical problems will grow as you subdivide your driver in this way.

During initialization (say, in *DriverEntry*), lock your pageable sections like this:

```
PVOID hPageIdleSection;
NTSTATUS DriverEntry(...)
  {
  hPageIdleSection = MmLockPagableCodeSection((PVOID) DispatchRead);
  }
```

When you call *MmLockPagableCodeSection*, you specify any address at all within the section you're trying to lock. The real purpose of making this call during *DriverEntry* is to obtain the handle value it returns, which I've shown you saving in a global variable named *hPageIdleSection*. You'll use that handle much later on, when you decide you don't need a particular section in memory for a while:

```
MmUnlockPagableImageSection(hPageIdleSection);
```

This call will unlock the pages containing the PAGEIDLE section and allow them to move in and out of memory on demand. If you later discover that you need those pages back again, you make this call:

```
MmLockPagableSectionByHandle(hPageIdleSection);
```

Following this call, the PAGEIDLE section will once again be in nonpaged memory (but not necessarily the same physical memory as previously). Note that this function call is available to you only in Windows 2000 and Windows XP, and then only if you've included ntddk.h instead of wdm.h. In other situations, you will have to call *MmLockPagableCodeSection* again.

You can do something similar to place data objects into pageable sections:

```
PVOID hPageDataSection;

#pragma data_seg("PAGE")
ULONG ulSomething;
#pragma data_seg()

hPageDataSection = MmLockPagableDataSection((PVOID)&ulSomething);

MmUnlockPagableImageSection(hPageDataSection);

MmLockPagableSectionByHandle(hPageDataSection);
```

I've played fast and loose with my syntax here—these statements would appear in widely separated parts of your driver.

The key idea behind the Memory Manager service functions I just described is that you initially lock a section containing one or more pages and obtain a handle for use in subsequent calls. You can then unlock the pages in a particular section by calling *MmUnlockPagableImageSection* and passing the corresponding handle. Relocking the section later on requires a call to *MmLockPagableSectionByHandle*.

IPR2024-00292
KASPERSKY EX 1008 Page 77

A quick shortcut is available if you're sure that no part of your driver will need to be resident for a while. *MmPageEntireDriver* will mark all the sections in a driver's image as being pageable. Conversely, *MmResetDriverPaging* will restore the compile-time pageability attributes for the entire driver. To call these routines, you just need the address of some piece of code or data in the driver. For example:

```
MmPageEntireDriver((PVOID) DriverEntry);
:
MmResetDriverPaging((PVOID) DriverEntry);
```



You need to exercise care when using any of the Memory Manager routines I've just described if your device uses an interrupt. If you page your entire driver, the system will also page your interrupt service routine (ISR). If your device or any device with which you share an interrupt vector should interrupt, the system will try to call your ISR. Even if you think your interrupt isn't shared and you've inhibited your device from generating an interrupt, bear in mind that spurious interrupts have been known to occur. If the ISR isn't present, the system will crash. You avoid this problem by disconnecting your interrupt before allowing the ISR to be paged.

3.3.2 Heap Allocator

The basic heap allocation service function in kernel mode is *ExAllocatePoolWithTag*. You call it like this:

```
PVOID p = ExAllocatePoolWithTag(type, nbytes, tag);
```

The type argument is one of the *POOL_TYPE* enumeration constants described in Table 3-4, and *nbytes* is the number of bytes you want to allocate. The tag argument is an arbitrary 32-bit value. The return value is a kernel-mode virtual address pointer to the allocated memory block.

In most drivers, including the samples in this book and in the DDK, you'll see calls to an older function named *ExAllocatePool*:

```
PVOID p = ExAllocatePool(type, nbytes);
```

ExAllocatePool was the heap allocation function in the earliest versions of Windows NT. In the Windows XP DDK, *ExAllocatePool* is actually a macro that invokes *ExAllocatePoolWithTag* using the tag value '*mdW*' ('*Wdm*' plus a trailing space after byte reversal).

Pool Type	*Description*
NonPagedPool	Allocates from the nonpaged pool of memory
PagedPool	Allocates from the paged pool of memory
NonPagedPoolCacheAligned	Allocates from the nonpaged pool and ensures that memory is aligned with the CPU cache
PagedPoolCacheAligned	Allocates from the paged pool of memory and ensures that memory is aligned with the CPU cache

Table 3-4. *Pool Type Arguments for ExAllocatePool*

The most basic decision you must make when you call *ExAllocatePoolWithTag* is whether the allocated memory block can be swapped out of memory. That choice depends simply on which parts of your driver will need to access the memory block. If you'll be using a memory block at or above *DISPATCH_LEVEL*, you must allocate it from the nonpaged pool. If you'll always use the memory block below *DISPATCH_LEVEL*, you can allocate from the paged or nonpaged pool as you choose.



Allocations from the *PagedPool* must occur at an IRQL less than *DISPATCH_LEVEL*. Allocations from the *NonPagedPool* must occur at an IRQL less than or equal to *DISPATCH_LEVEL*. The Driver Verifier bug checks whether you violate either of these rules.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 78

Limits on Pool Allocations
A frequently asked question is, "How much memory can I allocate with one call to *ExAllocatePoolWithTag*?" Unfortunately, there's no simple answer to the question. A starting point is to determine the maximum sizes of the paged and non-paged pools. You can consult Knowledge Base article Q126402 and Chapter 7 of *Inside Windows 2000* (Microsoft Press, 2000) for (probably) more information than you'll ever want to know about this topic. By way of example, on a 512 MB machine, I ended up with a maximum nonpaged pool size of 128 MB and an actual paged pool size of 168 MB.

Knowing the pool sizes is not the end of the story, though. I wouldn't expect to be able to allocate anywhere close to 128 MB of nonpaged memory on this 512-MB computer in one call to *ExAllocatePoolWithTag*. For one thing, other parts of the system will have used up significant amounts of nonpaged memory by the time my driver gets a chance to try, and the system would probably run very poorly if I took all that was left over. For another thing, the virtual address space is likely to be fragmented once the system has been running for a while, so the heap manager wouldn't be able to find an extremely large contiguous range of unused virtual addresses.

In actual, not-very-scientific tests, using the MEMTEST sample from the companion content, I was able to allocate about 129 MB of paged memory and 100 MB of nonpaged memory in a single call.

Sample Code
The MEMTEST sample uses *ExAllocatePoolWithTagPriority* to determine the largest contiguous allocations possible from the paged and nonpaged pools.

When you use *ExAllocatePoolWithTag*, the system allocates 4 more bytes of memory than you asked for and returns you a pointer that's 4 bytes into that block. The tag occupies the initial 4 bytes and therefore precedes the pointer you receive. The tag will be visible to you when you examine memory blocks while debugging or while poring over a crash dump, and it can help you identify the source of a memory block that's involved in some problem or another. For example:

```
#define DRIVERTAG 'KNUJ'
.
.
PVOID p = ExAllocatePoolWithTag(PagedPool, 42, DRIVERTAG);
```

Here I used a 32-bit integer constant as the tag value. On a little-endian computer such as an x86, the bytes that compose this value will be reversed in memory to spell out a common word in the English language. Several features of the Driver Verifier relate to specific memory tags, by the way, so you can do yourself a favor in the debugging department by using one more unique tags in your allocation calls.

☑ Do not specify zero or " GIB" (BIG with a space at the end, after byte reversal) as a tag value. Zero-tagged blocks can't be tracked, and the system internally uses the BIG tag for its own purposes. Do not request zero bytes. This restriction could be a special concern if you're writing your own C or C++ runtime support, since *malloc* and *operator new* allow requests for zero bytes.

More About Pool Tags
Several diagnostic mechanisms inside the kernel depend on pool tagging, and you can help yourself analyze your driver's performance by picking a unique set of tags. You must also explicitly enable kernel pool tagging in the retail release of the system (it's enabled by default in the checked build) by using the GFLAGS.EXE utility. GFLAGS is part of the platform SDK and other components.

Having done both of these things—using unique tags in your driver and enabling pool tagging in the kernel—you can profitably use a few tools. POOLMON and POOLTAG in the DDK tools directory report on memory usage by tag value. You can also ask GFLAGS to make one of your pools "special" in order to check for overwrites.

The pointer you receive will be aligned with at least an 8-byte boundary. If you place an instance of some structure in the allocated memory, members to which the compiler assigns an offset divisible by 4 or 8 will therefore occupy an address divisible by 4 or 8 too. On some RISC platforms, of course, you must have doubleword and quadword values aligned in this way. For performance reasons, you might want to be sure that the memory block will fit in the fewest possible number of processor cache lines. You can specify one of the *XxxCacheAligned* type codes to achieve that result. If you ask for less than a page's worth of memory, the block will be contained in a single page. If you ask for at least a page's worth of memory, the block will start on a page boundary.

NOTE
Asking for PAGE_SIZE + 1 bytes of memory is about the worst thing you can do to the heap allocator: the system reserves two pages, of which nearly half will ultimately be wasted.

It should go without saying that you need to be extra careful when accessing memory you've allocated from the free storage pools in kernel mode. Since driver code executes in the most privileged mode possible for the processor, there's almost no protection from wild stores.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 79

Using GFLAGS or the Driver Verifier's Special Pool option allows you to find memory overwrite errors more easily. With this option, allocations from the "special" pool lie at the end of a page that's followed in virtual memory by a not-present page. Trying to touch memory past the end of your allocated block will earn you an immediate page fault. In addition, the allocator fills the rest of the page with a known pattern. When you eventually release the memory, the system checks to see whether you overwrote the pattern. In combination, these checks make it much easier to detect bugs that result from "coloring outside the lines" of your allocated memory. You can also ask to have allocations at the start of a page preceded by a not-present page, by the way. Refer to Knowledge Base article Q192486 for more information about the special pool.

Handling Low-Memory Situations

If there isn't enough memory to satisfy your request, the pool allocator returns a *NULL* pointer. You should always test the return value and do something reasonable. For example:

```
PMYSTUFF p = (PMYSTUFF) ExAllocatePool(PagedPool, sizeof(MYSTUFF));
if (!p)
  return STATUS_INSUFFICIENT_RESOURCES;
```

Additional pool types include the concept *must succeed*. If there isn't enough heap memory to satisify a request from the must-succeed pool, the system bug checks. Drivers should not allocate memory using one of the must-succeed specifiers. This is because a driver can nearly always fail whatever operation is under way. Causing a system crash in a low-memory situation is not something a driver should do. Furthermore, only a limited pool of must-succeed memory exists in the entire system, and the operating system might not be able to allocate memory needed to keep the computer running if drivers tie up some. In fact, Microsoft wishes it had never documented the must-succeed options in the DDK to begin with.

The Driver Verifier will bug check whether a driver specifies one of the must-succeed pool types in an allocation request. In addition, if you turn on the low-resource-simulation option in the Driver Verifier, your allocations will begin randomly failing after the system has been up seven or eight minutes. Every five minutes or so, the system will fail all your allocations for a burst of 10 seconds.

In some situations, you might want to use a technique that's commonly used in file system drivers. If you OR the value *POOL_RAISE_IF_ALLOCATION_FAILURE (0x00000010)* into the pool type code, the heap allocator will raise a *STATUS_INSUFFICIENT_RESOURCES* exception instead of returning *NULL* if there isn't enough memory. You should use a structured exception frame to catch such an exception. For example:

```
#ifndef POOL RAISE IF ALLOCATION FAILURE
  #define POOL RAISE IF ALLOCATION FAILURE 16
#endif

#define PagedPoolRaiseException (POOL TYPE) \
  (PagedPool | POOL RAISE IF ALLOCATION FAILURE)
#define NonPagedPoolRaiseException (POOL TYPE) \
  (NonPagedPool | POOL RAISE IF ALLOCATION FAILURE)
.
.
.
NTSTATUS SomeFunction()
  {
  NTSTATUS status;
    try
.   {
.
.
    PMYSTUFF p = (PMYSTUFF)
      ExAllocatePoolWithTag(PagedPoolRaiseException,
      sizeof(MYSTUFF), DRIVERTAG);
    <Code that uses "p" without checking it for NULL>
    status = STATUS SUCCESS;
    }
    except(EXCEPTION EXECUTE HANDLER)
    {
    status = GetExceptionCode();
    }
  return status;
  }
```

NOTE
POOL_RAISE_IF_ALLOCATION_FAILURE is defined in NTIFS.H, a header file that's available only as part of the extra-cost Installable File System kit. Doing memory allocations with this flag set is so common in file system drivers, though, that I thought you should know about it.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 80

Incidentally, I suggest that you not go crazy trying to diagnose or recover from failures to allocate small blocks of memory. As a practical matter, an allocation request for, say, 32 bytes is never going to fail. If memory were that tight, the system would be running so sluggishly that someone would surely reboot the machine. You must not be the cause of a system crash in this situation, though, because you would thereby be the potential source of a denial-of-service exploit. But since the error won't arise in real life, there's no point in putting elaborate code into your driver to log errors, signal WMI events, print debugging messages, execute alternative algorithms, and so on. Indeed, the extra code needed to do all that might be the reason the system didn't have the extra 32 bytes to give you in the first place! Thus, I recommend testing the return value from every call to *ExAllocatePoolWithTag*. If it discloses an error, do any required cleanup and return a status code. Period.

Releasing a Memory Block

To release a memory block you previously allocated with *ExAllocatePoolWithTag*, you call *ExFreePool*:

```
ExFreePool((PVOID) p);
```

You do need to keep track somehow of the memory you've allocated from the pool in order to release it when it's no longer needed. No one else will do that for you. You must sometimes closely read the DDK documentation of the functions you call with an eye toward memory ownership. For example, in the *AddDevice* function I showed you in the previous chapter, there's a call to *IoRegisterDeviceInterface*. That function has a side effect: it allocates a memory block to hold the string that names the interface. You are responsible for releasing that memory later on.

☑ The Driver Verifier checks at *DriverUnload* time to ensure a verified driver has released all the memory it allocated. In addition, the verifier sanity-checks all calls to *ExFreePool* to make sure they refer to a complete block of memory allocated from a pool consistent with the current IRQL.

The DDK headers declare an undocumented function named *ExFreePoolWithTag*. This function was intended for internal use only in order to make sure that system components didn't inadvertently release memory belonging to other components. The function was politely called a "waste of time" by one of the Microsoft developers, which pretty much tells us that we needn't worry about what it does or how to use it. (Hint: you need to do some other undocumented things in order to use it successfully.)

Two More Functions

Although *ExAllocatePoolWithTag* is the function you should use for heap allocation, you can use two other pool allocation functions in special circumstances: *ExAllocatePoolWithQuotaTag* (and a macro named *ExAllocatePoolWithQuota* that supplies a default tag) and *ExAllocatePoolWithTagPriority*. *ExAllocatePoolWithQuotaTag* allocates a memory block and charges the current thread's scheduling quota. This function is for use by file system drivers and other drivers running in a nonarbitrary thread context for allocating memory that belongs to the current thread. A driver wouldn't ordinarily use this function because a quota violation causes the system to raise an exception.

ExAllocatePoolWithTagPriority, which is new with Windows XP, allows you to specify how important you consider it to be that a memory allocation request succeed:

```
PVOID p = ExAllocatePoolWithTagPriority(type, nbytes, tag, priority);
```

The arguments are the same as we've been studying except that you also supply an additional priority indicator. See Table 3-5.

Priority Argument	Description
LowPoolPriority	System may fail request if low on resources. Driver can easily cope with failure.
NormalPoolPriority	System may fail request if low on resources.
HighPoolPriority	System should not fail request unless completely out of resources.

Table 3-5. *Pool Priority Arguments for ExAllocatePoolWithTagPriority*

The DDK indicates that most drivers should specify *NormalPoolPriority* when calling this function. *HighPoolPriority* should be reserved for situations in which success is critically important to the continued working of the system.

☑ You can lexigraphically append the phrases *SpecialPoolOverrun* and *SpecialPoolUnderrun* to the names given in Table 3-5 (for example, *LowPoolPrioritySpecialPoolOverrun*, and so on). If an allocation would use the special pool, the overrun and underrun flags override the default placement of blocks.

At the time I'm writing this, *ExAllocatePoolWithTagPriority* turns into a simple call to *ExAllocatePoolWithTag* if you are asking for paged memory at high priority or nonpaged memory at any priority. The extra resource checking happens only with requests for paged memory at low or normal priority. This behavior could change in service packs or later versions of the operating system.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 81

3.3.3 Linked Lists

Windows XP makes extensive use of linked lists as a way of organizing collections of similar data structures. In this chapter, I'll discuss the basic service functions you use to manage doubly-linked and singly-linked lists. Separate service functions allow you to share linked lists between threads and across multiple processors; I'll describe those functions in the next chapter after I've explained the synchronization primitives on which they depend.

Whether you organize data structures into a doubly-linked or a singly-linked list, you normally embed a linking substructure—either a *LIST_ENTRY* or a *SINGLE_LIST_ENTRY*—into your own data structure. You also reserve a list head element somewhere that uses the same structure as the linking element. For example:

```
typedef struct  TWOWAY
  {
  LIST ENTRY linkfield;
  } TWOWAY, *PTWOWAY;

LIST ENTRY DoubleHead;

typedef struct  ONEWAY
  {
  SINGLE LIST ENTRY linkfield;
  } ONEWAY, *PONEWAY;

SINGLE_LIST_ENTRY SingleHead;
```

When you call one of the list-management service functions, you always work with the linking field or the list head—never directly with the containing structures themselves. So suppose you have a pointer (*pdElement*) to one of your *TWOWAY* structures. To put that structure on a list, you'd reference the embedded linking field like this:

```
InsertTailList(&DoubleHead, &pdElement->linkfield);
```

Similarly, when you retrieve an element from a list, you're really getting the address of the embedded linking field. To recover the address of the containing structure, you can use the *CONTAINING_RECORD* macro. (See Figure 3-7.)

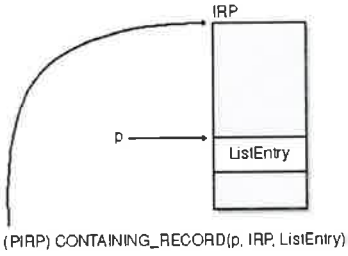


Figure 3-7. *The CONTAINING_RECORD macro.*

So if you wanted to process and discard all the elements in a singly-linked list, your code would look something like this:

```
PSINGLE LIST ENTRY psLink = PopEntryList(&SingleHead);
while (psLink)
  {
  PONEWAY psElement = CONTAINING RECORD(psLink, ONEWAY, linkfield);
  ExFreePool(psElement);
  psLink = PopEntryList(&SingleHead);
  }
```

Just before the start of this loop, and again after every iteration, you retrieve the current first element of the list by calling *PopEntryList*. *PopEntryList* returns the address of the linking field within a *ONEWAY* structure, or else it returns NULL to signify that the list is empty. Don't just indiscriminately use *CONTAINING_RECORD* to develop an element address that you then test for NULL—you need to test the link field address that *PopEntryList* returns!

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 82

Doubly-Linked Lists

A doubly-linked list links its elements both backward and forward in a circular fashion. See Figure 3-8. That is, starting with any element, you can proceed forward or backward in a circle and get back to the same element. The key feature of a doubly-linked list is that you can add or remove elements anywhere in the list.

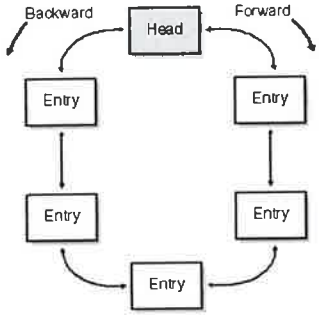


Figure 3-8. *Topology of a doubly-linked list.*

Table 3-6 lists the service functions you use to manage a doubly-linked list.

Service Function or Macro	Description
InitializeListHead	Initializes the *LIST_ENTRY* at the head of the list
InsertHeadList	Inserts element at the beginning
InsertTailList	Inserts element at the end
IsListEmpty	Determines whether list is empty
RemoveEntryList	Removes element
RemoveHeadList	Removes first element
RemoveTailList	Removes last element

Table 3-6. *Service Functions for Use with Doubly-Linked List*

Here is a fragment of a fictitious program to illustrate how to use some of these functions:

```
typedef struct  TWOWAY {
:
  LIST ENTRY linkfield;
:
   } TWOWAY, *PTWOWAY;

LIST ENTRY DoubleHead;

1
InitializeListHead(&DoubleHead);
ASSERT(IsListEmpty(&DoubleHead));

PTWOWAY pdElement = (PTWOWAY) ExAllocatePool(PagedPool,sizeof(TWOWAY));
2
InsertTailList(&DoubleHead, &pdElement->linkfield);
:
3
if (!IsListEmpty(&DoubleHead))
   {
4
   PLIST ENTRY pdLink = RemoveHeadList(&DoubleHead);
   pdElement = CONTAINING RECORD(pdLink, TWOWAY, linkfield);
:
   ExFreePool(pdElement);
   }
```

1. *InitializeListHead* initializes a *LIST_ENTRY* to point (both backward and forward) to itself. That configuration indicates that the list is empty.

2. *InsertTailList* puts an element at the end of the list. Notice that you specify the address of the embedded linking field instead of your own *TWOWAY* structure. You could call *InsertHeadList* to put the element at the beginning of the list

IPR2024-00292
KASPERSKY EX 1008 Page 83

instead of the end. By supplying the address of the link field in some existing *TWOWAY* structure, you could put the new element either just after or just before the existing one.

```
PTWOWAY prev;
InsertHeadList(&prev->linkfield, &pdElement->linkfield);

PTWOWAY next;
InsertTailList(&next->linkfield, &pdElement->linkfield);
```

3. Recall that an empty doubly-linked list has the list head pointing to itself, both backward and forward. Use *IsListEmpty* to simplify making this check. The return value from *RemoveXxxList* will never be *NULL*!

4. *RemoveHeadList* removes the element at the head of the list and gives you back the address of the linking field inside it. *RemoveTailList* does the same thing, just with the element at the end of the list instead.

It's important to know the exact way *RemoveHeadList* and *RemoveTailList* are implemented if you want to avoid errors. For example, consider the following innocent-looking statement:

```
if (<some-expr>)
  pdLink = RemoveHeadList(&DoubleHead);
```

What I obviously intended with this construction was to conditionally extract the first element from a list. C'est raisonnable, n'est-ce pas? But no, when you debug this later on, you find that elements keep mysteriously disappearing from the list. You discover that *pdLink* gets updated only when the *if* expression is *TRUE* but that *RemoveHeadList* seems to get called even when the expression is *FALSE*.

Mon dieu! What's going on here? Well, *RemoveHeadList* is really a macro that expands into multiple statements. Here's what the compiler really sees in the above statement:

```
if (<some-expr>)
  pdLink = (&DoubleHead)->Flink;
{{
PLIST ENTRY  EX Blink;
PLIST ENTRY  EX Flink;
 EX Flink = ((&DoubleHead)->Flink)->Flink;
 EX Blink = ((&DoubleHead)->Flink)->Blink;
 EX Blink->Flink =  EX Flink;
 EX Flink->Blink =  EX Blink;
}}
```

Aha! Now the reason for the mysterious disappearance of list elements becomes clear. The *TRUE* branch of the *if* statement consists of just the single statement *pdLink = (&DoubleHead)->Flink*, which stores a pointer to the first element. The logic that removes a list element stands alone outside the scope of the if statement and is therefore always executed. Both *RemoveHeadList* and *RemoveTailList* amount to an expression plus a compound statement, and you dare not use either of them in a spot where the syntax requires an expression or a statement alone. Zut alors!

The other list-manipulation macros don't have this problem, by the way. The difficulty with *RemoveHeadList* and *RemoveTailList* arises because they have to return a value and do some list manipulation. The other macros do only one or the other, and they're syntactically safe when used as intended.

Singly-Linked Lists

A singly-linked list links its elements in only one direction, as illustrated in Figure 3-9. Windows XP uses singly-linked lists to implement pushdown stacks, as suggested by the names of the service routines in Table 3-7. Just as was true for doubly-linked lists, these "functions" are actually implemented as macros in wdm.h, and similar cautions apply. *PushEntryList* and *PopEntryList* generate multiple statements, so you can use them only on the right side of an equal sign in a context in which the compiler is expecting multiple statements.

Service Function or Macro	Description
PushEntryList	Adds element to top of list
PopEntryList	Removes topmost element

Table 3-7. *Service Functions for Use with Singly-Linked Lists*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 84



Figure 3-9. *Topology of a singly-linked list.*

The following pseudofunction illustrates how to manipulate a singly-linked list:

```
typedef struct  ONEWAY {

SINGLE LIST ENTRY linkfield;
} ONEWAY, *PONEWAY;

SINGLE LIST ENTRY SingleHead;

1
SingleHead.Next = NULL;

PONEWAY psElement = (PONEWAY) ExAllocatePool(PagedPool,
  sizeof(ONEWAY));
2
PushEntryList(&SingleHead, &psElement->linkfield);

3
SINGLE LIST ENTRY psLink = PopEntryList(&SingleHead);
if (psLink)
  {
  psElement = CONTAINING RECORD(psLink, ONEWAY, linkfield);

  ExFreePool(psElement);
  }
```

1. Instead of invoking a service function to initialize the head of a singly-linked list, just set the *Next* field to *NULL*. Note also the absence of a service function for testing whether this list is empty; just test *Next* yourself.

2. *PushEntryList* puts an element at the head of the list, which is the only part of the list that's directly accessible. Notice that you specify the address of the embedded linking field instead of your own *ONEWAY* structure.

3. *PopEntryList* removes the first entry from the list and gives you back a pointer to the link field inside it. In contrast with doubly-linked lists, a *NULL* value indicates that the list is empty. In fact, there's no counterpart to *IsListEmpty* for use with a singly-linked list.

3.3.4 Lookaside Lists

Even employing the best possible algorithms, a heap manager that deals with randomly sized blocks of memory will require some scarce processor time to coalesce adjacent free blocks from time to time. Figure 3-10 illustrates how, when something returns block B to the heap at a time when blocks A and C are already free, the heap manager can combine blocks A, B, and C to form a single large block. The large block is then available to satisfy some later request for a block bigger than any of the original three components.

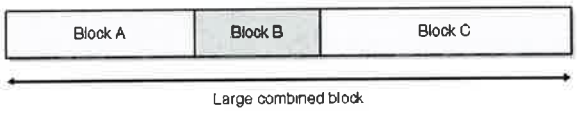


Figure 3-10. *Coalescing adjacent free blocks in a heap.*

If you know you're always going to be working with fixed-size blocks of memory, you can craft a much more efficient scheme for managing a heap. You can, for example, *preallocate* a large block of memory that you subdivide into pieces of the given fixed size. Then you can devise some scheme for knowing which blocks are free and which are in use, as suggested by Figure 3-11. Returning a block to such a heap merely involves marking it as free—you don't need to coalesce it with adjacent blocks

IPR2024-00292
KASPERSKY EX 1008 Page 85

because you never need to satisfy randomly sized requests.

Merely allocating a large block that you subdivide might not be the best way to implement a fixed-size heap, though. In general, it's hard to guess how much memory to preallocate. If you guess too high, you'll be wasting memory. If you guess too low, your algorithm will either fail when it runs out (bad!) or make too-frequent trips to a surrounding random heap manager to get space for more blocks (better). Microsoft has created the lookaside list object and a set of adaptive algorithms to deal with these shortcomings.

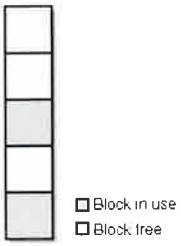


Figure 3-11. *A heap containing fixed-size blocks.*

Figure 3-12 illustrates the concept of a lookaside list. Imagine that you had a glass that you could (somehow—the laws of physics don't exactly make this easy!) balance upright in a swimming pool. The glass represents the lookaside list object. When you initialize the object, you tell the system how big the memory blocks (water drops, in this analogy) are that you'll be working with. In earlier versions of Windows NT, you could also specify the capacity of the glass, but the operating system now determines that adaptively. To allocate a memory block, the system first tries to remove one from the list (remove a water drop from the glass). If there are no more, the system dips into the surrounding memory pool. Conversely, to return a memory block, the system first tries to put it back on the list (add a water drop to the glass). But if the list is full, the block goes back into the pool using the regular heap manager routine (the drop slops over into the swimming pool).

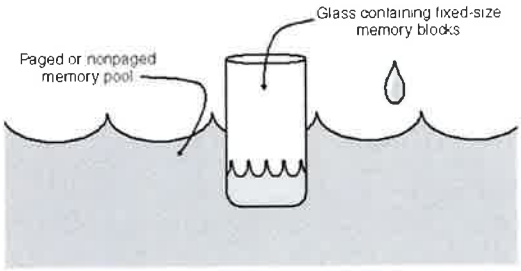


Figure 3-12. *Lookaside lists.*

The system periodically adjusts the depths of all lookaside lists based on actual usage. The details of the algorithm aren't really important, and they're subject to change in any case. Basically (in the current release, anyway), the system will reduce the depth of lookaside lists that haven't been accessed recently or that aren't forcing pool access at least 5 percent of the time. The depth never goes below 4, however, which is also the initial depth of a new list.

☑ When the Driver Verifier is running, all lookaside lists are set to a depth of zero, which forces all allocation and free calls to go directly to the pool. This action makes it more likely that driver problems involving memory corruption can be caught. Just bear this fact in mind when you're debugging your driver with the Driver Verifier engaged.

Table 3-8 lists the eight service functions that you use when you work with a lookaside list. There are really two sets of four functions, one set for a lookaside list that manages paged memory (the *ExXxxPagedLookasideList* set) and another for a lookaside list that manages nonpaged memory (the *ExXxxNPagedLookasideList* set). The first thing you must do is reserve nonpaged memory for a PAGED_LOOKASIDE_LIST or an *NPAGED_LOOKASIDE_LIST* object. Even the paged variety of object needs to be in nonpaged memory because the system will access the list object itself at an elevated IRQL.

Service Function	Description
ExInitializeNPagedLookasideList ExInitializePagedLookasideList	Initialize a lookaside list
ExAllocateFromNPagedLookasideList ExAllocateFromPagedLookasideList	Allocate a fixed-size block
ExFreeToNPagedLookasideList ExFreeToPagedLookasideList	Release a block back to a lookaside list
ExDeleteNPagedLookasideList ExDeletePagedLookasideList	Destroy a lookaside list

Table 3-8. *Service Functions for Lookaside List*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 86

After reserving storage for the lookaside list object somewhere, you call the appropriate initialization routine:

```
PPAGED_LOOKASIDE_LIST pagedlist;
PNPAGED_LOOKASIDE_LIST nonpagedlist;

ExInitializePagedLookasideList(pagedlist, Allocate, Free, 0, blocksize, tag, 0);
ExInitializeNPagedLookasideList(nonpagedlist, Allocate, Free,
  0, blocksize, tag, 0);
```

(The only difference between the two examples is the spelling of the function name and the first argument.)

The first argument to either of these functions points to the *[N]PAGED_LOOKASIDE_LIST* object for which you've already reserved space. Allocate and Free are pointers to routines you can write to allocate or release memory from a random heap. You can use *NULL* for either or both of these parameters, in which case *ExAllocatePoolWithTag* and *ExFreePool* will be used, respectively. The *blocksize* parameter is the size of the memory blocks you will be allocating from the list, and *tag* is the 32-bit tag value you want placed in front of each such block. The two zero arguments are placeholders for values that you supplied in previous versions of Windows NT but that the system now determines on its own; these values are flags to control the type of allocation and the depth of the lookaside list.

To allocate a memory block from the list, call the appropriate *AllocateFrom* function:

```
PVOID p = ExAllocateFromPagedLookasideList(pagedlist);
PVOID q = ExAllocateFromNPagedLookasideList(nonpagedlist);
```

To put a block back onto the list, call the appropriate *FreeTo* function:

```
ExFreeToPagedLookasideList(pagedlist, p);
ExFreeToNPagedLookasideList(nonpagedlist, q);
```

Finally, to destroy a list, call the appropriate *Delete* function:

```
ExDeletePagedLookasidelist(pagedlist);
ExDeleteNPagedLookasideList(nonpagedlist);
```



It is very important for you to explicitly delete a lookaside list before allowing the list object to pass out of scope. I'm told that a common programming mistake is to place a lookaside list object in a device extension and then forget to delete the object before calling *IoDeleteDevice*. If you make this mistake, the next time the system runs through its list of lookaside lists to tune their depths, it will put its foot down on the spot where your list object used to be, probably with bad results.

3.4 String Handling

WDM drivers can work with string data in any of four formats:

- A Unicode string, normally described by a *UNICODE_STRING* structure, contains 16-bit characters. Unicode has sufficient code points to accommodate the language scripts used on this planet. A whimsical attempt to standardize code points for the Klingon language, reported in the first edition, has been rejected. A reader of the first edition sent me the following e-mail comment about this:



  I suspect this is rude, and possibly obscene.

- An ANSI string, normally described by an *ANSI_STRING* structure, contains 8-bit characters. A variant is an *OEM_STRING*, which also describes a string of 8-bit characters. The difference between the two is that an *OEM* string has characters whose graphic depends on the current code page, whereas an ANSI string has characters whose graphic is independent of code page. WDM drivers won't normally deal with *OEM* strings because they would have to originate in user mode, and some other kernel-mode component will have already translated them into Unicode strings by the time the driver sees them.

- A null-terminated string of characters. You can express constants using normal C syntax, such as "Hello, world!" Strings employ 8-bit characters of type *CHAR*, which are assumed to be from the ANSI character set. The characters in string constants originate in whatever editor you used to create your source code. If you use an editor that relies on the then-current code page to display graphics in the editing window, be aware that some characters might have a different meaning when treated as part of the Windows ANSI character set.

IPR2024-00292
KASPERSKY EX 1008 Page 87

- A null-terminated string of wide characters (type *WCHAR*). You can express wide string constants using normal C syntax, such as *L"Goodbye, cruel world!"* Such strings look like Unicode constants, but, being ultimately derived from some text editor or another, actually use only the ASCII and Latin1 code points (0020-007F and 00A0-00FF) that correspond to the Windows ANSI set.

The *UNICODE_STRING* and *ANSI_STRING* data structures both have the layout depicted in Figure 3-13. The *Buffer* field of either structure points to a data area elsewhere in memory that contains the string data. *MaximumLength* gives the length of the buffer area, and *Length* provides the (current) length of the string without regard to any null terminator that might be present. Both length fields are in bytes, even for the *UNICODE_STRING* structure.

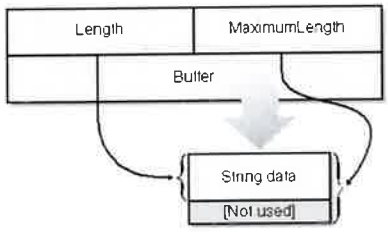


Figure 3-13 *The UNICODE_STRING and ANSI_STRING structures.*

The kernel defines three categories of functions for working with Unicode and ANSI strings. One category has names beginning with *Rtl* (for run-time library). Another category includes most of the functions that are in a standard C library for managing null-terminated strings. The third category includes the safe string functions from strsafe.h, which will hopefully be packaged in a DDK header named NtStrsafe.h by the time you read this. I can't add any value to the DDK documentation by repeating what it says about the *RtlXxx* functions. I have, however, distilled in Table 3-9 a list of now-deprecated standard C string functions and the recommended alternatives from NtStrsafe.h.

Standard function (deprecated)	Safe UNICODE Alternative	Safe ANSI Alternative
strcpy, wcscpy, strncpy, wcsncpy	*RtlStringCbCopyW, RtlStringCchCopyW*	*RtlStringCbCopyA, RtlStringCchCopyA*
strcat, wcscat, strncat, wcsncat	*RtlStringCbCatW, RtlStringCchCatW*	*RtlStringCbCatA, RtlStringCchCatA*
sprintf, swprintf, _snprintf, _snwprintf	*RtlStringCbPrintfW, RtlStringCchPrintfW*	*RtlStringCbPrintfA, RtlStringCchPrintfA*
vsprintf, vswprintf, vsnprintf, _vsnwprintf	*RtlStringCbVPrintfW, RtlStringCchVPrintfW*	*RtlStringCbVPrintfA, RtlStringCchVPrintfA*
strlen, wcslen	*RtlStringCbLengthW, RtlStringCchLengthW*	*RtlStringCbLengthA, RtlStringCchLengthA*

Table 3-9 *Safe Functions for String Manipulation*

NOTE
I based the contents of Table 3-9 on a description of how one of the kernel developers planned to craft NtStrsafe.h from an existing user-mode header named strsafe.h. Don't trust me—trust the contents of the DDK!

It's also okay, but not idiomatic, to use *memcpy*, *memmove*, *memcmp*, and *memset* in a driver. Nonetheless, most driver programmers use these *RtlXxx* functions in preference:

- *RtlCopyMemory* or *RtlCopyBytes* instead of *memcpy* to copy a "blob" of bytes from one place to another. These functions are actually identical in the current Windows XP DDK. Furthermore, for Intel 32-bit targets, both are macro'ed to *memcpy*, and *memcpy* is the subject of a *#pragma* intrinsic, so the compiler generates inline code to perform it.
- *RtlZeroMemory* instead of *memset* to zero an area of memory. *RtlZeroMemory* is macro'ed to *memset* for Intel 32-bit targets, and *memset* is mentioned in a *#pragma intrinsic*.



You should use the safe string functions in preference to standard run-time routines such as *strcpy* and the like. As I mentioned at the outset of this chapter, the standard string functions are available, but they're often too hard to use safely. Consider these points in choosing which string functions you'll use in your driver:

- The uncounted forms *strcpy*, *strcat*, *sprintf*, and *vsprintf* (and their Unicode equivalents) don't protect you against overrunning the target buffer. Neither does *strncat* (and its Unicode equivalent), wherein the length argument applies to the source string.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 88

- The *strncpy* and *wcsncpy* functions will fail to append a null terminator to the target if the source is at least as long as the specified length. In addition, these functions have the possibly expensive feature of filling any leftover portion of the target buffer with nulls.
- Any of the deprecated functions has the potential to walk off the end of a memory page looking in vain for a null terminator. This trait makes them especially dangerous when dealing with string data coming to you from user mode.
- As I write this, NtStrsafe.h doesn't currently define any comparison functions (*strcmp*, etc.). Keep your eye on the DDK for such functions. Note that case-insensitive comparisons of ANSI strings are tricky because they depend on localization settings that can vary from one session to another on the same computer.

Allocating and Releasing String Buffers

You often define *UNICODE_STRING* (or *ANSI_STRING*) structures as automatic variables or as parts of your own device extension. The string buffers to which these structures point usually occupy dynamically allocated memory, but you'll sometimes want to work with string constants too. Keeping track of who owns the memory to which a particular *UNICODE_STRING* or *ANSI_STRING* structure points can be a bit of a problem. Consider the following fragment of a function:

```
UNICODE_STRING foo;
if (bArriving)
  RtlInitUnicodeString(&foo, L"Hello, world!");
else
  {
  ANSI_STRING bar;
  RtlInitAnsiString(&bar, "Goodbye, cruel world!");
  RtlAnsiStringToUnicodeString(&foo, &bar, TRUE);
  }
:
RtlFreeUnicodeString(&foo); // <== don't do this!
```

In one case, we initialize *foo.Length*, *foo.MaximumLength*, and *foo.Buffer* to describe a wide character string constant in our driver. In another case, we ask the system (by means of the TRUE third argument to *RtlAnsiStringToUnicodeString*) to allocate memory for the Unicode translation of an ANSI string. In the first case, it's a mistake to call *RtlFreeUnicodeString* because it will unconditionally try to release a memory block that's part of our code or data. In the second case, it's mandatory to call *RtlFreeUnicodeString* eventually if we want to avoid a memory leak.

The moral of the preceding example is that you have to know where the memory comes from in any *UNICODE_STRING* structures you use so that you can release the memory only when necessary.

3.5 Miscellaneous Programming Techniques

In the remainder of this chapter, I'm going to discuss some miscellaneous topics that might be useful in various parts of your driver. I'll begin by describing how you access the registry database, which is where you can find various configuration and control information that might affect your code or your hardware. I'll go on to describe how you access disk files and other named devices. A few words will suffice to describe how you can perform floating-point calculations in a WDM driver. Finally I'll describe a few of the features you can embed in your code to make it easier to debug your driver in the unlikely event it shouldn't work correctly the first time you try it out.

3.5.1 Accessing the Registry

Windows XP and Windows 98/Me record configuration and other important information in a database called the registry. WDM drivers can call the functions listed in Table 3-10 to access the registry. If you've done user-mode programming involving registry access, you might be able to guess how to use these functions in a driver. I found the kernel-mode support functions sufficiently different, however, that I think it's worth describing how you might use them.

In this section, I'll discuss, among other things, the *ZwXxx* family of routines and *RtlDeleteRegistryValue*, which provide the basic registry functionality that suffices for most WDM drivers.

Opening a Registry Key

Before you can interrogate values in the registry, you need to open the key that contains them. You use *ZwOpenKey* to open an existing key. You use *ZwCreateKey* either to open an existing key or to create a new key. Either function requires you to first initialize an *OBJECT_ATTRIBUTES* structure with the name of the key and (perhaps) other information. The *OBJECT_ATTRIBUTES* structure has the following declaration:

IPR2024-00292
KASPERSKY EX 1008 Page 89

```
typedef struct  OBJECT ATTRIBUTES {
  ULONG Length;
  HANDLE RootDirectory;
  PUNICODE STRING ObjectName;
  ULONG Attributes;
  PVOID SecurityDescriptor;
  PVOID SecurityQualityOfService;
  } OBJECT_ATTRIBUTES;
```

Service Function	Description
IoOpenDeviceRegistryKey	Opens special key associated with a physical device object (PDO)
IoOpenDeviceInterfaceRegistryKey	Opens a registry key associated with a registered device interface
RtlDeleteRegistryValue	Deletes a registry value
RtlQueryRegistryValues	Reads several values from the registry
RtlWriteRegistryValue	Writes a value to the registry
ZwClose	Closes handle to a registry key
ZwCreateKey	Creates a registry key
ZwDeleteKey	Deletes a registry key
ZwDeleteValueKey	Deletes a value (Windows 2000 and later)
ZwEnumerateKey	Enumerates subkeys
ZwEnumerateValueKey	Enumerates values within a registry key
ZwFlushKey	Commits registry changes to disk
ZwOpenKey	Opens a registry key
ZwQueryKey	Gets information about a registry key
ZwQueryValueKey	Gets a value within a registry key
ZwSetValueKey	Sets a value within a registry key

Table 3-10. *Service Functions for Registry Access*

Rather than initialize an instance of this structure by hand, it's easiest to use the macro *InitializeObjectAttributes*, which I'm about to show you.

Suppose, for example, that we wanted to open the service key for our driver. The I/O Manager gives us the name of this key as a parameter to *DriverEntry*. So we could write code like the following:

```
NTSTATUS DriverEntry(PDRIVER OBJECT DriverObject, PUNICODE STRING RegistryPath)
: {
:
  OBJECT ATTRIBUTES oa;
1
  InitializeObjectAttributes(&oa, RegistryPath, OBJ KERNEL HANDLE |
  OBJ CASE INSENSITIVE, NULL, NULL);
  HANDLE hkey;
2
  status = ZwOpenKey(&hkey, KEY READ, &oa);
  if (NT SUCCESS(status))
: {
:
3
    ZwClose(hkey);
: }
:
  }
```

1. We're initializing the object attributes structure with the registry pathname supplied to us by the I/O Manager and with a *NULL* security descriptor. *ZwOpenKey* will ignore the security descriptor anyway—you can specify security attributes only when you create a key for the first time.
2. *ZwOpenKey* will open the key for reading and store the resulting handle in our *hkey* variable.
3. *ZwClose* is a generic routine for closing a handle to a kernel-mode object. Here we use it to close the handle we have to the registry key.



The *OBJ_KERNEL_HANDLE* flag, shown in the preceding code sample, is important for system integrity. If you're running in the context of a user thread when you call *ZwOpenKey*, and if you don't supply this flag bit, the handle you get will be available to the user-mode process. It might even happen that user-mode code will close the handle and open

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 90

a new object, receiving back the same numeric value. All of a sudden, your calls to registry functions will be dealing with the wrong kind of handle.

Even though we often refer to the registry as being a database, it doesn't have all of the attributes that have come to be associated with real databases. It doesn't allow for committing or rolling back changes, for example. Furthermore, the access rights you specify when you open a key (*KEY_READ* in the preceding example) are for security checking rather than for the prevention of incompatible sharing. That is, two different processes can have the same key open after specifying write access (for example). The system does guard against destructive writes that occur simultaneously with reads, however, and it does guarantee that a key won't be deleted while someone has an open handle to it.

Other Ways to Open Registry Keys

In addition to *ZwOpenKey*, Windows XP provides two other functions for opening registry keys.

IoOpenDeviceRegistryKey allows you to open one of the special registry keys associated with a device object:

```
HANDLE hkey;
Status = IoOpenDeviceRegistryKey(pdo, flag, access, &hkey);
```

where pdo is the address of the physical device object (PDO) at the bottom of your particular driver stack, flag is an indicator for which special key you want to open (see Table 3-11), and access is an access mask such as *KEY_READ*.

Flag Value	Selected Registry Key
PLUGPLAY_REGKEY_DEVICE	The hardware (instance) subkey of the Enum key
PLUGPLAY_REGKEY_DRIVER	The driver subkey of the class key

Table 3-11. *Registry Key Codes for IoOpenDeviceRegistryKey*

I use *IoOpenDeviceRegistryKey* with the PLUGPLAY_REGKEY_DEVICE flag very often in my own drivers. In Windows XP, this function opens the Device Parameters subkey of the hardware key for the device. In Windows 98/Me, it opens the hardware key itself. These keys are the right place to store parameter information about the hardware. I'll discuss this key more fully in Chapter 15 in connection with installing and distributing a driver.

IoOpenDeviceInterfaceRegistryKey opens the key associated with an instance of a registered device interface:

```
HANDLE hkey;
status = IoOpenDeviceInterfaceRegistryKey(linkname, access,   &hkey);
```

where linkname is the symbolic link name of the registered interface and access is an access mask such as *KEY_READ*.

The interface registry key is a subkey of HKLM\System\CurrentControlSet\Control\DeviceClasses that persists from one session to the next. It's a good place to store parameter information that you want to share with user-mode programs because user-mode code can call *SetupDiOpenDeviceInterfaceRegKey* to gain access to the same key.

Getting and Setting Values

Usually, you open a registry key because you want to retrieve a value from the database. The basic function you use for that purpose is *ZwQueryValueKey*. For example, to retrieve the *ImagePath* value in the driver's service key—I don't actually know why you'd want to know this, but that's not my department—you could use the following code:

```
UNICODE_STRING valname;
RtlInitUnicodeString(&valname, L"ImagePath");
size = 0;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation,
  NULL, 0, &size);
if (status == STATUS_OBJECT_NAME_NOT_FOUND || size == 0)
  <handle error>;
size = min(size, PAGE_SIZE);
PKEY_VALUE_PARTIAL_INFORMATION vpip =
  PKEY_VALUE_PARTIAL_INFORMATION) ExAllocatePool(PagedPool, size);
if (!vpip)
  <handle error>;
status = ZwQueryValueKey(hkey, &valname, KeyValuePartialInformation,
  vpip, size, &size);
if (!NT_SUCCESS(status))
  <handle error>;
<do something with vpip->Data>ExFreePool(vpip);
```

Here we make two calls to *ZwQueryValueKey*. The purpose of the first call is to determine how much space we need to allocate

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 91

for the *KEY_VALUE_PARTIAL_INFORMATION* structure we're trying to retrieve. The second call retrieves the information. I left the error checking in this code fragment because the errors didn't work out in practice the way I expected them to. In particular, I initially guessed that the first call to *ZwQueryValueKey* would return *STATUS_BUFFER_TOO_SMALL* (since I passed it a zero-length buffer). It didn't do that, though. The important failure code is *STATUS_OBJECT_NAME_NOT_FOUND*, which indicates that the value doesn't actually exist. Hence, I test for that value only. If there's some other error that prevents *ZwQueryValueKey* from working, the second call will uncover it.



NOTE
The reason for trimming the size to *PAGE_SIZE* is to impose a reasonableness limit on the amount of memory you allocate. If a malicious user were able to gain access to the key from which you're reading, he or she could replace the *ImagePath* value with an arbitrary amount of data. Your driver could then become an unwitting accomplice to a denial of service attack by consuming mass quantities of memory. Now, drivers ordinarily deal with registry keys that only administrators can modify, and an administrator has many other ways of attacking the system. It's nonetheless good to provide a defense in depth against all forms of attack.

The so-called "partial" information structure you retrieve in this way contains the value's data and a description of its data type:

```
typedef struct  KEY VALUE PARTIAL INFORMATION {
  ULONG   TitleIndex;
  ULONG   Type;
  ULONG   DataLength;
  UCHAR   Data[1];
  } KEY_VALUE_PARTIAL_INFORMATION,  *PKEY_VALUE_PARTIAL_INFORMATION;
```

Type is one of the registry data types listed in Table 3-12. (Additional data types are possible but not interesting to device drivers.) *DataLength* is the length of the data value, and Data is the data itself. *TitleIndex* has no relevance to drivers. Here are some useful facts to know about the various data types:

- *REG_DWORD* is a 32-bit unsigned integer in whatever format (big-endian or little-endian) is natural for the platform.
- *REG_SZ* describes a null-terminated Unicode string value. The null terminator is included in the *DataLength* count.
- To expand a *REG_EXPAND_SZ* value by substituting environment variables, you should use *RtlQueryRegistryValues* as your method of interrogating the registry. The internal routines for accessing environment variables aren't documented or exposed for use by drivers.
- *RtlQueryRegistryValues* is also a good way to interrogate *REG_MULTI_SZ* values in that it will call your designated callback routine once for each of the potentially many strings.

NOTE
Notwithstanding the apparent utility of *RtlQueryRegistryValues*, I've avoided using it ever since it caused a crash in one of my drivers. Apparently, the value I was reading required the function to call a helper function that was placed in the INITIALIZATION section of the kernel and that was, therefore, no longer present.

Data TypeConstant	Description
REG_BINARY	Variable-length binary data
REG_DWORD	Unsigned long integer in natural format for the platform
REG_EXPAND_SZ	Null-terminated Unicode string containing %-escapes for environment variable names
REG_MULTI_SZ	One or more null-terminated Unicode strings, followed by an extra null
REG_SZ	Null-terminated Unicode string

Table 3-12. *Types of Registry Values Useful to WDM Drivers*

To set a registry value, you must have *KEY_SET_VALUE* access to the parent key. I used *KEY_READ* earlier, which wouldn't give you such access. You could use *KEY_WRITE* or *KEY_ALL_ACCESS*, although you thereby gain more than the necessary permission. Then call *ZwSetValueKey*. For example:

```
RtlInitUnicodeString(&valname, L"TheAnswer");
ULONG value = 42;
ZwSetValueKey(hkey, &valname, 0, REG_DWORD, &value, sizeof(value));
```

Deleting Subkeys or Values

To delete a value in an open key, you can use *RtlDeleteRegistryValue* in the following special way:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 92

```
RtlDeleteRegistryValue(RTL_REGISTRY_HANDLE, (PCWSTR) hkey, L"TheAnswer");
```

RtlDeleteRegistryValue is a general service function whose first argument can designate one of several special places in the registry. When you use *RTL_REGISTRY_HANDLE*, as I did in this example, you indicate that you already have an open handle to the key within which you want to delete a value. You specify the key (with a cast to make the compiler happy) as the second argument. The third and final argument is the null-terminated Unicode name of the value you want to delete. This is one time when you don't have to create a *UNICODE_STRING* structure to describe the string.

In Windows 2000 and later, you can use *ZwDeleteValueKey* to delete a value (it's an oversight that this function isn't documented in the DDK):

```
UNICODE_STRING valname;
RtlInitUnicodeString(&valname, L"TheAnswer");
ZwDeleteValueKey(hkey, &valname);
```

You can delete only those keys that you've opened with at least *DELETE* permission (which you get with *KEY_ALL_ACCESS*). You call *ZwDeleteKey*:

```
ZwDeleteKey(hkey);
```

The key lives on until all handles are closed, but subsequent attempts to open a new handle to the key or to access the key by using any currently open handle will fail with *STATUS_KEY_DELETED*. Since you have an open handle at this point, you must be sure to call *ZwClose* sometime. (The DDK documentation entry for *ZwDeleteKey* says the handle becomes invalid. It doesn't—you must still close it by calling *ZwClose*.)

Enumerating Subkeys or Values

A complicated activity you can carry out with an open registry key is to enumerate the elements (subkeys and values) that the key contains. To do this, you'll first call *ZwQueryKey* to determine a few facts about the subkeys and values, such as their number, the length of the largest name, and so on. *ZwQueryKey* has an argument that indicates which of three types of information you want to retrieve about the key. These types are named basic, node, and full. To prepare for an enumeration, you'd be interested first in the full information:

```
typedef struct  KEY_FULL_INFORMATION {
    LARGE_INTEGER LastWriteTime;
    ULONG   TitleIndex;
    ULONG   ClassOffset;
    ULONG   ClassLength;
    ULONG   SubKeys;
    ULONG   MaxNameLen;
    ULONG   MaxClassLen;
    ULONG   Values;
    ULONG   MaxValueNameLen;
    ULONG   MaxValueDataLen;
    WCHAR   Class[1];
} KEY_FULL_INFORMATION, *PKEY_FULL_INFORMATION;
```

This structure is actually of variable length since *Class[0]* is just the first character of the class name. It's customary to make one call to find out how big a buffer you need to allocate and a second call to get the data, as follows:

```
ULONG size;
ZwQueryKey(hkey, KeyFullInformation, NULL, 0, &size);
size = min(size, PAGE_SIZE);
PKEY_FULL_INFORMATION fip = (PKEY_FULL_INFORMATION)
  ExAllocatePool(PagedPool, size);
ZwQueryKey(hkey, KeyFullInformation, fip, size, &size);
```

Were you now interested in the subkeys of your registry key, you could perform the following loop calling *ZwEnumerateKey*:

```
for (ULONG i = 0; i < fip->SubKeys; ++i)
  {
  ZwEnumerateKey(hkey, i, KeyBasicInformation, NULL, 0, &size);
  size = min(size, PAGE_SIZE);
  PKEY_BASIC_INFORMATION bip = (PKEY_BASIC_INFORMATION)
    ExAllocatePool(PagedPool, size);
  ZwEnumerateKey(hkey, i, KeyBasicInformation, bip, size, &size);
  <do something with bip->Name>
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 93

```
    ExFreePool(bip);
    }
```

The key fact you discover about each subkey is its name, which shows up as a counted Unicode string in the *KEY_BASIC_INFORMATION* structure you retrieve inside the loop:

```
typedef struct  KEY_BASIC_INFORMATION {
    LARGE_INTEGER LastWriteTime;
    ULONG   Type;
    ULONG   NameLength;
    WCHAR   Name[1];
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;
```

The name isn't null-terminated; you must use the *NameLength* member of the structure to determine its length. Don't forget that the length is in bytes! The name isn't the full registry path either; it's just the name of the subkey within whatever key contains it. This is actually lucky because you can easily open a subkey given its name and an open handle to its parent key.

To accomplish an enumeration of the values in an open key, employ the following method:

```
ULONG maxlen = fip->MaxValueNameLen + sizeof(KEY_VALUE_BASIC_INFORMATION);
maxlen = min(maxlen, PAGE_SIZE);
PKEY_VALUE_BASIC_INFORMATION vip = (PKEY_VALUE_BASIC_INFORMATION)
  ExAllocatePool(PagedPool, maxlen);
for (ULONG i = 0; i < fip->Values; ++i)
  {
  ZwEnumerateValueKey(hkey, i, KeyValueBasicInformation, vip, maxlen, &size);
  <do something with vip->Name>
  }
ExFreePool(vip);
```

Allocate space for the largest possible *KEY_VALUE_BASIC_INFORMATION* structure that you'll ever retrieve based on the *MaxValueNameLen* member of the *KEY_FULL_INFORMATION* structure. Inside the loop, you'll want to do something with the name of the value, which comes to you as a counted Unicode string in this structure:

```
typedef struct  KEY_VALUE_BASIC_INFORMATION {
    ULONG   TitleIndex;
    ULONG   Type;
    ULONG   NameLength;
    WCHAR   Name[1];
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

Once again, having the name of the value and an open handle to its parent key is just what you need to retrieve the value, as shown in the preceding section.

There are variations on *ZwQueryKey* and on these two enumeration functions that I haven't discussed. You can, for example, obtain full information about a subkey when you call *ZwEnumerateKey*. I showed you only how to get the basic information that includes the name. You can retrieve data values only, or names plus data values, from *ZwEnumerateValueKey*. I showed you only how to get the name of a value.

3.5.2 Accessing Files

It's sometimes useful to be able to read and write regular disk files from inside a WDM driver. Perhaps you need to download a large amount of microcode to your hardware, or perhaps you need to create your own extensive log of information for some purpose. There's a set of *ZwXxx* routines to help you do these things.

File access via the *ZwXxx* routines require you be running at PASSIVE_LEVEL (see the next chapter) in a thread that can safely be suspended. In practice, the latter requirement means that you must not have disabled Asynchronous Procedure Calls (APCs) by calling *KeEnterCriticalRegion*. You'll read in the next chapter that some synchronization primitives require you to raise the IRQL above *PASSIVE_LEVEL* or to disable APCs. Just bear in mind that those synchronization primitives and file access are incompatible.

The first step in accessing a disk file is to open a handle by calling *ZwCreateFile*. The full description of this function in the DDK is relatively complex because of all the ways in which it can be used. I'm going to show you two simple scenarios, however, that are useful if you just want to read or write a file whose name you already know.

Sample Code
The *FILEIO* sample driver in the companion content illustrates calls to some of the *ZwXxx* functions discussed in this section. This particular sample is valuable because it provides workarounds for the platform

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 94

Incompatibilities mentioned at the end of this chapter.

Opening an Existing File for Reading

To open an existing file so that you can read it, follow this example:

```
NTSTATUS status;
OBJECT ATTRIBUTES oa;
IO STATUS BLOCK iostatus;
HANDLE hfile;                 // the output from this process
PUNICODE STRING pathname;     // you've been given this

InitializeObjectAttributes(&oa, pathname,
  OBJ CASE INSENSITIVE | OBJ KERNEL HANDLE, NULL, NULL);
status = ZwCreateFile(&hfile, GENERIC READ, &oa, &iostatus,
  NULL, 0, FILE SHARE READ, FILE OPEN,
  FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
```

Creating or Rewriting a File

To create a new file, or to open and truncate to zero length an existing file, replace the call to *ZwCreateFile* in the preceding fragment with this one:

```
status = ZwCreateFile(&hfile, GENERIC WRITE, &oa, &iostatus,
  NULL, FILE ATTRIBUTE NORMAL, 0, FILE OVERWRITE IF,
  FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
```

In these fragments, we set up an *OBJECT_ATTRIBUTES* structure whose main purpose is to point to the full pathname of the file we're about to open. We specify the *OBJ_CASE_INSENSITIVE* attribute because the Win32 file system model doesn't treat case as significant in a pathname. We specify *OBJ_KERNEL_HANDLE* for the same reason we did so in the registry example shown earlier in this chapter. Then we call *ZwCreateFile* to open the handle.

The first argument to *ZwCreateFile(&hfile)* is the address of the HANDLE variable where ZwCreateFile will return the handle it creates. The second argument (*GENERIC_READ or GENERIC_WRITE*) specifies the access we need to the handle to perform either reading or writing. The third argument (&oa) is the address of the *OBJECT_ATTRIBUTES* structure containing the name of the file. The fourth argument points to an *IO_STATUS_BLOCK* that will receive a disposition code indicating how ZwCreateFile actually implemented the operation we asked it to perform. When we open a read-only handle to an existing file, we expect the Status field of this structure to end up equal to *FILE_OPENED*. When we open a write-only handle, we expect it to end up equal to *FILE_OVERWRITTEN* or *FILE_CREATED*, depending on whether the file did or didn't already exist. The fifth argument (*NULL*) can be a pointer to a 64-bit integer that specifies the initial allocation size for the file. This argument matters only when you create or overwrite a file, and omitting it as I did here means that the file grows from zero length as you write data. The sixth argument (0 or *FILE_ATTRIBUTE_NORMAL*) specifies file attribute flags for any new file that you happen to create. The seventh argument (*FILE_SHARE_READ* or *0*) specifies how the file can be shared by other threads. If you're opening for input, you can probably tolerate having other threads read the file simultaneously. If you're opening for sequential output, you probably don't want other threads trying to access the file at all.

The eighth argument (*FILE_OPEN* or *FILE_OVERWRITE_IF*) indicates how to proceed if the file either already exists or doesn't. In the read-only case, I specified *FILE_OPEN* because I expected to open an existing file and wanted a failure if the file didn't exist. In the write-only case, I specified *FILE_OVERWRITE_IF* because I wanted to overwrite any existing file by the same name or create a brand-new file as necessary. The ninth argument (*FILE_SYNCHRONOUS_IO_NONALERT*) specifies additional flag bits to govern the open operation and the subsequent use of the handle. In this case, I indicated that I'm going to be doing synchronous I/O operations (wherein I expect the read or write function not to return until the I/O is complete). The tenth and eleventh arguments (*NULL* and *0*) are, respectively, an optional pointer to a buffer for extended attributes and the length of that buffer.

You expect *ZwCreateFile* to return *STATUS_SUCCESS* and to set the handle variable. You can then carry out whatever read or write operations you please by calling *ZwReadFile* or *ZwWriteFile*, and then you close the handle by calling *ZwClose*:

```
ZwClose(hfile);
```

You can perform synchronous or asynchronous reads and writes, depending on the flags you specified to *ZwCreateFile*. In the simple scenarios I've outlined, you would do synchronous operations that don't return until they've completed. For example:

```
PVOID buffer;
ULONG bufsize;
status = ZwReadFile(hfile, NULL, NULL, NULL, &iostatus, buffer,
  bufsize, NULL, NULL);
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 95

```
            -or-

status = ZwWriteFile(hfile, NULL, NULL, NULL, &iostatus, buffer,
  bufsize, NULL, NULL);
```

These calls are analogous to a nonoverlapped *ReadFile* or *WriteFile* call from user mode. When the function returns, you might be interested in *iostatus.Information*, which will hold the number of bytes transferred by the operation.

Scope of Handles
Each process has a private handle table that associates numeric handles with pointers to kernel objects. When you open a handle using *ZwCreateFile*, or *NtCreateFile*, that handle belongs to the then-current process, unless you use the *OBJ_KERNEL_HANDLE* flag. A process-specific handle will go away if the process terminates. Moreover, if you use the handle in a different process context, you'll be indirectly referencing whatever object (if any) that that handle corresponds to in the other process. A kernel handle, on the other hand, is kept in a global table that doesn't disappear until the operating system shuts down and can be used without ambiguity in any process.

If you plan to read an entire file into a memory buffer, you'll probably want to call *ZwQueryInformationFile* to determine the total length of the file:

```
FILE_STANDARD_INFORMATION si;
ZwQueryInformationFile(hfile, &iostatus, &si, sizeof(si),
  FileStandardInformation);
ULONG length = si.EndOfFile.LowPart;
```

Timing of File Operations
You'll be likely to want to read a disk file in a WDM driver while you're initializing your device in response to an *IRP_MN_START_DEVICE* request. (See Chapter 6.) Depending on where your device falls in the initialization sequence, you might or might not have access to files using normal pathnames like \??\C:\dir\file.ext. To be safe, put your data files into some directory below the system root directory and use a filename like \SystemRoot\dir\file.ext. The *SystemRoot* branch of the namespace is always accessible since the operating system has to be able to read disk files to start up.

Sample Code
The RESOURCE sample combines several of the ideas discussed in this chapter to illustrate how to access data in a standard resource script from within a driver. This is not especially easy to do, as you'll see if you examine the sample code.

3.5.3 Floating-Point Calculations

There are times when integer arithmetic just isn't sufficient to get your job done and you need to perform floating-point calculations. On an Intel processor, the math coprocessor is also where Multimedia Extensions (MMX) instructions execute. Historically, there have been two problems with drivers carrying out floating-point calculations. The operating system will emulate a missing coprocessor, but the emulation is expensive and normally requires a processor exception to trigger it. Handling exceptions, especially at elevated IRQLs, can be difficult in kernel mode. Additionally, on computers that have hardware coprocessors, the CPU architecture might require a separate expensive operation to save and restore the coprocessor state during context switches. Therefore, conventional wisdom has forbidden kernel-mode drivers from using floating-point calculations.

Windows 2000 and later systems provide a way around past difficulties. First of all, a system thread—see Chapter 14—running at or below *DISPATCH_LEVEL* is free to use the math coprocessor all it wants. In addition, a driver running in an arbitrary thread context at or below *DISPATCH_LEVEL* can use these two system calls to bracket its use of the math coprocessor:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KFLOATING_SAVE FloatSave;
NTSTATUS status = KeSaveFloatingPointState(&FloatSave);
if (NT_SUCCESS(status))
. {
:
  KeRestoreFloatingPointState(&FloatSave);
  }
```

These calls, which must be paired as shown here, save and restore the "nonvolatile" state of the math coprocessor for the current CPU—that is, all the state information that persists beyond a single operation. This state information includes registers, control words, and so on. In some CPU architectures, no actual work might occur because the architecture inherently allows any process to perform floating-point operations. In other architectures, the work involved in saving and restoring state information can be quite substantial. For this reason, Microsoft recommends that you avoid using floating-point calculations in

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 96

a kernel-mode driver unless necessary.

Sample Code
The FPUTEST sample illustrates one way to use floating-point and MMX instructions in a WDM driver.

What happens when you call *KeSaveFloatingPointState* depends, as I said, on the CPU architecture. To give you an idea, on an Intel-architecture processor, this function saves the entire floating-point state by executing an *FSAVE* instruction. It can save the state information either in a context block associated with the current thread or in an area of dynamically allocated memory. It uses the opaque *FloatSave* area to record meta-information about the saved state to allow *KeRestoreFloatingPointState* to correctly restore the state later.

KeSaveFloatingPointState will fail with *STATUS_ILLEGAL_FLOAT_CONTEXT* if no real coprocessor is present. (All CPUs of a multi-CPU computer must have coprocessors, or else none of them can, by the way.) Your driver will therefore need alternative code to carry out whatever calculations you had in mind, or else you'll want to decline to load (by failing *DriverEntry*) if the computer doesn't have a coprocessor.

NOTE
You can call *ExIsProcessorFeaturePresent* to check up on various floating-point capabilities. Since this function isn't present in Windows 98/Me, you'll also need to ship WDMSTUB with your driver. Consult Appendix A for more information about this and related system incompatibilities.

3.5.4 Making Debugging Easier

My drivers always have bugs. Maybe you're as unlucky as I am. If so, you'll find yourself spending lots of time with a debugger trying to figure out what your code is doing or not doing correctly or incorrectly. I won't discuss the potentially divisive subject of which debugger is best or the noncontroversial but artistic subject of how to debug a driver. But you can do some things in your driver code that will make your life easier.

Checked and Free Builds

When you build your driver, you select either the *checked* or the *free* build environment. In the checked build environment, the preprocessor symbol DBG equals 1, whereas it equals 0 in the free build environment. So one of the things you can do in your own code is to provide additional code that will take effect only in the checked build:

```
#if DBG
  <extra debugging code>
#endif
```

The *KdPrint* macro

One of the most useful debugging techniques ever invented is to simply print messages from time to time. I used to do this when I was first learning to program (in FORTRAN on a computer made out of vacuum tubes, no less), and I still do it today. *DbgPrint* is a kernel-mode service routine you can call to display a formatted message in whatever output window your debugger provides. Another way to see the output from *DbgPrint* calls is to download the *DebugView* utility from http://www.sysinternals.com. Instead of directly referencing *DbgPrint* in your code, it's often easier to use the macro named *KdPrint*, which calls *DbgPrint* if DBG is true and generates no code at all if DBG is false:

```
KdPrint((DRIVERNAME " - KeReadProgrammersMind failed - %X\n",  status));
```

You use two sets of parentheses with *KdPrint* because of the way it's defined. The first argument is a string with %-escapes where you want to substitute values. The second, third, and following arguments provide the values to go with the %-escapes. The macro expands into a call to *DbgPrint*, which internally uses the standard run-time library routine _vsnprintf to format the string. You can, therefore, use the same set of %-escape codes that are available to application programs that call this routine but not the escapes for floating-point numbers.

In all of my drivers, I define a constant named DRIVERNAME like this:

```
#define DRIVERNAME "xxx"
```

where xxx is the name of the driver. Recall that the compiler sees two adjacent string constants as a single constant. Using this particular trick allows me to cut and paste entire subroutines, including their *KdPrint* calls, from one driver to another without needing to make source changes.

The ASSERT macro

Another useful debugging technique relies on the ASSERT macro:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 97

```
ASSERT(1 + 1 == 2);
```

In the checked build of your driver, *ASSERT* generates code to evaluate the Boolean expression. If the expression is false, *ASSERT* will try to halt execution in the debugger so that you can see what's going on. If the expression is true, your program continues executing normally. Kernel debuggers will halt when *ASSERT* failures happen, even in the retail build of the operating system, by the way.

IMPORTANT
An ASSERT failure in a retail build of the operating system with no kernel debugger running generates a bug check.

The Driver Verifier

☑ The Driver Verifier is part of the checked and free builds of the operating system and is fast becoming one of Microsoft's major tools for checking driver quality. You can launch Driver Verifier from the Start Menu, whereupon you'll be presented with a series of wizard pages. Here is a bit of a road map to guide you through these pages the first time.

Figure 3-14 illustrates the initial wizard page. I recommend checking the Create Custom Settings (For Code Developers) option. This choice will allow you to specify in detail which Driver Verifier options you want to engage.

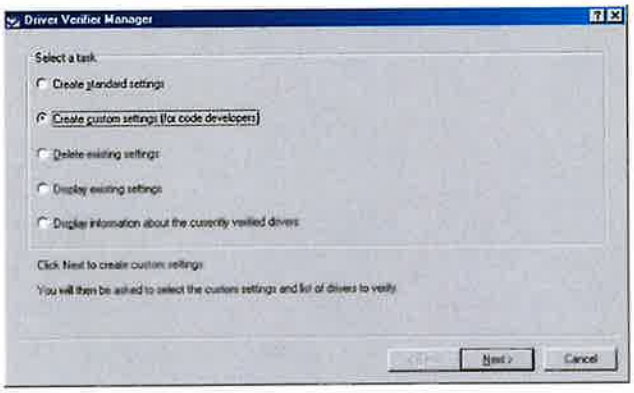


Figure 3-14. *Initial Driver Verifier wizard page*

After making my recommended choice from the first page, you'll be presented with a second page (see Figure 3-15). Here, I recommend checking the Select Individual Settings From A Full List option.

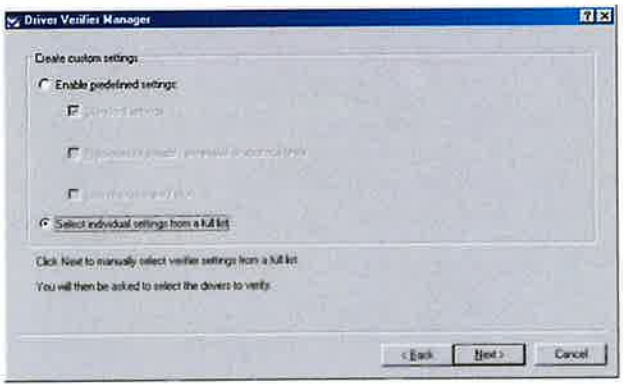


Figure 3-15. *Second Driver Verifier wizard page*

The next wizard page (see Figure 3-16) allows you to specify the verifier settings you desire. The specified checks are in addition to a number of checks that the Driver Verifier makes automatically.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 98

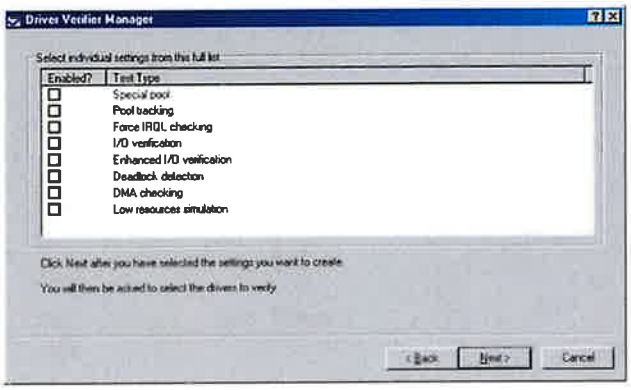


Figure 3-16. *Driver Verifier's custom settings wizard page*

The choices available at the time I'm writing this are as follows:

- *Special Pool* forces all memory allocations from verified drivers to be made from the special pool. As described earlier in this chapter, such allocations are at the end (or start) of a page, so that storing after (or before) the allocated memory leads to an immediate bug check.
- *Pool Tracking* causes the Driver Verifier to track memory allocations made by verified drivers. You can see statistics about memory usage as it changes from time to time. The Driver Verifier also ensures that all allocations are freed when the verified drivers unload to help you catch memory leaks.
- *Force IRQL Checking* essentially causes paged memory to be flushed whenever a verified driver raises *IRQL to DISPATCH_LEVEL* or above. This action helps find places where the driver is incorrectly accessing paged memory. The system runs quite slowly when this option is turned on.
- *I/O Verification* causes the Driver Verifier to make basic checks on how a driver handles IRPs that it creates or forwards to other drivers.
- *Enhanced I/O Verification* attempts to flush out driver errors in boundary cases, such as completing PnP and Power IRPs incorrectly, making assumptions about the order in which the PnP Manager loads drivers, and so on. Some of these tests occur when the driver initially starts, by the way, which can prevent the system from starting.
- *Deadlock* Detection creates a graph of the locking hierarchy for spin locks, mutexes, and fast mutexes used by verified drivers in order to detect potential deadlocks.
- *DMA Checking* ensures that verified drivers perform DMA using the methods prescribed by the DDK.
- *Low Resources Simulation* involves randomly failing pool allocations from verified drivers, beginning seven minutes after the system starts. The purpose of these failures is to ensure that drivers test the return value from pool allocation calls.

You can use a special procedure, described in the DDK, to activate checks on a SCSI miniport driver.

NOTE
There can be interactions between the options you specify. At the present time, for example, asking for DMA checking or deadlock detection turns off enhanced I/O verification.

Note too that the Driver Verifier is evolving rapidly even as we speak. Consult the DDK you happen to be working with for up-to-date information.

After you select the verifier options you want, you will see one final wizard page (see Figure 3-17). This page allows you to specify which drivers you want verified by checking boxes in a list. After making that selection, you'll need to restart the computer because many of the Driver Verifier checks require boot-time initialization. When I'm debugging one of my own drivers, I find it most convenient to not have my driver loaded when the restart occurs. My driver therefore won't already be in the list, and I'll have to add it via the Add Currently Not Loaded Driver(s) To The List button.

Driver Verifier failures are bug checks, by the way. You will need to be running a kernel debugger or to analyze the crash dump after the fact to isolate the cause of the failure.

Don't Ship the Checked Version!

Incidentally, I and every other user of kernel debuggers would greatly prefer that you not ship the debug version of your driver. It will probably contain a bunch of ASSERT statements that will go off while we're looking for our own bugs, and it will probably also print a lot of messages that will obscure the messages from our drivers. I recall a vendor who shipped a debug

IPR2024-00292
KASPERSKY EX 1008 Page 99

driver for a popular network card. This particular driver logs every packet it receives over the wire. Nowadays I carry with me to consulting gigs a little driver that patches their driver to eliminate the spew. And I buy other vendors' network cards.

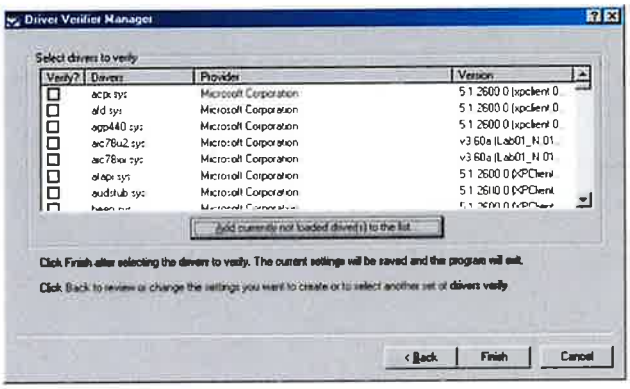


Figure 3-17. *Driver selection page for Driver Verifier.*

3.6 Windows 98/Me Compatibility Notes

3.6.1 File I/O

The *ZwXxx* routines for accessing disk files don't work well in Windows 98/Me because of two basic problems—one from the architecture of Windows and the other from what looks like an ordinary bug in the original release of Windows 98.

The first problem with file access has to do with the order in which Windows 98/Me initializes various virtual device drivers. The Configuration Manager (CONFIGMG.VXD) initializes before the Installable File System Manager (IFSMGR.VXD). WDM drivers for devices that exist at start-up time receive their *IRP_MN_START_DEVICE* requests during CONFIGMG's initialization phase. But since IFSMGR hasn't initialized at that point, it's not possible to perform file I/O operations by using *ZwCreateFile* and the other functions discussed earlier in this chapter. Furthermore, there's no way for a WDM driver to defer handling *IRP_MN_START_DEVICE* until file system functionality becomes available. If you don't have a debugger such as Soft-Ice/W running, the symptom you will see is a blue screen complaining of a Windows Protection error while initializing CONFIGMG.

A second and more crippling problem existed in the July 1998 retail release of Windows 98. The problem had to do with the validity checking that *ZwReadFile*, *ZwWriteFile*, and *ZwQueryInformationFile* perform on their arguments. If you supply an *IO_STATUS_BLOCK* in kernel-mode memory (and there's basically no way to do anything else), these functions probe a virtual address that doesn't exist. In this original release, the resulting page fault was caught by a structured exception handler and resulted in you getting back *STATUS_ACCESS_VIOLATION* even when you did everything right. I don't know of any workaround for this except by using the technique in the FILEIO sample. The problem was fixed in Windows 98, Second Edition, by the way.

The FILEIO sample in the companion content illustrates a way past these Windows 98/Me difficulties. FILEIO makes a run-time decision whether to call the *ZwXxx* functions or instead to call VxD services to perform file operations.

3.6.2 Floating Point

Floating-point operations are permissible in Windows 98/Me WDM drivers, but with some important restrictions relative to the situation in Windows XP:

- You can do floating-point (including MMX) operations in a WDM driver only in a system thread. System threads include those you create yourself by calling PsCreateSystemThread and system worker threads. Note that work item callbacks occur in a system worker thread, so you can do floating point in such a callback.
- You should do floating-point operations only at PASSIVE_LEVEL. (DISPATCH_LEVEL corresponds to VxD hardware interrupt handling in Windows 98/Me.)

The DDK cautions that you should not attempt to circumvent these rules. For example, you might be tempted to use *KeSaveFloatingPointState* and *KeRestoreFloatingPointState* in an IOCTL handler despite the explicit prohibition on doing so, or to manually save and restore the FPU state. What can happen to you is this: if an exception is pending in the coprocessor when you initially save the floating-point state, that exception will be taken when you restore the state. The kernel cannot correctly handle that exception. There is no workaround for this problem, which is inherent in the Intel processor design and the way VMCPD.VXD works.

Note that the FPUTEST sample program obeys these rules by refusing to work in Windows 98/Me.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 100

Chapter 4

Synchronization

Microsoft Windows XP is a multitasking operating system that can run in a symmetric multiprocessor environment. It's not my purpose here to provide a rigorous description of the multitasking capabilities of Microsoft Windows XP; one good place to get more information is David Solomon and Mark Russinovich's *Inside Windows 2000*, Third Edition (Microsoft Press, 2000). All we need to understand as driver writers is that our code executes in the context of one *thread* or another (and the thread context can change from one invocation of our code to another) and that the exigencies of multitasking can yank control away from us at practically any moment. Furthermore, true simultaneous execution of multiple threads is possible on a multiprocessor machine. In general, we need to assume two worst-case scenarios:

1. The operating system can preempt any subroutine at any moment for an arbitrarily long period of time, so we cannot be sure of completing critical tasks without interference or delay.

2. Even if we take steps to prevent preemption, code executing simultaneously on another CPU in the same computer can interfere with our code—it's even possible that the same set of instructions belonging to one of our programs could be executing in parallel in the context of two different threads.

Windows XP allows you to solve these general synchronization problems by using a variety of synchronization primitives. The system prioritizes the handling of hardware and software interrupts with the *interrupt request level* (IRQL). The system offers a variety of synchronization primitives. Some of these primitives are appropriate at times when you can safely block and unblock threads. One primitive, the *spin lock*, allows you to synchronize access to shared resources even at times when thread blocking wouldn't be allowed because of the priority level at which a program runs.

4.1 An Archetypal Synchronization Problem

A hackneyed example will motivate this discussion. Suppose your driver has a static integer variable that you use for some purpose, say, to count the number of I/O requests that are currently outstanding:

```
static LONG lActiveRequests;
```

Suppose further that you increment this variable when you receive a request and decrement it when you later complete the request:

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  ++lActiveRequests;
  ... // process PNP request
  --lActiveRequests;
  }
```

I'm sure you recognize already that a counter such as this one ought not to be a static variable: it should be a member of your device extension so that each device object has its own unique counter. Bear with me, and pretend that your driver always manages only a single device. To make the example more meaningful, suppose finally that a function in your driver will be called when it's time to delete your device object. You might want to defer the operation until no more requests are outstanding, so you might insert a test of the counter:

```
NTSTATUS HandleRemoveDevice(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  if (lActiveRequests)
    <wait for all requests to complete>
  IoDeleteDevice(fdo);
  }
```

This example describes a real problem, by the way, which we'll tackle in Chapter 6 in our discussion of Plug and Play (PnP) requests. The I/O Manager can try to remove one of our devices at a time when requests are active, and we need to guard against that by keeping some sort of counter. I'll show you in Chapter 6 how to use *IoAcquireRemoveLock* and some related functions to solve the problem.

A horrible synchronization problem lurks in the code fragments I just showed you, but it becomes apparent only if you look behind the increment and decrement operations inside *DispatchPnp*. On an x86 processor, the compiler might implement them

IPR2024-00292
KASPERSKY EX 1008 Page 101

using these instructions:

```
; ++lActiveRequests;
  mov eax, lActiveRequests
  add eax, 1
  mov lActiveRequests, eax
:
:
; --lActiveRequests;
  mov eax, lActiveRequests
  sub eax, 1
  mov lActiveRequests, eax
```

To expose the synchronization problem, let's consider first what might go wrong on a single CPU. Imagine two threads that are both trying to advance through *DispatchPnp* at roughly the same time. We know they're not both executing truly simultaneously because we have only a single CPU for them to share. But imagine that one of the threads is executing near the end of the function and manages to load the current contents of *lActiveRequests* into the EAX register just before the other thread preempts it. Suppose *lActiveRequests* equals 2 at that instant. As part of the thread switch, the operating system saves the EAX register (containing the value 2) as part of the outgoing thread's context image somewhere in main memory.

NOTE
The point being made in the text isn't limited to thread preemption that occurs as a result of a time slice expiring. Threads can also involuntarily lose control because of page faults, changes in CPU affinity, or priority changes instigated by outside agents. Think, therefore, of *preemption* as being an all-encompassing term that includes all means of giving control of a CPU to another thread without explicit permission from the currently running thread.

Now imagine that the other thread manages to get past the incrementing code at the beginning of *DispatchPnp*. It will increment *lActiveRequests* from 2 to 3 (because the first thread never got to update the variable). If the first thread preempts this other thread, the operating system will restore the first thread's context, which includes the value 2 in the EAX register. The first thread now proceeds to subtract 1 from EAX and store the result back in *lActiveRequests*. At this point, *lActiveRequests* contains the value 1, which is incorrect. Somewhere down the road, we might prematurely delete our device object because we've effectively lost track of one I/O request.

Solving this particular problem is easy on an x86 computer—we just replace the load/add/store and load/subtract/store instruction sequences with atomic instructions:

```
; ++lActiveRequests;
  inc lActiveRequests
:
; --lActiveRequests;
  dec lActiveRequests
```

On an Intel x86, the INC and DEC instructions cannot be interrupted, so there will never be a case in which a thread can be preempted in the middle of updating the counter. As it stands, though, this code still isn't safe in a multiprocessor environment because INC and DEC are implemented in several microcode steps. It's possible for two different CPUs to be executing their microcode just slightly out of step such that one of them ends up updating a stale value. The multi-CPU problem can also be avoided in the x86 architecture by using a LOCK prefix:

```
; ++lActiveRequests;
  lock inc lActiveRequests
:
; --lActiveRequests;
  lock dec lActiveRequests
```

The LOCK instruction prefix locks out all other CPUs while the microcode for the current instruction executes, thereby guaranteeing data integrity.

Not all synchronization problems have such an easy solution, unfortunately. The point of this example isn't to demonstrate how to solve one simple problem on one of the platforms where Windows XP runs but rather to illustrate the two sources of difficulty: preemption of one thread by another in the middle of a state change and simultaneous execution of conflicting state-change operations. We can avoid difficulty by judiciously using synchronization primitives, such as mutual exclusion objects, to block other threads while our thread accesses shared data. At times when thread blocking is impermissible, we can avoid preemption by using the IRQL priority scheme, and we can prevent simultaneous execution by judiciously using spin locks.

IPR2024-00292
KASPERSKY EX 1008 Page 102

4.2 Interrupt Request Level

Windows XP assigns an interrupt request level to each hardware interrupt and to a select few software events. Each CPU has its own IRQL. We label the different IRQL levels with names such as *PASSIVE_LEVEL*, *APC_LEVEL*, and so on. Figure 4-1 illustrates the range of IRQL values for the x86 platform. (In general, the numeric values of IRQL depend on which platform you're talking about.) Most of the time, the computer executes in user mode at *PASSIVE_LEVEL*. All of your knowledge about how multitasking operating systems work applies at *PASSIVE_LEVEL*. That is, the scheduler may preempt a thread at the end of a time slice or because a higher-priority thread has become eligible to run. Threads can also voluntarily block while they wait for events to occur.

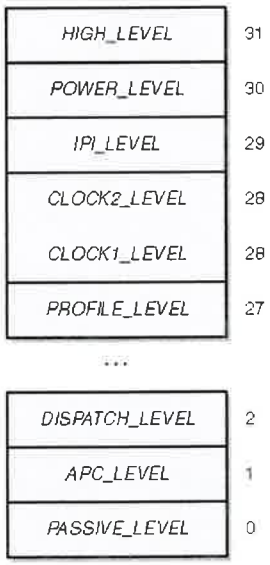


Figure 4-1. *Interrupt request levels.*

When an interrupt occurs, the kernel raises the IRQL on the interrupting CPU to the level associated with that interrupt. The *activity* of processing an interrupt can be—uh, *interrupted*—to process an interrupt at a higher IRQL but never to process an interrupt at the same or a lower IRQL. I'm sorry to use the word interrupt in two slightly different ways here. I struggled to find a word to describe the temporary suspension of an activity that wouldn't cause confusion with thread preemption, and that was the best choice.

What I just said is sufficiently important to be enshrined as a rule:

An activity on a given CPU can be interrupted only by an activity that executes at a higher IRQL.

You have to read this rule the way the computer does. Expiration of a time slice eventually invokes the thread scheduler at *DISPATCH_LEVEL*. The scheduler can then make a different thread current. When the IRQL returns to *PASSIVE_LEVEL*, a different thread is running. But it's still true that the first PASSIVE_LEVEL activity wasn't interrupted by the second *PASSIVE_LEVEL* activity. I thought this interpretation was incredible hair-splitting until it was pointed out to me that this arrangement allows a thread running at *APC_LEVEL* to be preempted by a different thread running at *PASSIVE_LEVEL*. Perhaps a more useful statement of the rule is this one:

An activity on a given CPU can be interrupted only by an activity that executes at a higher IRQL. An activity at or above DISPATCH_LEVEL cannot be suspended to perform another activity at or below the then-current IRQL.

Since each CPU has its own IRQL, it's possible for any CPU in a multiprocessor computer to run at an IRQL that's less than or equal to the IRQL of any other CPU. In the next major section, I'll tell you about spin locks, which combine the within-a-CPU synchronizing behavior of an IRQL with a multiprocessor lockout mechanism. For the time being, though, I'm talking just about what happens on a single CPU.

To repeat something I just said, user-mode programs execute at *PASSIVE_LEVEL*. When a user-mode program calls a function in the native API, the CPU switches to kernel mode but continues to run at *PASSIVE_LEVEL* in the same thread context. Many times, the native API function calls an entry point in a driver without raising the IRQL. Driver dispatch routines for most types of I/O request packet (IRP) execute at *PASSIVE_LEVEL*. In addition, certain driver subroutines, such as *DriverEntry* and *AddDevice*, execute at *PASSIVE_LEVEL* in the context of a system thread. In all of these cases, the driver code can be preempted just as a user-mode application can be.

Certain common driver routines execute at *DISPATCH_LEVEL*, which is higher than *PASSIVE_LEVEL*. These include the StartIo routine, deferred procedure call (DPC) routines, and many others. What they have in common is a need to access fields

IPR2024-00292
KASPERSKY EX 1008 Page 103

in the device object and the device extension without interference from driver dispatch routines and one another. When one of these routines is running, the rule stated earlier guarantees that no thread can preempt it on the same CPU to execute a driver dispatch routine because the dispatch routine runs at a lower IRQL. Furthermore, no thread can preempt it to run another of these special routines because that other routine will run at the same IRQL.

NOTE
Dispatch routine and DISPATCH_LEVEL unfortunately have similar names. Dispatch routines are so called because the I/O Manager dispatches I/O requests to them. DISPATCH_LEVEL is so called because it's the IRQL at which the kernel's thread dispatcher originally ran when deciding which thread to run next. (The thread dispatcher runs at SYNCH_LEVEL, if you care. This is the same as DISPATCH_LEVEL on a uniprocessor machine, if you really care.)

Between *DISPATCH_LEVEL* and *PROFILE_LEVEL* is room for various hardware interrupt levels. In general, each device that generates interrupts has an IRQL that defines its interrupt priority vis-à-vis other devices. A WDM driver discovers the IRQL for its interrupt when it receives an *IRP_MJ_PNP* request with the minor function code *IRP_MN_START_DEVICE*. The device's interrupt level is one of the many items of configuration information passed as a parameter to this request. We often refer to this level as the device *IRQL*, or *DIRQL* for short. DIRQL isn't a single request level. Rather, it's the IRQL for the interrupt associated with whichever device is under discussion at the time.

The other IRQL levels have meanings that sometimes depend on the particular CPU architecture. Since those levels are used internally by the kernel, their meanings aren't especially germane to the job of writing a device driver. The purpose of *APC_LEVEL*, for example, is to allow the system to schedule an *asynchronous procedure call* (APC), which I'll describe in detail later in this chapter. Operations that occur at *HIGH_LEVEL* include taking a memory snapshot just prior to hibernating the computer, processing a bug check, handling a totally spurious interrupt, and others. I'm not going to attempt to provide an exhaustive list here because, as I said, you and I don't really need to know all the details.

To summarize, drivers are normally concerned with three interrupt request levels:

- *PASSIVE_LEVEL*, at which many dispatch routines and a few special routines execute
- *DISPATCH_LEVEL*, at which StartIo and DPC routines execute
- *DIRQL*, at which an interrupt service routine executes

4.2.1 *IRQL* in Operation

To illustrate the importance of IRQL, refer to Figure 4-2, which illustrates a possible time sequence of events on a single CPU. At the beginning of the sequence, the CPU is executing at *PASSIVE_LEVEL*. At time t1, an interrupt arrives whose service routine executes at IRQL-1, one of the levels between *DISPATCH_LEVEL* and *PROFILE_LEVEL*. Then, at time t2, another interrupt arrives whose service routine executes at IRQL-2, which is less than IRQL-1. Because of the rule already discussed, the CPU continues servicing the first interrupt. When the first interrupt service routine completes at time t3, it might request a DPC. DPC routines execute at *DISPATCH_LEVEL*. Consequently, the highest priority pending activity is the service routine for the second interrupt, which therefore executes next. When it finishes at t4, assuming nothing else has occurred in the meantime, the DPC will run at *DISPATCH_LEVEL*. When the DPC routine finishes at t5, IRQL can drop back to *PASSIVE_LEVEL*.

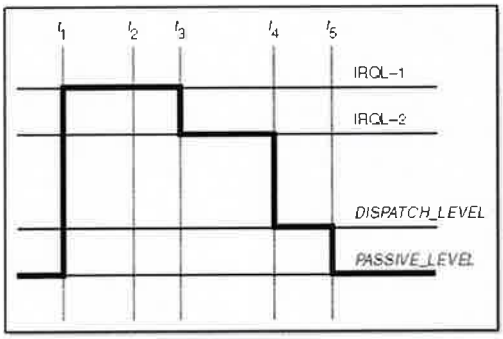


Figure 4-2. *Interrupt priority in action.*

4.2.2 IRQL Compared with Thread Priorities

Thread priority is a very different concept from IRQL. Thread priority controls the actions of the scheduler in deciding when to preempt running threads and what thread to start running next. The only "priority" that means anything at IRQLs above *APC_LEVEL* is IRQL itself, and it controls which programs can execute rather than the thread context within which they execute.

IPR2024-00292
KASPERSKY EX 1008 Page 104

4.2.3 IRQL and Paging

One consequence of running at elevated IRQL is that the system becomes incapable of servicing page faults. The rule this fact implies is simply stated:

Code executing at or above *DISPATCH_LEVEL* must not cause page faults.

One implication of this rule is that any of the subroutines in your driver that execute at or above DISPATCH_LEVEL must be in nonpaged memory. Furthermore, all the data you access in such a subroutine must also be in nonpaged memory. Finally, as IRQL rises, fewer and fewer kernel-mode support routines are available for your use.

- The DDK documentation explicitly states the IRQL restrictions on support routines. For example, the entry for *KeWaitForSingleObject* indicates two restrictions:
- The caller must be running at or below *DISPATCH_LEVEL*.

If a nonzero timeout period is specified in the call, the caller must be running strictly below *DISPATCH_LEVEL*.

Reading between the lines, what is being said here is this: if the call to *KeWaitForSingleObject* might conceivably block for any period of time (that is, you've specified a nonzero timeout), you must be below *DISPATCH_LEVEL*, where thread blocking is permitted. If all you want to do is check to see whether an event has been signaled, however, you can be at *DISPATCH_LEVEL*. You can't call this routine at all from an interrupt service routine or other routine running above *DISPATCH_LEVEL*.

For the sake of completeness, it's well to point out that the rule against page faults is really a rule prohibiting any sort of hardware exception, including page faults, divide checks, bounds exceptions, and so on. Software exceptions, like quota violations and probe failures on nonpaged memory, are permissible. Thus, it's acceptable to call *ExAllocatePoolWithQuota* to allocate nonpaged memory at *DISPATCH_LEVEL*.

4.2.4 Implicitly Controlling IRQL

Most of the time, the system calls the routines in your driver at the correct IRQL for the activities you're supposed to carry out. Although I haven't discussed many of these routines in detail, I want to give you an example of what I mean. Your first encounter with a new I/O request occurs when the I/O Manager calls one of your dispatch routines to process an IRP. The call usually occurs at *PASSIVE_LEVEL* because you might need to block the calling thread and you might need to call any support routine at all. You can't block a thread at a higher IRQL, of course, and *PASSIVE_LEVEL* is the level at which there are the fewest restrictions on the support routines you can call.

NOTE
Driver dispatch routines usually execute at PASSIVE_LEVEL but not always. You can designate that you want to receive IRP_MJ_POWER requests at *DISPATCH_LEVEL* by setting the *DO_POWER_INRUSH* flag, or by clearing the *DO_POWER_PAGABLE* flag, in a device object. Sometimes a driver architecture requires that other drivers be able to send certain IRPs at *DISPATCH_LEVEL*. The USB bus driver, for example, accepts data transfer requests at *DISPATCH_LEVEL* or below. A standard serial-port driver accepts any read, write, or control operation at or below *DISPATCH_LEVEL*.

If your dispatch routine queues the IRP by calling *IoStartPacket*, your next encounter with the request will be when the I/O Manager calls your *StartIo* routine. This call occurs at *DISPATCH_LEVEL* because the system needs to access the queue of I/O requests without interference from the other routines that are inserting and removing IRPs from the queue. As I'll discuss later in this chapter, queue access occurs under protection of a spin lock, and that carries with it execution at *DISPATCH_LEVEL*.

Later on, your device might generate an interrupt, whereupon your interrupt service routine will be called at DIRQL. It's likely that some registers in your device can't safely be shared. If you access those registers only at DIRQL, you can be sure that no one can interfere with your interrupt service routine (ISR) on a single-CPU computer. If other parts of your driver need to access these crucial hardware registers, you would guarantee that those other parts execute only at DIRQL. The *KeSynchronizeExecution* service function helps you enforce that rule, and I'll discuss it in Chapter 7 in connection with interrupt handling.

Still later, you might arrange to have a DPC routine called. DPC routines execute at *DISPATCH_LEVEL* because, among other things, they need to access your IRP queue to remove the next request from a queue and pass it to your *StartIo* routine. You call the *IoStartNextPacket* service routine to extract the next request from the queue, and it must be called at *DISPATCH_LEVEL*. It might call your *StartIo* routine before returning. Notice how neatly the IRQL requirements dovetail here: queue access, the call to *IoStartNextPacket*, and the possible call to *StartIo* are all required to occur at *DISPATCH_LEVEL*, and that's the level at which the system calls the DPC routine.

Although it's possible for you to explicitly control IRQL (and I'll explain how in the next section), there's seldom any reason to do so because of the correspondence between your needs and the level at which the system calls you. Consequently, you don't need to get hung up on which IRQL you're executing at from moment to moment: it's almost surely the correct level for the work you're supposed to do right then.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 105

4.2.5 Explicitly Controlling IRQL

When necessary, you can raise and subsequently lower the IRQL on the current processor by calling *KeRaiseIrql* and *KeLowerIrql*. For example, from within a routine running at *PASSIVE_LEVEL*:

```
1
KIRQL oldirql;
2
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
3
KeRaiseIrql(DISPATCH_LEVEL, &oldirql);
:
4
KeLowerIrql(oldirql);
```

1. KIRQL is the typedef name for an integer that holds an IRQL value. We'll need a variable to hold the current IRQL, so we declare it this way.
2. This *ASSERT* expresses a necessary condition for calling *KeRaiseIrql*: the new IRQL must be greater than or equal to the current level. If this relation isn't true, *KeRaiseIrql* will *bugcheck* (that is, report a fatal error via a blue screen of death).
3. *KeRaiseIrql* raises the current IRQL to the level specified by the first argument. It also saves the current IRQL at the location pointed to by the second argument. In this example, we're raising IRQL to *DISPATCH_LEVEL* and saving the current level in *oldirql*.
4. After executing whatever code we desired to execute at elevated IRQL, we lower the request level back to its previous value by calling *KeLowerIrql* and specifying the *oldirql* value previously returned by *KeRaiseIrql*.

After raising the IRQL, you should eventually restore it to the original value. Otherwise, various assumptions made by code you call later or by the code that called you can later turn out to be incorrect. The DDK documentation says that you must always call *KeLowerIrql* with the same value as that returned by the immediately preceding call to *KeRaiseIrql*, but this information isn't exactly right. The only rule that *KeLowerIrql* actually applies is that the new IRQL must be less than or equal to the current one. You can lower the IRQL in steps if you want to.

It's a mistake (and a big one!) to lower IRQL below whatever it was when a system routine called your driver, even if you raise it back before returning. Such a break in synchronization might allow some activity to preempt you and interfere with a data object that your caller assumed would remain inviolate.

4.3 Spin Locks

To help you synchronize access to shared data in the symmetric multiprocessing world of Windows XP, the kernel lets you define any number of *spin lock* objects. To acquire a spin lock, code on one CPU executes an atomic operation that tests and then sets a memory variable in such a way that no other CPU can access the variable until the operation completes. If the test indicates that the lock was previously free, the program continues. If the test indicates that the lock was previously held, the program repeats the test-and-set in a tight loop: it "spins." Eventually the owner releases the lock by resetting the variable, whereupon one of the waiting CPUs' test-and-set operations will report the lock as free.

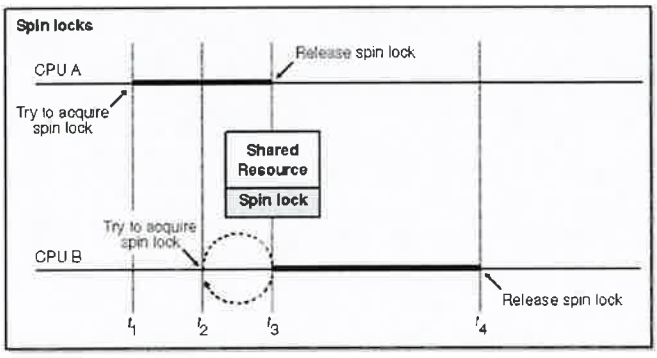


Figure 4-3. *Using a spin lock to guard a shared resource.*

Figure 4-3 illustrates the concept of using a spin lock. Suppose we have some "resource" that might be used simultaneously on two different CPUs. To make the example concrete, imagine that the resource is the *LIST_ENTRY* cell that anchors a linked list of IRPs. The list might be accessed by one or more dispatch routines, a cancel routine, a DPC routine, and perhaps others as well. Any number of these routines might be executing simultaneously on different CPUs and trying to modify the list anchor.

IPR2024-00292
KASPERSKY EX 1008 Page 106

To prevent chaos, we associate a spin lock with this "resource."

Suppose now that code executing on CPU A wants to access the shared resource at time t1. It acquires the spin lock and begins its access. Shortly afterward, at time t2, code executing on CPU B also wants to access the same resource. The CPU-B program tries to acquire the spin lock. Since CPU A currently owns the spin lock, CPU B spins in a tight loop, continually checking and rechecking the spin lock to see whether it has become free. When CPU A releases the lock at time t3, CPU B finds the lock free and claims it. Then CPU B has unfettered access to the resource. Finally, at time t4, CPU B finishes its access and releases the lock.

I want to be very clear about how a spin lock and a shared resource come to be associated. We make the association when we design the driver. We decide that we will access the resource only while owning the spin lock. The operating system isn't aware of our decision. Furthermore, we can define as many spin locks as we want, to guard as many shared resources as we want.

4.3.1 Some Facts About Spin Locks

You need to know several important facts about spin locks. First of all, if a CPU already owns a spin lock and tries to obtain it a second time, the CPU will deadlock. No usage counter or owner identifier is associated with a spin lock; somebody either owns the lock or not. If you try to acquire the lock when it's owned, you'll wait until the owner releases it. If your CPU happens to already be the owner, the code that would release the lock can never execute because you're spinning in a tight loop testing and setting the lock variable.

CAUTION
You can certainly avoid the deadlock that occurs when a CPU tries to acquire a spin lock it already owns by following this rule: make sure that the subroutine that claims the lock releases it and never tries to claim it twice, and then don't call any other subroutine while you own the lock. There's no policeman in the operating system to ensure you don't call other subroutines—it's just an engineering rule of thumb that will help you avoid an inadvertent mistake. The danger you're guarding against is that you (or some maintenance programmer who follows in your footsteps) might forget that you've already claimed a certain spin lock. I'll tell you about an ugly exception to this salutary rule in Chapter 5, when I discuss IRP cancel routines.

In addition, acquiring a spin lock raises the IRQL to *DISPATCH_LEVEL* automatically. Consequently, code that acquires a lock must be in nonpaged memory and must not block the thread in which it runs. (There is an exception in Windows XP and later systems. *KeAcquireInterruptSpinLock* raises the IRQL to the DIRQL for an interrupt and claims the spin lock associated with the interrupt.)

As an obvious corollary of the previous fact, you can request a spin lock only when you're running at or below *DISPATCH_LEVEL*. Internally, the kernel is able to acquire spin locks at an IRQL higher than *DISPATCH_LEVEL*, but you and I are unable to accomplish that feat.

Another fact about spin locks is that very little useful work occurs on a CPU that's waiting for a spin lock. The spinning happens at *DISPATCH_LEVEL* with interrupts enabled, so a CPU that's waiting for a spin lock can service hardware interrupts. But to avoid harming performance, you need to minimize the amount of work you do while holding a spin lock that some other CPU is likely to want.

Two CPUs can simultaneously hold two different spin locks, by the way. This arrangement makes sense: you associate a spin lock with a certain shared resource, or some collection of shared resources. There's no reason to hold up processing related to different resources protected by different spin locks.

As it happens, there are separate uniprocessor and multiprocessor kernels. The Windows XP setup program decides which kernel to install after inspecting the computer. The multiprocessor kernel implements spin locks as I've just described. The uniprocessor kernel realizes, however, that another CPU can't be in the picture, so it implements spin locks a bit more simply. On a uniprocessor system, acquiring a spin lock raises the IRQL to *DISPATCH_LEVEL* and does nothing else. Do you see how you still get the synchronization benefit from claiming the so-called lock in this case? For some piece of code to attempt to claim the same spin lock (or any other spin lock, actually, but that's not the point here), it would have to be running at or below *DISPATCH_LEVEL*—you can request a lock starting at or below *DISPATCH_LEVEL* only. But we already know that's impossible because, once you're above *PASSIVE_LEVEL*, you can't be interrupted by any other activity that would run at the same or a lower IRQL. Q., as we used to say in my high school geometry class, E.D.

4.3.2 Working with Spin Locks

To use a spin lock explicitly, allocate storage for a KSPIN_LOCK object in nonpaged memory. Then call *KeInitializeSpinLock* to initialize the object. Later, while running at or below DISPATCH_LEVEL, acquire the lock, perform the work that needs to be protected from interference, and then release the lock. For example, suppose your device extension contains a spin lock named *QLock* that you use for guarding access to a special IRP queue you've set up. You'll initialize this lock in your *AddDevice* function:

```
typedef struct  DEVICE EXTENSION {
  .
  .
```


IPR2024-00292
KASPERSKY EX 1008 Page 107

```
  KSPIN LOCK QLock;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;
:
:

NTSTATUS AddDevice(...)
: {
:
  PDEVICE EXTENSION pdx = ...;
: KeInitializeSpinLock(&pdx->QLock);
:
  }
```

Elsewhere in your driver, say in the dispatch function for some type of IRP, you can claim (and quickly release) the lock around some queue manipulation that you need to perform. Note that this function must be in nonpaged memory because it executes for a period of time at an elevated IRQL.

```
NTSTATUS DispatchSomething(...)
  {
  KIRQL oldirql;
  PDEVICE EXTENSION pdx = ...;
1
  KeAcquireSpinLock(&pdx->QLock, &oldirql);
:
2
  KeReleaseSpinLock(&pdx->QLock, oldirql);
  }
```

1. When *KeAcquireSpinLock* acquires the spin lock, it also raises IRQL to *DISPATCH_LEVEL* and returns the current (that is, preacquisition) level in the variable to which the second argument points.

2. When *KeReleaseSpinLock* releases the spin lock, it also lowers IRQL back to the value specified in the second argument.

If you know you're already executing at *DISPATCH_LEVEL*, you can save a little time by calling two special routines. This technique is appropriate, for example, in DPC, *StartIo*, and other driver routines that execute at *DISPATCH_LEVEL*:

```
KeAcquireSpinLockAtDpcLevel(&pdx->QLock);
:
KeReleaseSpinLockFromDpcLevel(&pdx->QLock);
```

4.3.3 Queued Spin Locks

Windows XP introduces a new type of spin lock, called an in-stack queued spin lock, that has a more efficient implementation than a regular spin lock. The mechanics of using this new kind of lock are a bit different from what I just described. You still allocate a *KSPIN_LOCK* object in nonpaged memory to which all relevant parts of your driver have access, and you still initialize it by calling *KeAcquireSpinLock*. To acquire and release the lock, however, you use code like the following:

```
1
KLOCK QUEUE HANDLE qh;
2
KeAcquireInStackQueuedSpinLock(&pdx->QLock, &qh);
:
3
KeReleaseInStackQueuedSpinLock(&qh);
```

1. The *KLOCK_QUEUE_HANDLE* structure is opaque—you're not supposed to know what it contains, but you do have to reserve storage for it. The best way to do that is to define an automatic variable (hence the *in-stack* part of the name).

2. Call *KeAcquireInStackQueuedSpinLock* instead of *KeAcquireSpinLock* to acquire the lock, and supply the address of the *KLOCK_QUEUE_HANDLE* object as the second argument.

3. Call *KeReleaseInStackQueuedSpinLock* instead of *KeReleaseSpinLock* to release the lock.

The reason an in-stack queued spin lock is more efficient relates to the performance impact of a standard spin lock. With a standard spin lock, each CPU that is contending for ownership constantly modifies the same memory location. Each modification requires every contending CPU to reload the same dirty cache line. A queued spin lock, introduced for internal use in Windows 2000, avoids this adverse effect by cleverly using interlocked exchange and compare-exchange operations to track users and waiters for a lock. A waiting CPU continually reads (but does not write) a unique memory location. A CPU that releases a lock alters the memory variable on which the next waiter is spinning.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 108

Internal queued spin locks can't be directly used by driver code because they rely on a fixed-size table of lock pointers to which drivers don't have access. Windows XP added the *in-stack* queued spin lock, which relies on an automatic variable instead of the fixed-size table.

In addition to the two routines I showed you for acquiring and releasing this new kind of spin lock, you can also use two other routines if you know you're already executing at *DISPATCH_LEVEL*: *KeAcquireInStackQueuedSpinLockAtDpcLevel* and *KeReleaseInStackQueuedSpinLockFromDpcLevel*. (Try spelling those names three times fast!)

NOTE
Because Windows versions earlier than XP don't support the in-stack queued spin lock or interrupt spin lock routines, you can't directly call them in a driver intended to be binary portable between versions. The SPINLOCK sample driver shows how to make a run-time decision to use the newer spin locks under XP and the old spin locks otherwise.

4.4 Kernel Dispatcher Objects

The kernel provides five types of synchronization objects that you can use to control the flow of nonarbitrary threads. See Table 4-1 for a summary of these kernel *dispatcher object* types and their uses. At any moment, one of these objects is in one of two states: *signaled* or *not-signaled*. At times when it's permissible for you to block a thread in whose context you're running, you can wait for one or more objects to reach the signaled state by calling KeWaitForSingleObject or KeWaitForMultipleObjects. The kernel also provides routines for initializing and controlling the state of each of these objects.

Object	Data Type	Description
Event	*KEVENT*	Blocks a thread until some other thread detects that an event has occurred
Semaphore	*KSEMAPHORE*	Used instead of an event when an arbitrary number of wait calls can be satisfied
Mutex	*KMUTEX*	Excludes other threads from executing a particular section of code
Timer	*KTIMER*	Delays execution of a thread for a given period of time
Thread	*KTHREAD*	Blocks one thread until another thread terminates

Table 4-1. *Kernel Dispatcher Objects*

In the next few sections, I'll describe how to use the kernel dispatcher objects. I'll start by explaining when you can block a thread by calling one of the wait primitives, and then I'll discuss the support routines that you use with each of the object types. I'll finish this section by discussing the related concepts of thread alerts and asynchronous procedure call delivery.

4.4.1 How and When You Can Block

To understand when and how it's permissible for a WDM driver to block a thread on a kernel dispatcher object, you have to recall some of the basic facts about threads from Chapter 2. In general, whatever thread was executing at the time of a software or hardware interrupt continues to be the current thread while the kernel processes the interrupt. We speak of executing kernel-mode code *in the context* of this current thread. In response to interrupts of various kinds, the scheduler might decide to switch threads, of course, in which case a new thread becomes "current."

We use the terms *arbitrary thread context* and *nonarbitrary thread context* to describe the precision with which we can know the thread in whose context we're currently operating in a driver subroutine. If we know that we're in the context of the thread that initiated an I/O request, the context is not arbitrary. Much of the time, however, a WDM driver can't know this fact because chance usually controls which thread is active when the interrupt occurs that results in the driver being called. When applications issue I/O requests, they cause a transition from user mode to kernel mode. The I/O Manager routines that create an IRP and send it to a driver dispatch routine continue to operate in this nonarbitrary thread context, as does the first dispatch routine to see the IRP. We use the term *highest-level driver* to describe the driver whose dispatch routine first receives the IRP.

As a general rule, only a highest-level driver can know for sure that it's operating in a nonarbitrary thread context. Let's suppose you are a dispatch routine in a lower-level driver, and you're wondering whether you're getting called in an arbitrary thread. If the highest-level driver just sent you an IRP directly from its dispatch routine, you'd be in the original, nonarbitrary, thread. But suppose that driver had put an IRP on a queue and then returned to the application. That driver would have removed the IRP from the queue in an arbitrary thread and then sent it or another IRP to you. Unless you know that didn't happen, you should assume you're in an arbitrary thread if you're not the highest-level driver.

Notwithstanding what I just said, in many situations you can be sure of the thread context. Your *DriverEntry* and *AddDevice* routines are called in a system thread that you can block if you need to. You won't often need to explicitly block inside these routines, but you *could* if you wanted to. You receive *IRP_MJ_PNP* requests in a system thread too. In many cases, you *must* block that thread to correctly process the request. Finally, you'll sometimes receive I/O requests directly from an application, in which case you'll know you're in a thread belonging to the application.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 109

NOTE
Microsoft uses the term *highest-level driver* primarily to distinguish between file system drivers and the storage device drivers they call to do actual I/O. The file system driver is "highest level," while the storage driver is not. It would be easy to confuse this concept with the layering of WDM drivers, but it's not the same. The way I think of things is that all the WDM drivers for a given piece of hardware, including all the filter drivers, the function driver, and the bus driver, are collectively either "highest level" or not. A filter driver has no business queuing an IRP that, but for the intervention of the filter, would have flowed down the stack in the original thread context. So if the thread context was nonarbitrary when the IRP got to the topmost filter dispatch object (FiDO), it should still be nonarbitrary in every lower dispatch routine.

Also recall from the discussion earlier in this chapter that you must not block a thread if you're executing at or above *DISPATCH_LEVEL*.

Having recalled these facts about thread context and IRQL, we can state a simple rule about when it's OK to block a thread:

Block only the thread that originated the request you're working on, and only when executing at IRQL strictly less than DISPATCH_LEVEL.

Several of the dispatcher objects, and the so-called Executive Fast Mutex I'll discuss later in this chapter, offer "mutual exclusion" functionality. That is, they permit one thread to access a given shared resource without interference from other threads. This is pretty much what a spin lock does, so you might wonder how to choose between synchronization methods. In general, I think you should prefer to synchronize below *DISPATCH_LEVEL* if you can because that strategy allows a thread that owns a mutual exclusion lock to cause page faults and to be preempted by other threads if the thread continues to hold the lock for a long time. In addition, this strategy allows other CPUs to continue doing useful work, even though threads have blocked on those CPUs to acquire the same lock. If any of the code that accesses a shared resource can run at *DISPATCH_LEVEL*, though, you must use a spin lock because the *DISPATCH_LEVEL* code might interrupt code running at lower IRQL.

4.4.2 Waiting on a Single Dispatcher Object

You call *KeWaitForSingleObject* as illustrated in the following example:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER timeout;
NTSTATUS status = KeWaitForSingleObject(object, WaitReason,
  WaitMode, Alertable, &timeout);
```

As suggested by the *ASSERT*, you must be executing at or below *DISPATCH_LEVEL* to even call this service routine.

In this call, object points to the object you want to wait on. Although this argument is typed as a PVOID, it should be a pointer to one of the dispatcher objects listed in Table 4-1. The object must be in nonpaged memory—for example, in a device extension structure or other data area allocated from the nonpaged pool. For most purposes, the execution stack can be considered nonpaged.

WaitReason is a purely advisory value chosen from the *KWAIT_REASON* enumeration. No code in the kernel actually cares what value you supply here, so long as you don't specify *WrQueue*. (Internally, scheduler code bases some decisions on whether a thread is currently blocked for this "reason.") The reason a thread is blocked is saved in an opaque data structure, though. If you knew more about that data structure and were trying to debug a deadlock of some kind, you could perhaps gain clues from the reason code. The bottom line: always specify *Executive* for this parameter; there's no reason to say anything else.

WaitMode is one of the two values of the MODE enumeration: *KernelMode* or *UserMode*. *Alertable* is a simple Boolean value. Unlike *WaitReason*, these parameters do make a difference in the way the system behaves by controlling whether the wait can be terminated early in order to deliver asynchronous procedure calls of various kinds. I'll explain these interactions in more detail in "Thread Alerts and APCs" later in this chapter. Waiting in user mode also authorizes the Memory Manager to swap your thread's kernel-mode stack out. You'll see examples in this book and elsewhere where drivers create event objects, for instance, as automatic variables. A bug check would result if some other thread were to call *KeSetEvent* at elevated IRQL at a time when the event object was absent from memory. The bottom line: you should probably always wait in *KernelMode* and specify *FALSE* for the *Alertable* parameter.

The last parameter to *KeWaitForSingleObject* is the address of a 64-bit timeout value, expressed in 100-nanosecond units. A positive number for the timeout is an absolute timestamp relative to the January 1, 1601, epoch of the system clock. You can determine the current time by calling *KeQuerySystemTime*, and you can add a constant to that value. A negative number is an interval relative to the current time. If you specify an absolute time, a subsequent change to the system clock alters the duration of the timeout you might experience. That is, the timeout doesn't expire until the system clock equals or exceeds whatever absolute value you specify. In contrast, if you specify a relative timeout, the duration of the timeout you experience is unaffected by changes in the system clock.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 110

Why January 1, 1601?
Years ago, when I was first learning the Win32 API, I was bemused by the choice of January 1, 1601, as the origin for the timestamps in Windows NT. I understood the reason for this choice when I had occasion to write a set of conversion routines. Everyone knows that years divisible by four are leap years. Many people know that century years (such as 1900) are exceptions—they're not leap years even though they're divisible by 4. A few people know that every fourth century year (such as 1600 and 2000) is an exception to the exception—they are leap years. January 1, 1601, was the start of a 400-year cycle that ends in a leap year. If you base timestamps on this origin, it's possible to write programs that convert a Windows NT timestamp to a conventional representation of the date (and vice versa) without doing any jumps.

Specifying a zero timeout causes *KeWaitForSingleObject* to return immediately with a status code indicating whether the object is in the signaled state. *If you're executing at DISPATCH_LEVEL, you must specify a zero timeout because blocking is not allowed.* Each kernel dispatcher object offers a *KeReadStateXxx* service function that allows you to determine the state of the object. Reading the state isn't completely equivalent to waiting for zero time, however: when *KeWaitForSingleObject* discovers that the wait is satisfied, it performs the side effects that the particular object requires. In contrast, reading the state of the object doesn't perform the operations, even if the object is already signaled and a wait would be satisfied if it were requested right now.

Specifying a *NULL* pointer for the timeout parameter is *OK* and indicates an infinite wait.

The return value indicates one of several possible results. *STATUS_SUCCESS* is the result you expect and indicates that the wait was satisfied. That is, either the object was in the signaled state when you made the call to *KeWaitForSingleObject* or else the object was in the not-signaled state and later became signaled. When the wait is satisfied in this way, operations might need to be performed on the object. The nature of these operations depends on the type of the object, and I'll explain them later in this chapter in connection with discussing each type of object. (For example, a synchronization type of event will be reset after your wait is satisfied.)

The return value *STATUS_TIMEOUT* indicates that the specified timeout occurred without the object reaching the signaled state. If you specify a zero timeout, *KeWaitForSingleObject* returns immediately with either this code (indicating that the object is not-signaled) or *STATUS_SUCCESS* (indicating that the object is signaled). This return value isn't possible if you specify a NULL timeout parameter pointer because you thereby request an infinite wait.

Two other return values are possible. *STATUS_ALERTED* and *STATUS_USER_APC* mean that the wait has terminated without the object having been signaled because the thread has received an alert or a user-mode APC, respectively. I'll discuss these concepts a bit further on in "Thread Alerts and APCs."

Note that *STATUS_TIMEOUT*, *STATUS_ALERTED*, and *STATUS_USER_APC* all pass the *NT_SUCCESS* test. Therefore, don't simply use *NT_SUCCESS* on the return code from *KeWaitForSingleObject* in the expectation that it will distinguish between cases in which the object was signaled and cases in which the object was not signaled.

Windows 98/Me Compatibility Note
KeWaitForSingleObject and *KeWaitForMultipleObjects* have a horrible bug in Windows 98 and Millennium in that they can return the undocumented and nonsensical value *0xFFFFFFFF* in two situations. One situation occurs when a thread terminates while blocked on a WDM object. The wait returns early with this bogus code. The return code should never happen (because it's undocumented), and the wait shouldn't terminate early unless you specify *TRUE* for the *Alertable parameter*. You can work around this problem by just reissuing the wait.

The other circumstance in which you can get the bogus return occurs if the thread you're trying to block is *already* blocked. How, you might well ask, could you be executing in the context of a thread that's really blocked? This situation happens in Windows 98/Me when someone blocks on a VxD-level object with the *BLOCK_SVC_INTS* flag and the system later calls a function in your driver at what's called event time. You can nominally be in the context of the blocked thread, and you simply cannot block a second time on a WDM object. In fact, I've even seen *KeWaitForSingleObject* return with the IRQL raised to *DISPATCH_LEVEL* in this circumstance. As far as I know, there's no workaround for the problem. Thankfully, it seems to occur only with drivers for serial devices, in which there's a crossover between VxD and WDM code.

4.4.3 Waiting on Multiple Dispatcher Objects

KeWaitForMultipleObjects is a companion function to *KeWaitForSingleObject* that you use when you want to wait for one or all of several dispatcher objects simultaneously. Call this function as in this example:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER timeout;
NTSTATUS status = KeWaitForMultipleObjects(count, objects,
  WaitType, WaitReason, WaitMode, Alertable, &timeout, waitblocks);
```

Here *objects* is the address of an array of pointers to dispatcher objects, and *count* is the number of pointers in the array. The count must be less than or equal to the value *MAXIMUM_WAIT_OBJECTS*, which currently equals 64. The array, as well as each of the objects to which the elements of the array point, must be in nonpaged memory. *WaitType* is one of the enumeration

IPR2024-00292
KASPERSKY EX 1008 Page 111

values *WaitAll* or *WaitAny* and specifies whether you want to wait until all of the objects are simultaneously in the signaled state or whether, instead, you want to wait until any one of the objects is signaled.

The *waitblocks argument* points to an array of *KWAIT_BLOCK* structures that the kernel will use to administer the wait operation. You don't need to initialize these structures in any way—the kernel just needs to know where the storage is for the group of wait blocks that it will use to record the status of each of the objects during the wait. If you're waiting for a small number of objects (specifically, a number no bigger than *THREAD_WAIT_OBJECTS*, which currently equals 3), you can supply NULL for this parameter. If you supply *NULL*, *KeWaitForMultipleObjects* uses a *preallocated* array of wait blocks that lives in the thread object. If you're waiting for more objects than this, you must provide nonpaged memory that's at least *count* * *sizeof(KWAIT_BLOCK)* bytes in length.

The remaining arguments to *KeWaitForMultipleObjects* are the same as the corresponding arguments to *KeWaitForSingleObject*, and most return codes have the same meaning.

If you specify *WaitAll*, the return value *STATUS_SUCCESS* indicates that all the objects managed to reach the signaled state simultaneously. If you specify *WaitAny*, the return value is numerically equal to the objects array index of the single object that satisfied the wait. If more than one of the objects happens to be signaled, you'll be told about one of them—maybe the lowest-numbered of all the ones that are signaled at that moment, but maybe some other one. You can think of this value being *STATUS_WAIT_0* plus the array index. You can't simply perform the usual *NT_SUCCESS* test of the returned status before extracting the array index from the status code, though, because other possible return codes (including *STATUS_TIMEOUT*, *STATUS_ALERTED*, and *STATUS_USER_APC*) would also pass the test. Use code like this:

```
NTSTATUS status = KeWaitForMultipleObjects(...);
if ((ULONG) status < count)
  {
  ULONG iSignaled = (ULONG) status - (ULONG) STATUS WAIT 0;
  :
  }
```

When *KeWaitForMultipleObjects* returns a status code equal to an object's array index in a *WaitAny* case, it also performs the operations required by that object. If more than one object is signaled and you specified *WaitAny*, the operations are performed only for the one that's deemed to satisfy the wait and whose index is returned. That object isn't necessarily the first one in your array that happens to be signaled.

4.4.4 Kernel Events

You use the service functions listed in Table 4-2 to work with kernel event objects. To initialize an event object, first reserve nonpaged storage for an object of type KEVENT and then call *KeInitializeEvent*:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeEvent(event, EventType, initialstate);
```

Event is the address of the event object. *EventType* is one of the enumeration values *NotificationEvent* and *SynchronizationEvent*. A notification event has the characteristic that, when it is set to the signaled state, it stays signaled until it's explicitly reset to the not-signaled state. Furthermore, all threads that wait on a notification event are released when the event is signaled. This is like a manual-reset event in user mode. A synchronization event, on the other hand, gets reset to the not-signaled state as soon as a single thread gets released. This is what happens in user mode when someone calls *SetEvent* on an auto-reset event object. The only operation performed on an event object by *KeWaitXxx* is to reset a synchronization event to not-signaled. Finally, *initialstate* is *TRUE* to specify that the initial state of the event is to be signaled and *FALSE* to specify that the initial state is to be not-signaled.

Service Function	Description
KeClearEvent	Sets event to not-signaled; doesn't report previous state
KeInitializeEvent	Initializes event object
KeReadStateEvent	Determines current state of event (Windows XP and Windows 2000 only)
KeResetEvent	Sets event to not-signaled; returns previous state
KeSetEvent	Sets event to signaled; returns previous state

Table 4-2. *Service Functions for Use with Kernel Event Objects*

NOTE
In this series of sections on synchronization primitives, I'm repeating the IRQL restrictions that the DDK documentation describes. In the current release of Microsoft Windows XP, the DDK is sometimes more restrictive than the operating system actually is. For example, *KeClearEvent* can be called at any IRQL, not just at or below *DISPATCH_LEVEL*. *KeInitializeEvent* can be called at any IRQL, not just at *PASSIVE_LEVEL*. However, you should regard the statements in the DDK as being tantamount to saying that Microsoft might someday impose the documented restriction, which is why I haven't tried to report the true state of affairs.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 112

You can call *KeSetEvent* to place an event in the signaled state:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG wassignaled = KeSetEvent(event, boost, wait);
```

As implied by the *ASSERT*, you must be running at or below *DISPATCH_LEVEL* to call this function. The *event* argument is a pointer to the event object in question, and *boost* is a value to be added to a waiting thread's priority if setting the event results in satisfying someone's wait. See the sidebar ("That Pesky Third Argument to *KeSetEvent*") for an explanation of the Boolean *wait* argument, which a WDM driver would almost never want to specify as *TRUE*. The return value is nonzero if the event was already in the signaled state before the call and 0 if the event was in the not-signaled state.

A multitasking scheduler needs to artificially boost the priority of a thread that waits for I/O operations or synchronization objects in order to avoid starving threads that spend lots of time waiting. This is because a thread that blocks for some reason generally relinquishes its time slice and won't regain the CPU until either it has a relatively higher priority than other eligible threads or other threads that have the same priority finish their time slices. A thread that never blocks, however, gets to complete its time slices. Unless a boost is applied to the thread that repeatedly blocks, therefore, it will spend a lot of time waiting for CPU-bound threads to finish their time slices.

You and I won't always have a good idea of what value to use for a priority boost. A good rule of thumb to follow is to specify *IO_NO_INCREMENT* unless you have a good reason not to. If setting the event is going to wake up a thread that's dealing with a time-sensitive data flow (such as a sound driver), supply the boost that's appropriate to that kind of device (such as *IO_SOUND_INCREMENT*). The important thing is not to boost the waiter for a silly reason. For example, if you're trying to handle an *IRP_MJ_PNP* request synchronously—see Chapter 6—you'll be waiting for lower-level drivers to handle the IRP before you proceed, and your completion routine will be calling *KeSetEvent*. Since Plug and Play requests have no special claim on the processor and occur only infrequently, specify *IO_NO_INCREMENT*, even for a sound card.

That Pesky Third Argument to *KeSetEvent*

The purpose of the wait argument to *KeSetEvent* is to allow internal code to hand off control from one thread to another very quickly. System components other than device drivers can, for example, create paired event objects that are used by client and server threads to gate their communication. When the server wants to wake up its paired client, it will call *KeSetEvent* with the wait argument set to *TRUE* and then *immediately* call *KeWaitXxx* to put itself to sleep. The use of wait allows these two operations to be done atomically so that no other thread can be awakened in between and possibly wrest control from the client and the server.

The DDK has always sort of described what happens internally, but I've found the explanation confusing. I'll try to explain it in a different way so that you can see why you should always say *FALSE* for this parameter. Internally, the kernel uses a *dispatcher database lock* to guard operations related to thread blocking, waking, and scheduling. KeSetEvent needs to acquire this lock, and so do the *KeWaitXxx* routines. If you say TRUE for the wait argument, *KeSetEvent* sets a flag so that *KeWaitXxx* will know you did so, and it returns to you without releasing this lock. When you turn around and (immediately, please—you're running at a higher IRQL than every hardware device, and you own a spin lock that's *very* frequently in contention) call *KeWaitXxx*, it needn't acquire the lock all over again. The net effect is that you'll wake up the waiting thread and put yourself to sleep without giving any other thread a chance to start running.

You can see, first of all, that a function that calls *KeSetEvent* with wait set to *TRUE* has to be in nonpaged memory because it will execute briefly above *DISPATCH_LEVEL*. But it's hard to imagine why an ordinary device driver would even need to use this mechanism because it would almost never know better than the kernel which thread ought to be scheduled next. The bottom line: always say *FALSE* for this parameter. In fact, it's not clear why the parameter has even been exposed to tempt us.

You can determine the current state of an event (at any IRQL) by calling *KeReadStateEvent*:

```
LONG signaled = KeReadStateEvent(event);
```

The return value is nonzero if the event is signaled, 0 if it's not-signaled.

NOTE

***KeReadStateEvent* isn't supported in Microsoft Windows 98/Me, even though the other *KeReadStateXxx* functions described here are. The absence of support has to do with how events and other synchronization primitives are implemented in Windows 98/Me.**

You can determine the current state of an event and, immediately thereafter, place it in the not-signaled state by calling the *KeResetEvent* function (at or below *DISPATCH_LEVEL*):

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signaled = KeResetEvent(event);
```

If you're not interested in the previous state of the event, you can save a little time by calling *KeClearEvent* instead:

IPR2024-00292
KASPERSKY EX 1008 Page 113

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeClearEvent(event);
```

KeClearEvent is faster because it doesn't need to capture the current state of the event before setting it to not-signaled. But beware of calling *KeClearEvent* when another thread might be using the same event since there's no good way to control the races between you clearing the event and some other thread setting it or waiting on it.

Using a Synchronization Event for Mutual Exclusion

I'll tell you later in this chapter about two types of mutual exclusion objects—a kernel mutex and an executive fast mutex—that you can use to limit access to shared data in situations in which a spin lock is inappropriate for some reason. Sometimes you can simply use a synchronization event for this purpose. First define the event in nonpaged memory, as follows:

```
typedef struct  DEVICE_EXTENSION {
  :
  KEVENT lock;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Initialize it as a synchronization event in the signaled state:

```
KeInitializeEvent(&pdx->lock, SynchronizationEvent, TRUE);
```

Enter your lightweight critical section by waiting on the event. Leave by setting the event.

```
KeWaitForSingleObject(&pdx->lock, Executive, KernelMode, FALSE, NULL);
  :
KeSetEvent(&pdx->lock, EVENT_INCREMENT, FALSE);
```



Use this trick only in a system thread, though, to prevent a user-mode call to *NtSuspendThread* from creating a deadlock. (This deadlock can easily happen if a user-mode debugger is running on the same process.) If you're running in a user thread, you should prefer to use an executive fast mutex. Don't use this trick at all for code that executes in the paging path, as explained later in connection with the "unsafe" way of acquiring an executive fast mutex.

4.4.5 Kernel Semaphores

A kernel semaphore is an integer counter with associated synchronization semantics. The semaphore is considered signaled when the counter is positive and not-signaled when the counter is 0. The counter cannot take on a negative value. Releasing a semaphore increases the counter, whereas successfully waiting on a semaphore decrements the counter. If the decrement makes the count 0, the semaphore is then considered not-signaled, with the consequence that other KeWaitXxx callers who insist on finding it signaled will block. Note that if more threads are waiting for a semaphore than the value of the counter, not all of the waiting threads will be unblocked.

The kernel provides three service functions to control the state of a semaphore object. (See Table 4-3.) You initialize a semaphore by making the following function call at *PASSIVE_LEVEL*:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeSemaphore(semaphore, count, limit);
```

In this call, *semaphore* points to a *KSEMAPHORE* object in nonpaged memory. The *count* variable is the initial value of the counter, and *limit* is the maximum value that the counter will be allowed to take on, which must be as large as the initial count.

Service Function	*Description*
KeInitializeSemaphore	Initializes semaphore object
KeReadStateSemaphore	Determines current state of semaphore
KeReleaseSemaphore	Sets semaphore object to the signaled state

Table 4-3. *Service Functions for Use with Kernel Semaphore Objects*

If you create a semaphore with a limit of 1, the object is somewhat similar to a mutex in that only one thread at a time will be able to claim it. A kernel mutex has some features that a semaphore lacks, however, to help prevent deadlocks. Accordingly, there's almost no point in creating a semaphore with a limit of 1.

If you create a semaphore with a limit bigger than 1, you have an object that allows multiple threads to access a given resource. A familiar theorem in queuing theory dictates that providing a single queue for multiple servers is more fair (that is, results in less variation in waiting times) than providing a separate queue for each of several servers. The average waiting time is the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 114

same in both cases, but the variation in waiting times is smaller with the single queue. (This is why queues in stores are increasingly organized so that customers wait in a single line for the next available clerk.) This kind of semaphore allows you to organize a set of software or hardware servers to take advantage of that theorem.

The owner (or one of the owners) of a semaphore releases its claim to the semaphore by calling *KeReleaseSemaphore*:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG wassignaled = KeReleaseSemaphore(semaphore, boost,  delta, wait);
```

This operation adds *delta*, which must be positive, to the counter associated with semaphore, thereby putting the semaphore in the signaled state and allowing other threads to be released. In most cases, you'll specify 1 for this parameter to indicate that one claimant of the semaphore is releasing its claim. The boost and wait parameters have the same import as the corresponding parameters to *KeSetEvent*, discussed earlier. The return value is 0 if the previous state of the semaphore was not-signaled and nonzero if the previous state was signaled.

KeReleaseSemaphore doesn't allow you to increase the counter beyond the limit specified when you initialized the semaphore. If you try, it doesn't adjust the counter at all, and it raises an exception with the code *STATUS_SEMAPHORE_LIMIT_EXCEEDED*. Unless someone has a structured exception handler to trap the exception, a bug check will eventuate.

You can also interrogate the current state of a semaphore with this call:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signaled = KeReadStateSemaphore(semaphore);
```

The return value is nonzero if the semaphore is signaled and 0 if the semaphore is not-signaled. You shouldn't assume that the return value is the current value of the counter—it could be any nonzero value if the counter is positive.

Having told you all this about how to use kernel semaphores, I feel I ought to tell you that I've never seen a driver that uses one of them.

4.4.6 Kernel Mutexes

The word *mutex* is a contraction of *mutual exclusion*. A kernel mutex object provides one method (and not necessarily the best one) to serialize access by competing threads to a given shared resource. The mutex is considered signaled if no thread owns it and not-signaled if a thread currently does own it. When a thread gains control of a mutex after calling one of the *KeWaitXxx* routines, the kernel also prevents delivery of any but special kernel APCs to help avoid possible deadlocks. This is the operation referred to in the earlier discussion of *KeWaitForSingleObject* (in the section "Waiting on a Single Dispatcher Object").

It's generally better to use an executive fast mutex rather than a kernel mutex, as I'll explain in more detail later in "Fast Mutex Objects." The main difference between the two is that acquiring a fast mutex raises the IRQL to *APC_LEVEL*, whereas acquiring a kernel mutex doesn't change the IRQL. Among the reasons you care about this fact is that completion of so-called synchronous IRPs requires delivery of a special kernel-mode APC, which cannot occur if the IRQL is higher than *PASSIVE_LEVEL*. Thus, you can create and use synchronous IRPs while owning a kernel mutex but not while owning an executive fast mutex. Another reason for caring arises for drivers that execute in the paging path, as elaborated later on in connection with the "unsafe" way of acquiring an executive fast mutex.

Another, less important, difference between the two kinds of mutex object is that a kernel mutex can be acquired recursively, whereas an executive fast mutex cannot. That is, the owner of a kernel mutex can make a subsequent call to *KeWaitXxx* specifying the same mutex and have the wait immediately satisfied. A thread that does this must release the mutex an equal number of times before the mutex will be considered free.

Table 4-4 lists the service functions you use with mutex objects.

Service Function	Description
KeInitializeMutex	Initializes mutex object
KeReadStateMutex	Determines current state of mutex
KeReleaseMutex	Sets mutex object to the signaled state

Table 4-4. *Service Functions for Use with Kernel Mutex Objects*

To create a mutex, you reserve nonpaged memory for a *KMUTEX* object and make the following initialization call:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeMutex(mutex, level);
```

where *mutex* is the address of the *KMUTEX* object, and *level* is a parameter originally intended to help avoid deadlocks when your own code uses more than one mutex. Since the kernel currently ignores the *level* parameter, I'm not going to attempt to

IPR2024-00292
KASPERSKY EX 1008 Page 115

describe what it used to mean.

The mutex begins life in the signaled—that is, unowned—state. An immediate call to *KeWaitXxx* would take control of the mutex and put it in the not-signaled state.

You can interrogate the current state of a mutex with this function call:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG signaled = KeReadStateMutex(mutex);
```

The return value is 0 if the mutex is currently owned, nonzero if it's currently unowned.

The thread that owns a mutex can release ownership and return the mutex to the signaled state with this function call:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LONG wassignaled = KeReleaseMutex(mutex, wait);
```

The *wait* parameter means the same thing as the corresponding argument to *KeSetEvent*. The return value is always 0 to indicate that the mutex was previously owned because, if this were not the case, *KeReleaseMutex* would have bugchecked (it being an error for anyone but the owner to release a mutex).

Just for the sake of completeness, I want to mention a macro in the DDK named *KeWaitForMutexObject*. (See WDM.H.) It's defined simply as follows:

```
#define KeWaitForMutexObject KeWaitForSingleObject
```

Using this special name offers no benefit at all. You don't even get the benefit of having the compiler insist that the first argument be a pointer to a KMUTEX instead of any random pointer type.

4.4.7 Kernel Timers

The kernel provides a timer object that functions something like an event that automatically signals itself at a specified absolute time or after a specified interval. It's also possible to create a timer that signals itself repeatedly and to arrange for a DPC callback following the expiration of the timer. Table 4-5 lists the service functions you use with timer objects.

Service Function	Description
KeCancelTimer	Cancels an active timer
KeInitializeTimer	Initializes a one-time notification timer
KeInitializeTimerEx	Initializes a one-time or repetitive notification or synchronization timer
KeReadStateTimer	Determines current state of a timer
KeSetTimer	(Re)specifies expiration time for a notification timer
KeSetTimerEx	(Re)specifies expiration time and other properties of a timer

Table 4-5. *Service Functions for Use with Kernel Timer Objects*

There are several usage scenarios for timers, which I'll describe in the next few sections:

- Timer used like a self-signaling event
- Timer with a DPC routine to be called when a timer expires
- Periodic timer used to call a DPC routine over and over again

Notification Timers Used like Events

In this scenario, we'll create a notification timer object and wait until it expires. First allocate a *KTIMER* object in nonpaged memory. Then, running at or below *DISPATCH_LEVEL*, initialize the timer object, as shown here:

```
PKTIMER timer;      // <== someone gives you this
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeInitializeTimer(timer);
```

At this point, the timer is in the not-signaled state and isn't counting down—a wait on the timer would never be satisfied. To start the timer counting, call *KeSetTimer* as follows:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
```


IPR2024-00292
KASPERSKY EX 1008 Page 116

```
BOOLEAN wascounting = KeSetTimer(timer, duetime, NULL);
```

The *duetime* value is a 64-bit time value expressed in 100-nanosecond units. If the value is positive, it's an absolute time relative to the same January 1, 1601, epoch used for the system timer. If the value is negative, it's an interval relative to the current time. If you specify an absolute time, a subsequent change to the system clock alters the duration of the timeout you experience. That is, the timer doesn't expire until the system clock equals or exceeds whatever absolute value you specify. In contrast, if you specify a relative timeout, the duration of the timeout you experience is unaffected by changes in the system clock. These are the same rules that apply to the timeout parameter to *KeWaitXxx*.

The return value from *KeSetTimer*, if *TRUE*, indicates that the timer was already counting down (in which case, our call to *KeSetTimer* would have canceled it and started the count all over again).

At any time, you can determine the current state of a timer:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
BOOLEAN counting = KeReadStateTimer(timer);
```

KeInitializeTimer and *KeSetTimer* are actually older service functions that have been superseded by newer functions. We could have initialized the timer with this call:

```
ASSERT(KeGetCurrentIqrl() <= DISPATCH_LEVEL);
KeInitializeTimerEx(timer, NotificationTimer);
```

We could also have used the extended version of the set timer function, *KeSetTimerEx*:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimerEx(timer, duetime, 0, NULL);
```

I'll explain a bit further on in this chapter the purpose of the extra parameters in these extended versions of the service functions.

Once the timer is counting down, it's still considered to be not-signaled until the specified due time arrives. At that point, the object becomes signaled, and all waiting threads are released. The system guarantees only that the expiration of the timer will be noticed no sooner than the due time you specify. If you specify a due time with a precision finer than the granularity of the system timer (which you can't control), the timeout will be noticed later than the exact instant you specify. You can call *KeQueryTimeIncrement* to determine the granularity of the system clock.

Notification Timers Used with a DPC

In this scenario, we want expiration of the timer to trigger a DPC. You would choose this method of operation if you wanted to be sure that you could service the timeout no matter what priority level your thread had. (Since you can wait only below *DISPATCH_LEVEL*, regaining control of the CPU after the timer expires is subject to the normal vagaries of thread scheduling. The DPC, however, executes at elevated IRQL and thereby effectively preempts all threads.)

We initialize the timer object in the same way. We also have to initialize a KDPC object for which we allocate nonpaged memory. For example:

```
PKDPC dpc;  // <== points to KDPC you've allocated
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KeInitializeTimer(timer);
KeInitializeDpc(dpc, DpcRoutine, context);
```

You can initialize the timer object by using either *KeInitializeTimer* or *KeInitializeTimerEx*, as you please. *DpcRoutine* is the address of a deferred procedure call routine, which must be in nonpaged memory. The *context* parameter is an arbitrary 32-bit value (typed as a PVOID) that will be passed as an argument to the DPC routine. The *dpc* argument is a pointer to a KDPC object for which you provide nonpaged storage. (It might be in your device extension, for example.)

When we want to start the timer counting down, we specify the DPC object as one of the arguments to *KeSetTimer* or *KeSetTimerEx*:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimer(timer, duetime, dpc);
```

You could also use the extended form *KeSetTimerEx* if you wanted to. The only difference between this call and the one we examined in the preceding section is that we've specified the DPC object address as an argument. When the timer expires, the system will queue the DPC for execution as soon as conditions permit. This would be at least as soon as you'd be able to wake

IPR2024-00292
KASPERSKY EX 1008 Page 117

up from a wait. Your DPC routine would have the following skeletal appearance:

```
VOID DpcRoutine(PKDPC dpc, PVOID context, PVOID junk1, PVOID junk2)
{
  :
}
```

For what it's worth, even when you supply a DPC argument to *KeSetTimer* or *KeSetTimerEx*, you can still call *KeWaitXxx* to wait at *PASSIVE_LEVEL* or *APC_LEVEL* if you want. On a single-CPU system, the DPC would occur before the wait could finish because it executes at a higher IRQL.

Synchronization Timers

Like event objects, timer objects come in both notification and synchronization flavors. A notification timer allows any number of waiting threads to proceed once it expires. A synchronization timer, by contrast, allows only a single thread to proceed. Once a thread's wait is satisfied, the timer switches to the not-signaled state. To create a synchronization timer, you must use the extended form of the initialization service function:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
KeInitializeTimerEx(timer, SynchronizationTimer);
```

SynchronizationTimer is one of the values of the *TIMER_TYPE* enumeration. The other value is *NotificationTimer*.

If you use a DPC with a synchronization timer, think of queuing the DPC as being an extra thing that happens when the timer expires. That is, expiration puts the timer in the signaled state and queues a DPC. One thread can be released as a result of the timer being signaled.

The only use I've ever found for a synchronization timer is when you want a periodic timer (see the next section).

Periodic Timers

So far, I've discussed only timers that expire exactly once. By using the extended set timer function, you can also request a periodic timeout:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
LARGE_INTEGER duetime;
BOOLEAN wascounting = KeSetTimerEx(timer, duetime,  period, dpc);
```

Here period is a periodic timeout, expressed in milliseconds (ms), and dpc is an optional pointer to a KDPC object. A timer of this kind expires once at the due time and periodically thereafter. To achieve exact periodic expiration, specify the same relative due time as the interval. Specifying a zero due time causes the timer to immediately expire, whereupon the periodic behavior takes over. It often makes sense to start a periodic timer in conjunction with a DPC object, by the way, because doing so allows you to be notified without having to repeatedly wait for the timeout.



Be sure to call *KeCancelTimer* to cancel a periodic timer before the *KTIMER* object or the DPC routine disappears from memory. It's quite embarrassing to let the system unload your driver and, 10 nanoseconds later, call your nonexistent DPC routine. Not only that, but it causes a bug check. These problems are so hard to debug that the Driver Verifier makes a special check for releasing memory that contains an active *KTIMER*.

An Example

One use for kernel timers is to conduct a polling loop in a system thread dedicated to the task of repeatedly checking a device for activity. Not many devices nowadays need to be served by a polling loop, but yours may be one of the few exceptions. I'll discuss this subject in Chapter 14, and the companion content includes a sample driver (POLLING) that illustrates all of the concepts involved. Part of that sample is the following loop that polls the device at fixed intervals. The logic of the driver is such that the loop can be broken by setting a kill event. Consequently, the driver uses KeWaitForMultipleObjects. The code is actually a bit more complicated than the following fragment, which I've edited to concentrate on the part related to the timer:

```
VOID PollingThreadRoutine(PDEVICE_EXTENSION pdx)
  {
  NTSTATUS status;
  KTIMER timer;
1
  KeInitializeTimerEx(&timer, SynchronizationTimer);
2
  PVOID pollevents[] = {
    (PVOID) &pdx->evKill,
```


IPR2024-00292
KASPERSKY EX 1008 Page 118

```
    (PVOID) &timer,
    };
  C_ASSERT(arraysize(pollevents) <= THREAD_WAIT_OBJECTS);

  LARGE_INTEGER duetime = {0};
  #define POLLING_INTERVAL 500
3
  KeSetTimerEx(&timer, duetime, POLLING_INTERVAL, NULL);
  while (TRUE)
    {
4
    status = KeWaitForMultipleObjects(arraysize(pollevents),
      pollevents, WaitAny, Executive, KernelMode, FALSE,
      NULL, NULL);
    if (status == STATUS_WAIT_0)
      break;
5
    if (<device needs attention>)
      <do something>;
    }
  KeCancelTimer(&timer);
  PsTerminateSystemThread(STATUS_SUCCESS);
  }
```

1. Here we initialize a kernel timer. You must specify a *SynchronizationTimer* here, because a *NotificationTimer* stays in the signaled state after the first expiration.

2. We'll need to supply an array of dispatcher object pointers as one of the arguments to KeWaitForMultipleObjects, and this is where we set that up. The first element of the array is the kill event that some other part of the driver might set when it's time for this system thread to exit. The second element is the timer object. The C_ASSERT statement that follows this array verifies that we have few enough objects in our array that we can implicitly use the default array of wait blocks in our thread object.

3. The KeSetTimerEx statement starts a periodic timer running. The duetime is 0, so the timer goes immediately into the signaled state. It will expire every 500 ms thereafter.

4. Within our polling loop, we wait for the timer to expire or for the kill event to be set. If the wait terminates because of the kill event, we leave the loop, clean up, and exit this system thread. If the wait terminates because the timer has expired, we go on to the next step.

5. This is where our device driver would do something related to our hardware.

Alternatives to Kernel Timers

Rather than use a kernel timer object, you can use two other timing functions that might be more appropriate. First of all, you can call *KeDelayExecutionThread* to wait at *PASSIVE_LEVEL* for a given interval. This function is obviously less cumbersome than creating, initializing, setting, and awaiting a timer by using separate function calls.

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
LARGE_INTEGER duetime;
NSTATUS status = KeDelayExecutionThread(WaitMode,  Alertable, &duetime);
```

Here *WaitMode*, *Alertable*, and the returned status code have the same meaning as the corresponding parameters to *KeWaitXxx*, and *duetime* is the same kind of timestamp that I discussed previously in connection with kernel timers. Note that this function requires a *pointer* to a large integer for the timeout parameter, whereas other functions related to timers require the large integer itself.

If your requirement is to delay for a very brief period of time (less than 50 microseconds), you can call *KeStallExecutionProcessor* at any IRQL:

```
KeStallExecutionProcessor(nMicroSeconds);
```

The purpose of this delay is to allow your hardware time to prepare for its next operation before your program continues executing. The delay might end up being significantly longer than you request because *KeStallExecutionProcessor* can be preempted by activities that occur at a higher IRQL than that which the caller is using.

4.4.8 Using Threads for Synchronization

The Process Structure component of the operating system provides a few routines that WDM drivers can use for creating and

IPR2024-00292
KASPERSKY EX 1008 Page 119

controlling system threads. I'll be discussing these routines later on in Chapter 14 from the perspective of how you can use these functions to help you manage a device that requires periodic polling. For the sake of thoroughness, I want to mention here that you can use a pointer to a kernel thread object in a call to *KeWaitXxx* to wait for the thread to complete. The thread terminates itself by calling *PsTerminateSystemThread*.

Before you can wait for a thread to terminate, you need to first obtain a pointer to the opaque *KTHREAD* object that internally represents that thread, which poses a bit of a problem. While running in the context of a thread, you can determine your own *KTHREAD* easily:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
PKTHREAD thread = KeGetCurrentThread();
```

Unfortunately, when you call *PsCreateSystemThread* to create a new thread, you get back only an opaque HANDLE for the thread. To get the *KTHREAD* pointer, you use an Object Manager service function:

```
HANDLE hthread;
PKTHREAD thread;
PsCreateSystemThread(&hthread, ...);
ObReferenceObjectByHandle(hthread, THREAD_ALL_ACCESS,
  NULL, KernelMode, (PVOID*) &thread, NULL);
ZwClose(hthread);
```

ObReferenceObjectByHandle converts your handle to a pointer to the underlying kernel object. Once you have the pointer, you can discard the handle by calling *ZwClose*. At some point, you need to release your reference to the thread object by making a call to *ObDereferenceObject*:

```
ObDereferenceObject(thread);
```

4.4.9 Thread Alerts and APCs

Internally, the Windows NT kernel uses *thread alerts* as a way of waking threads. It uses an asynchronous procedure call as a way of waking a thread to execute some particular subroutine in that thread's context. The support routines that generate alerts or APCs aren't exposed for use by WDM driver writers. But since the DDK documentation and header files contain a great many references to these concepts, I want to finish this discussion of kernel dispatcher objects by explaining them.

I'll start by describing the "plumbing"—how these two mechanisms work. When someone blocks a thread by calling one of the *KeWaitXxx* routines, they specify by means of a Boolean argument whether the wait is to be *alertable*. An alertable wait might finish early—that is, without any of the wait conditions or the timeout being satisfied—because of a thread alert. Thread alerts originate in user mode when someone calls the native API function *NtAlertThread*. The kernel returns the special status value *STATUS_ALERTED* when a wait terminates early because of an alert.

An APC is a mechanism whereby the operating system can execute a function in the context of a particular thread. The *asynchronous* part of an APC stems from the fact that the system effectively interrupts the target thread to execute an out-of-line subroutine.

APCs come in three flavors: user mode, kernel mode, and special kernel mode. User-mode code requests a user-mode APC by calling the Win32 API *QueueUserAPC*. Kernel-mode code requests an APC by calling an undocumented function for which the DDK headers have no prototype. Diligent reverse engineers probably already know the name of this routine and something about how to call it, but it's really just for internal use and I'm not going to say any more about it. The system queues APCs to a specific thread until appropriate execution conditions exist. Appropriate execution conditions depend on the type of APC, as follows:

- Special kernel APCs execute as soon as possible—that is, as soon as an activity at *APC_LEVEL* can be scheduled in the thread. A special kernel APC can even temporarily awaken a blocked thread in many circumstances.
- Normal kernel APCs execute after all special APCs have been executed but only when the target thread is running and no other kernel-mode APC is executing in this thread. Delivery of normal kernel and user-mode APCs can be blocked by calling *KeEnterCriticalRegion*.
- User-mode APCs execute after both flavors of kernel-mode APC for the target thread have been executed but only if the thread has previously been in an alertable wait in user mode. Execution actually occurs the next time the thread is dispatched for execution in user mode.

If the system awakens a thread to deliver a user-mode APC, the wait primitive on which the thread was previously blocked returns with one of the special status values *STATUS_KERNEL_APC* and *STATUS_USER_APC*.

The Strange Role of APC_LEVEL

The IRQ level named *APC_LEVEL* works in a way that I found to be unexpected. You're allowed to block a thread running at

IPR2024-00292
KASPERSKY EX 1008 Page 120

APC_LEVEL (or at *PASSIVE_LEVEL*, but we're concerned only with *APC_LEVEL* right now). An *APC_LEVEL* thread can also be interrupted by any hardware device, following which a higher-priority thread might become eligible to run. In either situation, the thread scheduler can then give control of the CPU to another thread, which might be running at *PASSIVE_LEVEL* or *APC_LEVEL*. In effect, the IRQL levels *PASSIVE_LEVEL* and *APC_LEVEL* pertain to a thread, whereas the higher IRQLs pertain to a *CPU*.

How APCs Work with I/O Requests

The kernel uses the APC concept for several purposes. We're concerned in this book just with writing device drivers, though, so I'm only going to explain how APCs relate to the process of performing an I/O operation. In one of many possible scenarios, when a user-mode program performs a synchronous *ReadFile* operation on a handle, the Win32 subsystem calls a kernel-mode routine named *NtReadFile*. *NtReadFile* creates and submits an IRP to the appropriate device driver, which often returns *STATUS_PENDING* to indicate that it hasn't finished the operation. *NtReadFile* returns this status code to *ReadFile*, which thereupon calls *NtWaitForSingleObject* to wait on the file object to which the user-mode handle points. *NtWaitForSingleObject*, in turn, calls *KeWaitForSingleObject* to perform a nonalertable user-mode wait on an event object within the file object.

When the device driver eventually finishes the read operation, it calls *IoCompleteRequest*, which queues a special kernel-mode APC. The APC routine calls *KeSetEvent* to signal the file object, thereby releasing the application to continue execution. Some sort of APC is required because some of the tasks that need to be performed when an I/O request is completed (such as buffer copying) must occur in the address context of the requesting thread. A kernel-mode APC is required because the thread in question is not in an alertable wait state. A special APC is required because the thread is actually ineligible to run at the time we need to deliver the APC. In fact, the APC routine is the mechanism for awakening the thread.

Kernel-mode routines can call *ZwReadFile*, which turns into a call to *NtReadFile*. If you obey the injunctions in the DDK documentation when you call *ZwReadFile*, your call to *NtReadFile* will look almost like a user-mode call and will be processed in almost the same way, with just two differences. The first, which is quite minor, is that any waiting will be done in kernel mode. The other difference is that if you specified in your call to *ZwCreateFile* that you wanted to do synchronous operations, the I/O Manager will automatically wait for your read to finish. The wait will be alertable or not, depending on the exact option you specify to *ZwCreateFile*.

How to Specify Alertable and WaitMode Parameters

Now you have enough background to understand the ramifications of the *Alertable* and *WaitMode* parameters in the calls to the various wait primitives. As a general rule, you'll never be writing code that responds synchronously to requests from user mode. You *could* do so for, say, certain I/O control requests. Generally speaking, however, it's better to pend any operations that take a long time to finish (by returning *STATUS_PENDING* from your dispatch routine) and to finish them asynchronously. So, to continue speaking generally, you don't often call a wait primitive in the first place. Thread blocking is appropriate in a device driver in only a few scenarios, which I'll describe in the following sections.

Kernel Threads

Sometimes you'll create your own kernel-mode thread—when your device needs to be polled periodically, for example. In this scenario, any waits performed will be in kernel mode because the thread runs exclusively in kernel mode.

Handling Plug and Play Requests

I'll show you in Chapter 6 how to handle the I/O requests that the PnP Manager sends your way. Several such requests require *synchronous handling* on your part. In other words, you pass them down the driver stack to lower levels and wait for them to complete. You'll be calling *KeWaitForSingleObject* to wait in kernel mode because the PnP Manager calls you within the context of a kernel-mode thread. In addition, if you need to perform subsidiary requests as part of handling a PnP request—for example, to talk to a universal serial bus (USB) device—you'll be waiting in kernel mode.

Handling Other I/O Requests

When you're handling other sorts of I/O requests and you know that you're running in the context of a nonarbitrary thread that must get the results of your deliberations before proceeding, it might conceivably be appropriate to block that thread by calling a wait primitive. In such a case, you want to wait in the same processor mode as the entity that called you. Most of the time, you can simply rely on the *RequestorMode* in the IRP you're currently processing. If you gained control by means other than an IRP, you could call *ExGetPreviousMode* to determine the previous processor mode. If you're going to wait for a long time, it would be well to use the result of these tests as the *WaitMode* argument in your *KeWaitXxx* call, and it would also be well to specify *TRUE* for the *Alertable* argument.

NOTE
The bottom line: perform nonalertable waits unless you know you shouldn't.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 121

4.5 Other Kernel-Mode Synchronization Primitives

The Windows XP kernel offers some additional methods for synchronizing execution between threads or for guarding access to shared objects. In this section, I'll discuss the *fast mutex*, which is a mutual exclusion object that offers faster performance than a kernel mutex because it's optimized for the case in which no contention is actually occurring. I'll also describe the category of support functions that include the word *Interlocked* somewhere in their name. These functions carry out certain common operations—such as incrementing or decrementing an integer or inserting or removing an entry from a linked list—in an atomic way that prevents multitasking or multiprocessing interference.

4.5.1 Fast Mutex Objects

An *executive fast mutex* provides an alternative to a kernel mutex for protecting a critical section of code. Table 4-6 summarizes the service functions you use to work with this kind of object.

Service Function	Description
ExAcquireFastMutex	Acquires ownership of mutex, waiting if necessary
ExAcquireFastMutexUnsafe	Acquires ownership of mutex, waiting if necessary, in circumstance in which caller has already disabled receipt of APCs
ExInitializeFastMutex	Initializes mutex object
ExReleaseFastMutex	Releases mutex
ExReleaseFastMutexUnsafe	Releases mutex without reenabling APC delivery
ExTryToAcquireFastMutex	Acquires mutex if possible to do so without waiting

Table 4-6. *Service Functions for Use with Executive Fast Mutexes*

Compared with kernel mutexes, fast mutexes have the strengths and weaknesses summarized in Table 4-7. On the plus side, a fast mutex is much faster to acquire and release if there's no actual contention for it. On the minus side, a thread that acquires a fast mutex will not be able to receive certain types of asynchronous procedure call, depending on exactly which functions you call, and this constrains how you send IRPs to other drivers.

Kernel Mutex	Fast Mutex
Can be acquired recursively by a single thread (system maintains a claim counter)	Cannot be acquired recursively
Relatively slower	Relatively faster
Owner will receive only "special" kernel APCs	Owner won't receive any APCs unless you use the XxxUnsafe functions
Can be part of a multiple-object wait	Cannot be used as an argument to KeWaitForMultipleObjects

Table 4-7. *Comparison of Kernel and Fast Mutex Objects*

Incidentally, the DDK documentation about kernel mutex objects has long said that the kernel gives a priority boost to a thread that claims a mutex. I'm reliably informed that this hasn't actually been true since 1992 (the year, that is, not the Windows build number). The documentation has also long said that a thread holding a mutex can't be removed from the balance set (that is, subjected to having all of its pages moved out of physical memory). This was true when Windows NT was young but hasn't been true for a long time.

To create a fast mutex, you must first allocate a *FAST_MUTEX* data structure in nonpaged memory. Then you initialize the object by "calling" *ExInitializeFastMutex*, which is really a macro in WDM.H:

```
ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
ExInitializeFastMutex(FastMutex);
```

where *FastMutex* is the address of your *FAST_MUTEX* object. The mutex begins life in the unowned state. To acquire ownership later on, call one of these functions:

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExAcquireFastMutex(FastMutex);
```

or

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExAcquireFastMutexUnsafe(FastMutex);
```

The first of these functions waits for the mutex to become available, assigns ownership to the calling thread, and then raises the

IPR2024-00292
KASPERSKY EX 1008 Page 122

current processor IRQL to *APC_LEVEL*. Raising the IRQL has the effect of blocking delivery of all APCs. The second of these functions doesn't change the IRQL.



You need to think about potential deadlocks if you use the "unsafe" function to acquire a fast mutex. A situation to avoid is allowing user-mode code to suspend a thread in which you hold a mutex. That would deadlock other threads that need the mutex. For this reason, the DDK recommends (and the Driver Verifier requires) that you forestall the delivery of user-mode and normal kernel-mode APCs either by raising the IRQL to *APC_LEVEL* or by calling *KeEnterCriticalRegion* before *ExAcquireFastMutexUnsafe*. (Thread suspension involves an APC, so user-mode code can't suspend your thread if you disallow user-mode APCs. Yes, I know the reasoning here is a bit of a stretch!)

Another possible deadlock can arise with a driver in the paging path—in other words, a driver that gets called to help the memory manager process a page fault. Suppose you simply call *KeEnterCriticalRegion* and then *ExAcquireFastMutexUnsafe*. Now suppose the system tries to execute a special kernel-mode APC in the same thread, which is possible because *KeEnterCriticalRegion* doesn't forestall special kernel APCs. The APC routine might page fault, which might then lead to you being reentered and deadlocking on a second attempt to claim the same mutex. You avoid this situation by raising IRQL to *APC_LEVEL* before acquiring the mutex in the first place or, more simply, by using *KeAcquireFastMutex* instead of *KeAcquireFastMutexUnsafe*. The same problem can arise if you use a regular *KMUTEX* or a synchronization event, of course.

IMPORTANT
If you use *ExAcquireFastMutex*, you will be at *APC_LEVEL*. This means you can't create any synchronous IRPs. (The routines that do this must be called at *PASSIVE_LEVEL*.) Furthermore, you'll deadlock if you try to wait for a synchronous IRP to complete (because completion requires executing an APC, which can't happen because of the IRQL). In Chapter 5, I'll discuss how to use asynchronous IRPs to work around this problem.

If you don't want to wait if the mutex isn't immediately available, use the "try to acquire" function:

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
BOOLEAN acquired = ExTryToAcquireFastMutex(FastMutex);
```

If the return value is *TRUE*, you now own the mutex. If it's *FALSE*, someone else owns the mutex and has prevented you from acquiring it.

To release control of a fast mutex and allow some other thread to claim it, call the release function corresponding to the way you acquired the fast mutex:

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExReleaseFastMutex(FastMutex);
```

or

```
ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
ExReleaseFastMutexUnsafe(FastMutex);
```

A fast mutex is fast because the acquisition and release steps are optimized for the usual case in which there's no contention for the mutex. The critical step in acquiring the mutex is to atomically decrement and test an integer counter that indicates how many threads either own or are waiting for the mutex. If the test indicates that no other thread owns the mutex, no additional work is required. If the test indicates that another thread *does* own the mutex, the current thread blocks on a synchronization event that's part of the *FAST_MUTEX* object. Releasing the mutex entails atomically incrementing and testing the counter. If the test indicates that no thread is currently waiting, no additional work is required. If another thread is waiting, however, the owner calls *KeSetEvent* to release one of the waiters.

Note on Deadlock Prevention
Whenever you use synchronization objects such as spin locks, fast mutexes, and so on, in a driver, you should be on the lookout for potential deadlocks. We've already talked about two deadlock issues: trying to acquire a spin lock that you already hold and trying to claim a fast mutex or synchronization event with APCs enabled in the thread. This sidebar concerns a more insidious potential deadlock that can arise when your driver uses more than one synchronization object.

Suppose there are two synchronization objects, A and B. It doesn't matter what types of objects these are, and they needn't even be the same type. Now suppose we have two subroutines—I'll call them Fred and Barney just so I have names to work with. Subroutine Fred claims object A followed by object B. Subroutine Barney claims B followed by A. This sets up a potential deadlock if Fred and Barney can be simultaneously active or if a thread running one of those routines can be preempted by a thread running the other routine.

The deadlock arises, as you probably remember from studying this sort of thing in school, when two threads manage to execute Fred and Barney at about the same time. The Fred thread gets object A, while the Barney thread gets object B. Fred now tries to get object B, but can't have it (Barney has it). Barney, on the other hand, now tries to get object A, but can't have it (Fred has it). Both threads are now deadlocked, waiting for the other one to release the object each needs.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 123

The easiest way to prevent this kind of deadlock is to always acquire objects such as A and B in the same order, everywhere. The order in which you decide to acquire a set of resources is called the *locking hierarchy*. There are other schemes, which involve conditional attempts to acquire resources combined with back-out loops, but these are much harder to implement.

If you engage the Deadlock Detection option, the Driver Verifier will look for potential deadlocks resulting from locking hierarchy violations involving spin locks, kernel mutexes, and executive fast mutexes.

The DDK documents another synchronization primitive that I didn't discuss in this chapter: an ERESOURCE. File system drivers use ERESOURCE objects extensively because they allow for shared and exclusive ownership. Because file system drivers often have to use complex locking logic, the Driver Verifier doesn't check the locking hierarchy for an ERESOURCE.

4.5.2 Interlocked Arithmetic

You can call several service functions in a WDM driver to perform arithmetic in a way that's thread-safe and multiprocessor-safe. (See Table 4-8.) These routines come in two flavors. The first type of routine has a name beginning with *Interlocked* and performs an atomic operation in such a way that no other thread or CPU can interfere. The other flavor has a name beginning with *ExInterlocked* and uses a spin lock.

Service Function	Description
InterlockedCompareExchange	Compares and conditionally exchanges
InterlockedDecrement	Subtracts 1 from an integer
InterlockedExchange	Exchanges two values
InterlockedExchangeAdd	Adds two values and returns sum
InterlockedIncrement	Adds 1 to an integer
InterlockedOr	ORs bits into an integer
InterlockedAnd	ANDs bits into an integer
InterlockedXor	Exclusive-ORs bits into an integer
ExInterlockedAddLargeInteger	Adds value to 64-bit integer
ExInterlockedAddLargeStatistic	Adds value to *ULONG*
ExInterlockedAddUlong	Adds value to *ULONG* and returns initial value
ExInterlockedCompareExchange64	Exchanges two 64-bit values

Table 4-8. *Service Functions for Interlocked Arithmetic*

The *InterlockedXxx* functions can be called at any IRQL; they can also handle pageable data at *PASSIVE_LEVEL* because they don't require a spin lock. Although the *ExInterlockedXxx* routines can be called at any IRQL, they operate on the target data at or above *DISPATCH_LEVEL* and therefore require a nonpaged argument. The only reason to use an *ExInterlockedXxx* function is if you have a data variable that you sometimes need to increment or decrement and sometimes need to access throughout some series of instructions. You would explicitly claim the spin lock around the multi-instruction accesses and use the *ExInterlockedXxx* function to perform the simple increments or decrements.

InterlockedXxx Functions

InterlockedIncrement adds 1 to a long integer in memory and returns the postincrement value to you:

```
LONG result = InterlockedIncrement(pLong);
```

where *pLong* is the address of a variable typed as a *LONG* (that is, a long integer). Conceptually, the operation of the function is equivalent to the statement return ++*pLong in C, but the implementation differs from that simple statement in order to provide thread safety and multiprocessor safety. *InterlockedIncrement* guarantees that the integer is successfully incremented even if code on other CPUs or in other eligible threads on the same CPU is simultaneously trying to alter the same variable. In the nature of the operation, *InterlockedIncrement* cannot guarantee that the value it returns is still the value of the variable even one machine cycle later because other threads or CPUs will be able to modify the variable as soon as the atomic increment operation completes.

InterlockedDecrement is similar to *InterlockedIncrement*, but it subtracts 1 from the target variable and returns the postdecrement value, just like the C statement return --*pLong but with thread safety and multiprocessor safety.

```
LONG result = InterlockedDecrement(pLong);
```

You call *InterlockedCompareExchange* like this:

IPR2024-00292
KASPERSKY EX 1008 Page 124

```
LONG target;
LONG result = InterlockedCompareExchange(&target, newval, oldval);
```

Here *target* is a long integer used both as input and output to the function, *oldval* is your guess about the current contents of the target, and *newval* is the new value that you want installed in the target if your guess is correct. The function performs an operation similar to that indicated in the following C code but does so via an atomic operation that's both thread-safe and multiprocessor-safe:

```
LONG CompareExchange(PLONG ptarget, LONG newval, LONG oldval)
  {
  LONG value = *ptarget;
  if (value == oldval)
    *ptarget = newval;
  return value;
  }
```

In other words, the function always returns the previous value of the target variable to you. In addition, if that previous value equals *oldval*, it sets the target equal to the *newval* you specify. The function uses an atomic operation to do the compare and exchange so that the replacement happens only if you're correct in your guess about the previous contents.

You can also call the *InterlockedCompareExchangePointer* function to perform a similar sort of compare-and-exchange operation with a pointer. This function is defined either as a compiler-intrinsic (that is, a function for which the compiler supplies an inline implementation) or a real function call, depending on how wide pointers are on the platform for which you're compiling and on the ability of the compiler to generate inline code.

The last function in this class is *InterlockedExchange*, which simply uses an atomic operation to replace the value of an integer variable and to return the previous value:

```
LONG value;
LONG oldval = InterlockedExchange(&value, newval);
```

As you might have guessed, there's also an *InterlockedExchangePointer* that exchanges a pointer value (64-bit or 32-bit, depending on the platform). Be sure to cast the target of the exchange operation to avoid a compiler error when building 64-bit drivers:

```
PIRP Irp = (PIRP) InterlockedExchangePointer( (PVOID*) &foo, NULL);
```

InterlockedOr, *InterlockedAnd* and *InterlockedXor* are new with the XP DDK. You can use them in drivers that will run on earlier Windows versions because they're actually implemented as compiler-intrinsic functions.

Interlocked Fetches and Stores?

A frequently asked question is how to do simple fetch-and-store operations on data that's otherwise being accessed by *InterlockedXxx* functions. You don't have to do anything special to fetch a self-consistent value from a variable that other people are modifying with interlocked operations so long as the data in question is aligned on a natural address boundary. Data so aligned cannot cross a memory cache boundary, and the memory controller will always update a cache-sized memory block atomically. Thus, if someone is updating a variable at about the same time you're trying to read it, you'll get either the preupdate or the postupdate value but never anything in between.

For store operations, however, the answer is more complex. Suppose you write the following sort of code to guard access to some shared data:

```
if (InterlockedExchange(&lock, 42) == 0)
  {
  sharedthing++;
  lock = 0;   // <== don't do this
  }
```

This code will work fine on an Intel x86 computer, where every CPU sees memory writes in the same order. On another type of CPU, though, there could be a problem. One CPU might actually change the memory variable *lock* to 0 before updating memory for the increment statement. That behavior could allow two CPUs to simultaneously access *sharedthing*. This problem could happen because of the way the CPU performs operations in parallel or because of quirks in the memory controller. Consequently, you should rework the code to use an interlocked operation for both changes to lock:

```
if (InterlockedExchange(&lock, 42) == 0)
  {
  sharedthing++;
```


IPR2024-00292
KASPERSKY EX 1008 Page 125

```
    InterlockedExchange(&lock, 0);
    }
```

ExInterlockedXxx Functions

Each of the *ExInterlockedXxx* functions requires that you create and initialize a spin lock before you call it. Note that the operands of these functions must all be in nonpaged memory because the functions operate on the data at elevated IRQL.

```
ExInterlockedAddLargeInteger adds two 64-bit integers and returns the previous value
of the target:
LARGE_INTEGER value, increment;
KSPIN_LOCK spinlock;
LARGE_INTEGER prev = ExInterlockedAddLargeInteger(&value,  increment, &spinlock);
```

Value is the target of the addition and one of the operands. *Increment* is an integer operand that's added to the target. *Spinlock* is a spin lock that you previously initialized. The return value is the target's value before the addition. In other words, the operation of this function is similar to the following function except that it occurs under protection of the spin lock:

```
int64 AddLargeInteger(  int64* pvalue,   int64 increment)
{
  int64 prev = *pvalue;
*pvalue += increment;
return prev;
}
```

Note that the return value is the *preaddition* value, which contrasts with the postincrement return from *InterlockedExchange* and similar functions. (Also, not all compilers support the *__int64* integer data type, and not all computers can perform a 64-bit addition operation using atomic instructions.)

ExInterlockedAddUlong is analogous to *ExInterlockedAddLargeInteger* except that it works with 32-bit unsigned integers:

```
ULONG value, increment;
KSPIN_LOCK spinlock;
ULONG prev = ExInterlockedAddUlong(&value, increment, &spinlock);
```

This function likewise returns the preaddition value of the target of the operation.

ExInterlockedAddLargeStatistic is similar to *ExInterlockedAddUlong* in that it adds a 32-bit value to a 64-bit value:

```
VOID ExInterlockedAddLargeStatistic(PLARGE_INTEGER Addend, ULONG Increment);
```

This new function is faster than *ExInterlockedAddUlong* because it doesn't need to return the preincrement value of the Addend variable. It therefore doesn't need to employ a spin lock for synchronization. The atomicity provided by this function is, however, only with respect to other callers of the same function. In other words, if you had code on one CPU calling *ExInterlockedAddLargeStatistic* at the same time as code on another CPU was accessing the *Addend* variable for either reading or writing, you could get inconsistent results. I can explain why this is so by showing you this paraphrase of the Intel x86 implementation of the function (not the actual source code):

```
mov eax, Addend
mov ecx, Increment
lock add [eax], ecx
lock adc [eax+4], 0
```

This code works correctly for purposes of incrementing the *Addend* because the lock prefixes guarantee atomicity of each addition operation and because no carries from the low-order 32 bits can ever get lost. The instantaneous value of the 64-bit *Addend* isn't always consistent, however, because an incrementer might be poised between the ADD and the ADC just at the instant someone makes a copy of the complete 64-bit value. Therefore, even a caller of *ExInterlockedCompareExchange64* on another CPU could obtain an inconsistent value.

4.5.3 Interlocked List Access

The Windows NT executive offers three sets of support functions for dealing with linked lists in a thread-safe and multiprocessor-safe way. These functions support doubly-linked lists, singly-linked lists, and a special kind of singly-linked list called an *S-List*. I discussed noninterlocked doubly-linked and singly-linked lists in the preceding chapter. To close this chapter on synchronization within WDM drivers, I'll explain how to use these interlocked accessing primitives.

If you need the functionality of a FIFO queue, you should use a doubly-linked list. If you need the functionality of a

IPR2024-00292
KASPERSKY EX 1008 Page 126

thread-safe and multiprocessor-safe pushdown stack, you should use an S-List. In both cases, to achieve thread safety and multiprocessor safety, you will allocate and initialize a spin lock. The S-List might not actually use the spin lock, however, because the presence of a sequence number might allow the kernel to implement it using just atomic compare-exchange sorts of operations.

The support functions for performing interlocked access to list objects are similar, so I've organized this section along functional lines. I'll explain how to initialize all three kinds of lists. Then I'll explain how to insert an item into all three kinds. After that, I'll explain how to remove items.

Initialization

You can initialize these lists as shown here:

```
LIST ENTRY DoubleHead;
SINGLE LIST ENTRY SingleHead;
SLIST HEADER SListHead;

InitializeListHead(&DoubleHead);

SingleHead.Next = NULL;

ExInitializeSListHead(&SListHead);
```

Don't forget that you must also allocate and initialize a spin lock for each list. Furthermore, the storage for the list heads and all the items you put into the lists must come from nonpaged memory because the support routines perform their accesses at elevated IRQL. Note that the spin lock isn't used during initialization of the list head because it doesn't make any sense to allow contention for list access before the list has been initialized.

Inserting Items

You can insert items at the head and tail of a doubly-linked list and at the head (only) of a singly-linked list or an S-List:

```
PLIST ENTRY pdElement, pdPrevHead, pdPrevTail;
PSINGLE LIST ENTRY psElement, psPrevHead;
PKSPIN LOCK spinlock;

pdPrevHead = ExInterlockedInsertHeadList(&DoubleHead, pdElement, spinlock);
pdPrevTail = ExInterlockedInsertTailList(&DoubleHead, pdElement, spinlock);

psPrevHead = ExInterlockedPushEntryList(&SingleHead, psElement, spinlock);

psPrevHead = ExInterlockedPushEntrySList(&SListHead, psElement, spinlock);
```

The return values are the addresses of the elements previously at the head (or tail) of the list in question. Note that the element addresses you use with these functions are the addresses of list entry structures that are usually embedded in larger structures of some kind, and you'll need to use the *CONTAINING_RECORD* macro to recover the address of the surrounding structure.

Removing Items

You can remove items from the head of any of these lists:

```
pdElement = ExInterlockedRemoveHeadList(&DoubleHead, spinlock);

psElement = ExInterlockedPopEntryList(&SingleHead, spinlock);

psElement = ExInterlockedPopEntrySList(&SListHead, spinlock);
```

The return values are *NULL* if the respective lists are empty. Be sure to test the return value for *NULL before* applying the *CONTAINING_RECORD* macro to recover a containing structure pointer.

IRQL Restrictions

You can call the S-List functions only while running at or below *DISPATCH_LEVEL*. The *ExInterlockedXxx* functions for accessing doubly-linked or singly-linked lists can be called at any IRQL so long as all references to the list use an *ExInterlockedXxx* call. The reason for no IRQL restrictions is that the implementations of these functions disable interrupts, which is tantamount to raising IRQL to the highest possible level. Once interrupts are disabled, these functions then acquire the spin lock you've specified. Since no other code can gain control on the same CPU, and since no code on another CPU can

IPR2024-00292
KASPERSKY EX 1008 Page 127

acquire the spin lock, your lists are protected.

NOTE
The DDK documentation states this rule in an overly restrictive way for at least some of the *ExInterlockedXxx* functions. It says that all callers must be running at some single IRQL less than or equal to the DIRQL of your interrupt object. There is, in fact, no requirement that all callers be at the same IRQL because you can call the functions at any IRQL. Likewise, no <= DIRQL restriction exists either, but there's also no reason for the code you and I write to raise IRQL higher than that.

It's perfectly OK for you to use *ExInterlockedXxx* calls to access a singly-linked or doubly-linked list (but not an S-List) in some parts of your code and to use the noninterlocked functions (*InsertHeadList* and so on) in other parts of your code if you follow a simple rule. Before using a noninterlocked primitive, acquire the same spin lock that your interlocked calls use. Furthermore, restrict list access to code running at or below *DISPATCH_LEVEL*. For example:

```
// Access list using noninterlocked calls:

VOID Function1()
  {
  ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
  KIRQL oldirql;
  KeAcquireSpinLock(spinlock, &oldirql);
  InsertHeadList(...);
  RemoveTailList(...);
  .
  .
  KeReleaseSpinLock(spinlock, oldirql);
  }

// Access list using interlocked calls:

VOID Function2()
  {
  ASSERT(KeGetCurrentIrql() <= DISPATCH_LEVEL);
  ExInterlockedInsertTailList(..., spinlock);
  }
```

The first function must be running at or below *DISPATCH_LEVEL* because that's a requirement of calling *KeAcquireSpinLock*. The reason for the IRQL restriction on the interlocked calls in the second function is as follows: Suppose *Function1* acquires the spin lock in preparation for performing some list accesses. Acquiring the spin lock raises IRQL to *DISPATCH_LEVEL*. Now suppose an interrupt occurs on the same CPU at a higher IRQL and *Function2* gains control to use one of the *ExInterlockedXxx* routines. The kernel will now attempt to acquire the same spin lock, and the CPU will deadlock. This problem arises from allowing code running at two different IRQLs to use the same spin lock: *Function1* is at *DISPATCH_LEVEL*, and *Function2* is—practically speaking, anyway—at *HIGH_LEVEL* when it tries to recursively acquire the lock.

4.5.4 Windows 98/Me Compatibility Notes

In addition to the horrible problem with *KeWaitXxx* functions described in an earlier sidebar and the fact that *KeReadStateEvent* isn't supported, note the following additional compatibility issues between Windows 98/Me on the one hand and Windows 2000/XP on the other.

You cannot wait on a *KTHREAD* object in Windows 98/Me. Attempting to do so crashes the system because the thread object doesn't have the fields necessary for *VWIN32* to wait on it.

DISPATCH_LEVEL in a WDM driver corresponds to what is called "interrupt time" in a VxD driver. Every WDM interrupt service routine runs at a higher IRQL, which means that WDM interrupts have higher priority than non-WDM interrupts. If a WDM device shares an interrupt with a VxD device, however, both interrupt routines run at the WDM driver's DIRQL.

WDM driver code running at *PASSIVE_LEVEL* won't be preempted in Windows 98/Me unless it blocks explicitly by waiting for a dispatcher object or implicitly by causing a page fault.

Windows 98/Me is inherently a single-CPU operating system, so the spin lock primitives always just raise the IRQL. This fact, combined with the fact that nonpaged driver code won't be preempted, means that synchronization problems are much less frequent in this environment. (Therefore, do most of your debugging in Windows XP so you'll trip on the problems.)

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 128

Chapter 5

The I/O Request Packet

The operating system uses a data structure known as an I/O request packet, or IRP, to communicate with a kernel-mode device driver. In this chapter, I'll discuss this important data structure and the means by which it's created, sent, processed, and ultimately destroyed. I'll include a discussion of the relatively complex subject of IRP cancellation.

This chapter is rather abstract, I'm afraid, because I haven't yet talked about any of the concepts that surround specific types of I/O request packets (IRPs). You might, therefore, want to skim this chapter and refer back to it while you're reading later chapters. The last major section of this chapter contains a cookbook, if you will, that presents the bare-bones code for handling IRPs in eight different scenarios. You can use the cookbook without understanding all the theory that this chapter contains.

5.1 Data Structures

Two data structures are crucial to the handling of I/O requests: the I/O request packet itself and the *IO_STACK_LOCATION* structure. I'll describe both structures in this section.

5.1.1 Structure of an IRP

Figure 5-1 illustrates the IRP data structure, with opaque fields shaded in the usual convention of this book. A brief description of the important fields follows.

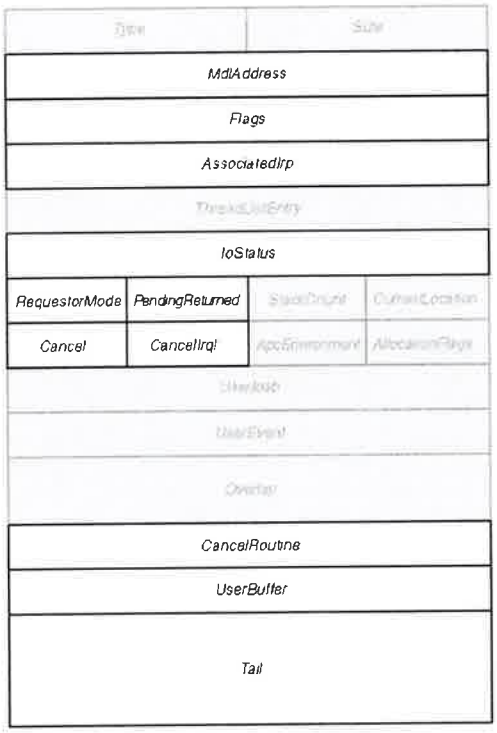


Figure 5-1. *I/O request packet data structure.*

MdlAddress (*PMDL*) is the address of a memory descriptor list (MDL) describing the user-mode buffer associated with this request. The I/O Manager creates this MDL for *IRP_MJ_READ* and *IRP_MJ_WRITE* requests if the topmost device object's flags indicate *DO_DIRECT_IO*. It creates an MDL for the output buffer used with an *IRP_MJ_DEVICE_CONTROL* request if the control code indicates *METHOD_IN_DIRECT* or *METHOD_OUT_DIRECT*. The MDL itself describes the user-mode virtual buffer and also contains the physical addresses of locked pages containing that buffer. A driver has to do additional work, which can be quite minimal, to actually access the user-mode buffer.

IPR2024-00292
KASPERSKY EX 1008 Page 129

Flags (*ULONG*) contains flags that a device driver can read but not directly alter. None of these flags are relevant to a Windows Driver Model (WDM) driver.

AssociatedIrp (union) is a union of three possible pointers. The alternative that a typical WDM driver might want to access is named *AssociatedIrp.SystemBuffer*. The *SystemBuffer* pointer holds the address of a data buffer in nonpaged kernel-mode memory. For *IRP_MJ_READ* and *IRP_MJ_WRITE* operations, the I/O Manager creates this data buffer if the topmost device object's flags specify *DO_BUFFERED_IO*. For *IRP_MJ_DEVICE_CONTROL* operations, the I/O Manager creates this buffer if the I/O control function code indicates that it should. (See Chapter 9.) The I/O Manager copies data sent by user-mode code to the driver into this buffer as part of the process of creating the IRP. Such data includes the data involved in a *WriteFile* call or the so-called input data for a call to *DeviceIoControl*. For read requests, the device driver fills this buffer with data; the I/O Manager later copies the buffer back to the user-mode buffer. For control operations that specify *METHOD_BUFFERED*, the driver places the so-called output data in this buffer, and the I/O Manager copies it to the user-mode output buffer.

IoStatus (*IO_STATUS_BLOCK*) is a structure containing two fields that drivers set when they ultimately complete a request. *IoStatus.Status* will receive an *NTSTATUS* code, while *IoStatus.Information* is a *ULONG_PTR* that will receive an information value whose exact content depends on the type of IRP and the completion status. A common use of the Information field is to hold the total number of bytes transferred by an operation such as *IRP_MJ_READ* that transfers data. Certain Plug and Play (PnP) requests use this field as a pointer to a structure that you can think of as the answer to a query.

RequestorMode will equal one of the enumeration constants *UserMode* or *KernelMode*, depending on where the original I/O request originated. Drivers sometimes inspect this value to know whether to trust some parameters.

PendingReturned (*BOOLEAN*) is meaningful in a completion routine and indicates whether the next lower dispatch routine returned *STATUS_PENDING*. This chapter contains a disagreeably long discussion of how to use this flag.

Cancel (*BOOLEAN*) is *TRUE* if *IoCancelIrp* has been called to cancel this request and *FALSE* if it hasn't (yet) been called. IRP cancellation is a relatively complex topic that I'll discuss fully later on in this chapter (in "Cancelling I/O Requests").

CancelIrql (*KIRQL*) is the interrupt request level (IRQL) at which the special cancel spin lock was acquired. You reference this field in a cancel routine when you release the spin lock.

CancelRoutine (*PDRIVER_CANCEL*) is the address of an IRP cancellation routine in your driver. You use *IoSetCancelRoutine* to set this field instead of modifying it directly.

UserBuffer (*PVOID*) contains the user-mode virtual address of the output buffer for an *IRP_MJ_DEVICE_CONTROL* request for which the control code specifies *METHOD_NEITHER*. It also holds the user-mode virtual address of the buffer for read and write requests, but a driver should usually specify one of the device flags *DO_BUFFERED_IO* or *DO_DIRECT_IO* and should therefore not usually need to access the field for reads or writes. When handling a *METHOD_NEITHER* control operation, the driver can create its own MDL using this address.

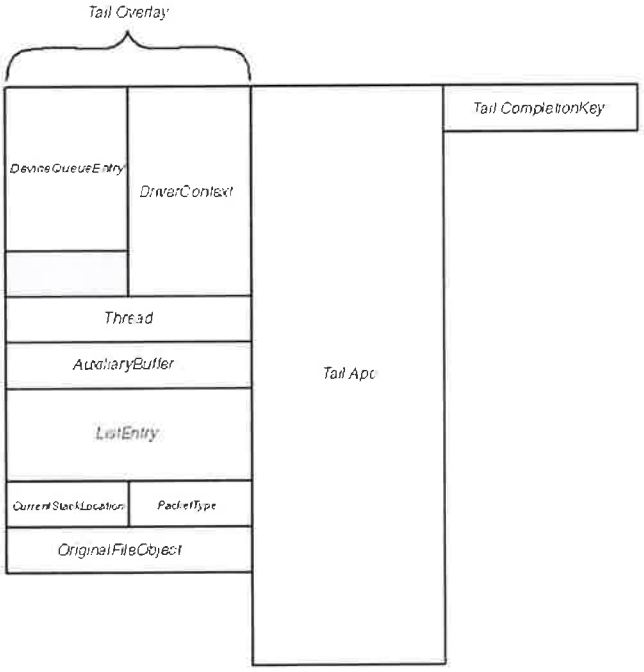


Figure 5-2. *Map of the Tail union in an IRP.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 130

Tail.Overlay is a structure within a union that contains several members potentially useful to a WDM driver. Refer to Figure 5-2 for a map of the Tail union. In the figure, items at the same level as you read left to right are alternatives within a union, while the vertical dimension portrays successive locations within a structure. *Tail.Overlay.DeviceQueueEntry* (*KDEVICE_QUEUE_ENTRY*) and *Tail.Overlay.DriverContext* (*PVOID[4]*) are alternatives within an unnamed union within *Tail.Overlay*. The I/O Manager uses *DeviceQueueEntry* as a linking field within the standard queue of requests for a device. The cancel-safe queuing routines *IoCsqXxx* use the last entry in the *DriverContext* array. If these system usages don't get in your way, at moments when the IRP is not in some queue that uses this field and when you own the IRP, you can use the four pointers in *DriverContext* in any way you please. *Tail.Overlay.ListEntry (LIST_ENTRY)* is available for you to use as a linking field for IRPs in any private queue you choose to implement.

CurrentLocation (*CHAR*) and *Tail.Overlay.CurrentStackLocation* (*PIO_STACK_LOCATION*) aren't documented for use by drivers because support functions such as *IoGetCurrentIrpStackLocation* can be used instead. During debugging, however, it might help you to realize that *CurrentLocation* is the index of the current I/O stack location and *CurrentStackLocation* is a pointer to it.

5.1.2 The I/O Stack

Whenever any kernel-mode program creates an IRP, it also creates an associated array of *IO_STACK_LOCATION* structures: one stack location for each of the drivers that will process the IRP and sometimes one more stack location for the use of the originator of the IRP. (See Figure 5-3.) A stack location contains type codes and parameter information for the IRP as well as the address of a completion routine. Refer to Figure 5-4 for an illustration of the stack structure.

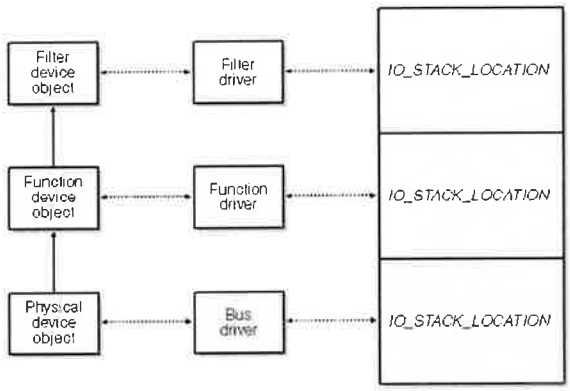


Figure 5-3. *Parallelism between driver and I/O stacks.*

NOTE
I'll discuss the mechanics of creating IRPs a bit further on in this chapter. It helps to know right now that the *StackSize* field of a *DEVICE_OBJECT* indicates how many locations to reserve for an IRP sent to that device's driver.

MajorFunction (*UCHAR*) is the major function code associated with this IRP. This code is a value such as *IRP_MJ_READ* that corresponds to one of the dispatch function pointers in the *MajorFunction* table of a driver object. Because the code is in the I/O stack location for a particular driver, it's conceivable that an IRP could start life as an *IRP_MJ_READ* (for example) and be transformed into something else as it progresses down the stack of drivers. I'll show you examples in Chapter 12 of how a USB driver changes the personality of a read or write request into an internal control operation to submit the request to the USB bus driver.

MinorFunction (*UCHAR*) is a minor function code that further identifies an IRP belonging to a few major function classes. *IRP_MJ_PNP* requests, for example, are divided into a dozen or so subtypes with minor function codes such as *IRP_MN_START_DEVICE*, *IRP_MN_REMOVE_DEVICE*, and so on.

Parameters (union) is a union of substructures, one for each type of request that has specific parameters. The substructures include, for example, *Create (for IRP_MJ_CREATE requests)*, *Read (for IRP_MJ_READ requests)*, and *StartDevice* (for the *IRP_MN_START_DEVICE* subtype of *IRP_MJ_PNP*).

DeviceObject (PDEVICE_OBJECT) is the address of the device object that corresponds to this stack entry. *IoCallDriver* fills in this field.

FileObject (*PFILE_OBJECT*) is the address of the kernel file object to which the IRP is directed. Drivers often use the *FileObject* pointer to correlate IRPs in a queue with a request (in the form of an *IRP_MJ_CLEANUP*) to cancel all queued IRPs in preparation for closing the file object.

IPR2024-00292
KASPERSKY EX 1008 Page 131

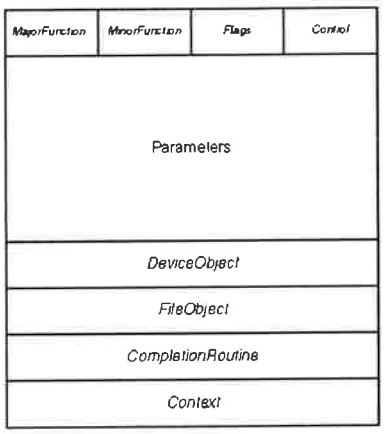


Figure 5-4. *I/O stack location data structure.*

CompletionRoutine (PIO_COMPLETION_ROUTINE) is the address of an I/O completion routine installed by the driver above the one to which this stack *location corresponds*. You never set this field directly—instead, you call *IoSetCompletionRoutine*, which knows to reference the stack location below the one that your driver owns. The lowest-level driver in the hierarchy of drivers for a given device never needs a completion routine because it must complete the request. The originator of a request, however, sometimes does need a completion routine but doesn't usually have its own stack location. That's why each level in the hierarchy uses the next lower stack location to hold its own completion routine pointer.

Context (PVOID) is an arbitrary context value that will be passed as an argument to the completion routine. You never set this field directly; it's set automatically from one of the arguments to *IoSetCompletionRoutine*.

5.2 The "Standard Model" for IRP Processing

Particle physics has its "standard model" for the universe, and so does WDM. Figure 5-5 illustrates a typical flow of ownership for an IRP as it progresses through various stages in its life. Not every type of IRP will go through these steps, and some of the steps might be missing or altered depending on the type of device and the type of IRP. Notwithstanding the possible variability, however, the picture provides a useful starting point for discussion.

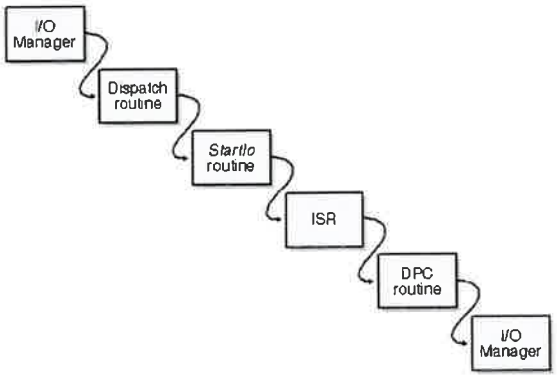


Figure 5-5. *The "standard model" for IRP processing.*

☑ When you engage I/O Verification, the Driver Verifier makes a few basic checks on how you handle IRPs. Extended I/O Verification includes many more checks. Because there are so many tests, however, I didn't put the Driver Verifier flag in the margin for every one of them. Basically, if the DDK or this chapter tells you not to do something, there is probably a Driver Verifier test to make sure you don't.

5.2.1 Creating an IRP

The IRP begins life when an entity calls an I/O Manager function to create it. In Figure 5-5, I used the term I/O Manager to describe this entity, as though there were a single system component responsible for creating IRPs. In reality, no such single actor in the population of operating system routines exists, and it would have been more accurate to just say that somebody creates the IRP. Your own driver will be creating IRPs from time to time, for example, and you'll occupy the initial ownership

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 132

box for those particular IRPs.

You can use any of four functions to create a new IRP:

- *IoBuildAsynchronousFsdRequest* builds an IRP on whose completion you don't plan to wait. This function and the next are appropriate for building only certain types of IRP.
- *IoBuildSynchronousFsdRequest* builds an IRP on whose completion you do plan to wait.
- *IoBuildDeviceIoControlRequest* builds a synchronous *IRP_MJ_DEVICE_CONTROL* or *IRP_MJ_INTERNAL_DEVICE_CONTROL* request.
- *IoAllocateIrp* builds an asynchronous IRP of any type.

The *Fsd* in the first two of these function names stands for *file system driver* (*FSD*). Any driver is allowed to call these functions to create an IRP destined for any other driver, though. The DDK also documents a function named *IoMakeAssociatedIrp* for building an IRP that's subordinate to some other IRP. WDM drivers should not call this function. Indeed, completion of associated IRPs doesn't work correctly in Microsoft Windows 98/Me anyway.

NOTE
Throughout this chapter, I use the terms *synchronous* and *asynchronous* IRPs because those are the terms used in the DDK. Knowledgeable developers in Microsoft wish that the terms *threaded* and *nonthreaded* had been chosen because they better reflect the way drivers use these two types of IRP. As should become clear, you use a synchronous, or threaded, IRP in a non-arbitrary thread that you can block while you wait for the IRP to finish. You use an asynchronous, or nonthreaded, IRP in every other case.

Creating Synchronous IRPs

Deciding which of these functions to call and determining what additional initialization you need to perform on an IRP is a rather complicated matter. *IoBuildSynchronousFsdRequest* and *IoBuildDeviceIoControlRequest* create a so-called synchronous IRP. The I/O Manager considers that a synchronous IRP belongs to the thread in whose context you create the IRP. This ownership concept has several consequences:

- If the owning thread terminates, the I/O Manager automatically cancels any pending synchronous IRPs that belong to that thread.
- Because the creating thread owns a synchronous IRP, you shouldn't create one in an arbitrary thread—you most emphatically do not want the I/O Manager to cancel the IRP because this thread happens to terminate.
- Following a call to *IoCompleteRequest*, the I/O Manager automatically cleans up a synchronous IRP and signals an event that you must provide.
- You must take care that the event object still exists at the time the I/O Manager signals it.

Refer to IRP handling scenario number 6 at the end of this chapter for a code sample involving a synchronous IRP.

You must call these two functions at *PASSIVE_LEVEL* only. In particular, you must not be at *APC_LEVEL* (say, as a result of acquiring a fast mutex) because the I/O Manager won't then be able to deliver the special kernel asynchronous procedure call (APC) that does all the completion processing. In other words, you mustn't do this:

```
PIRP Irp = IoBuildSynchronousFsdRequest(...);
ExAcquireFastMutex(...);
NTSTATUS status = IoCallDriver(...);
if (status == STATUS_PENDING)
  KeWaitForSingleObject(...);  // <== don't do this
ExReleaseFastMutex(...);
```

The problem with this code is that the *KeWaitForSingleObject* call will deadlock: when the IRP completes, *IoCompleteRequest* will schedule an APC in this thread. The APC routine, if it could run, would set the event. But because you're already at *APC_LEVEL*, the APC cannot run in order to set the event.

If you need to synchronize IRPs sent to another driver, consider the following alternatives:

- Use a regular kernel mutex instead of an executive fast mutex. The regular mutex leaves you at *PASSIVE_LEVEL* and doesn't inhibit special kernel APCs.
- Use *KeEnterCriticalRegion* to inhibit all but special kernel APCs, and then use *ExAcquireFastMutexUnsafe* to acquire the mutex. This technique won't work in the original release of Windows 98 because *KeEnterCriticalRegion* wasn't supported there. It will work on all later WDM platforms.
- Use an asynchronous IRP. Signal an event in the completion routine. Refer to IRP-handling scenario 8 at the end of this chapter for a code sample.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 133

A final consideration in calling the two synchronous IRP routines is that you can't create just any kind of IRP using these routines. See Table 5-1 for the details. A common trick for creating another kind of synchronous IRP is to ask for an *IRP_MJ_SHUTDOWN*, which has no parameters, and then alter the *MajorFunction* code in the first stack location.

Support Function	Types of IRP You Can Create
IoBuildSynchronousFsdRequest	*IRP_MJ_READ* *IRP_MJ_WRITE* *IRP_MJ_FLUSH_BUFFERS* *IRP_MJ_SHUTDOWN* *IRP_MJ_PNP* *IRP_MJ_POWER* (but only for *IRP_MN_POWER_SEQUENCE*)
IoBuildDeviceIoControlRequest	*IRP_MJ_DEVICE_CONTROL* *IRP_MJ_INTERNAL_DEVICE_CONTROL*

Table 5-1. *Synchronous IRP Types*

Creating Asynchronous IRPs

The other two IRP creation functions—*IoBuildAsynchronousFsdRequest* and *IoAllocateIrp*—create an asynchronous IRP. Asynchronous IRPs don't belong to the creating thread, and the I/O Manager doesn't schedule an APC and doesn't clean up when the IRP completes. Consequently:

- When a thread terminates, the I/O Manager doesn't try to cancel any asynchronous IRPs that you happen to have created in that thread.
- It's OK to create asynchronous IRPs in an arbitrary or nonarbitrary thread.
- Because the I/O Manager doesn't do any cleanup when the IRP completes, you must provide a completion routine that will release buffers and call *IoFreeIrp* to release the memory used by the IRP.
- Because the I/O Manager doesn't automatically cancel asynchronous IRPs, you might have to provide code to do that when you no longer want the operation to occur.
- Because you don't wait for an asynchronous IRP to complete, you can create and send one at *IRQL <= DISPATCH_LEVEL* (assuming, that is, that the driver to which you send the IRP can handle the IRP at elevated IRQL—you must check the specifications for that driver!). Furthermore, it's OK to create and send an asynchronous IRP while owning a fast mutex.

Refer to Table 5-2 for a list of the types of IRP you can create using the two asynchronous IRP routines. Note that *IoBuildSynchronousFsdRequest* and *IoBuildAsynchronousFsdRequest* support the same IRP major function codes.

Support Function	Types of IRP You Can Create
IoBuildAsynchronousFsdRequest	*IRP_MJ_READ* *IRP_MJ_WRITE* *IRP_MJ_FLUSH_BUFFERS* *IRP_MJ_SHUTDOWN* *IRP_MJ_PNP* *IRP_MJ_POWER* (but only for *IRP_MN_POWER_SEQUENCE*)
IoAllocateIrp	Any (but you must initialize the MajorFunction field of the first stack location)

Table 5-2. *Asynchronous IRP Types*

IRP-handling scenario numbers 5 and 8 at the end of this chapter contain "cookbook" code for using asynchronous IRPs.

5.2.2 Forwarding to a Dispatch Routine

After you create an IRP, you call *IoGetNextIrpStackLocation* to obtain a pointer to the first stack location. Then you initialize just that first location. If you've used *IoAllocateIrp* to create the IRP, you need to fill in at least the *MajorFunction* code. If you've used another of the four IRP-creation functions, the I/O Manager might have already done the required initialization. You might then be able to skip this step, depending on the rules for that particular type of IRP. Having initialized the stack, you call *IoCallDriver* to send the IRP to a device driver:

```
PDEVICE OBJECT DeviceObject; // <== somebody gives you this
PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP MJ Xxx;
<other initialization of
"stack">NTSTATUS status = IoCallDriver(DeviceObject, Irp);
```

The first argument to *IoCallDriver* is the address of a device object that you've obtained somehow. Often you're sending an

IPR2024-00292
KASPERSKY EX 1008 Page 134

IRP to the driver under yours in the PnP stack. In that case, the *DeviceObject* in this fragment is the *LowerDeviceObject* you saved in your device extension after calling *IoAttachDeviceToDeviceStack*. I'll describe some other common ways of locating a device object in a few paragraphs.

The I/O Manager initializes the stack location pointer in the IRP to 1 before the actual first location. Because the I/O stack is an array of *IO_STACK_LOCATION* structures, you can think of the stack pointer as being initialized to point to the "-1" element, which doesn't exist. (In fact, the stack "grows" from high toward low addresses, but that detail shouldn't obscure the concept I'm trying to describe here.) We therefore ask for the "next" stack location when we want to initialize the first one.

What IoCallDriver Does

You can imagine *IoCallDriver* as looking something like this (but I hasten to add that this is not a copy of the actual source code):

```
NTSTATUS IoCallDriver(PDEVICE OBJECT DeviceObject, PIRP Irp)
  {
  IoSetNextIrpStackLocation(Irp);
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  stack->DeviceObject = DeviceObject;
  ULONG fcn = stack->MajorFunction;
  PDRIVER OBJECT driver = DeviceObject->DriverObject;
  return (*driver->MajorFunction[fcn])(DeviceObject, Irp);
  }
```

As you can see, *IoCallDriver* simply advances the stack pointer and calls the appropriate dispatch routine in the driver for the target device object. It returns the status code that that dispatch routine returns. Sometimes I see online help requests wherein people attribute one or another unfortunate action to *IoCallDriver*. (For example, "*IoCallDriver* is returning an error code for my IRP....") As you can see, the real culprit is a dispatch routine in another driver.

Locating Device Objects

Apart from *IoAttachDeviceToDeviceStack*, drivers can locate device objects in at least two ways. I'll tell you here about *IoGetDeviceObjectPointer* and *IoGetAttachedDeviceReference*.

IoGetDeviceObjectPointer

If you know the name of the device object, you can call *IoGetDeviceObjectPointer* as shown here:

```
PUNICODE STRING devname; // <== somebody gives you this
ACCESS MASK access;      // <== more about this later
PDEVICE OBJECT DeviceObject;
PFILE OBJECT FileObject;
NTSTATUS status;
ASSERT(KeGetCurrentIrql() == PASSIVE LEVEL);
status = IoGetDeviceObjectPointer(devname, access, &FileObject, &DeviceObject);
```

This function returns two pointers: one to a *FILE_OBJECT* and one to a *DEVICE_OBJECT*.



To help defeat elevation-of-privilege attacks, specify the most restricted access consistent with your needs. For example, if you'll just be reading data, specify *FILE_READ_DATA*.

When you create an IRP for a target you discover this way, you should set the *FileObject* pointer in the first stack location. Furthermore, it's a good idea to take an extra reference to the file object until after *IoCallDriver* returns. The following fragment illustrates both these ideas:

```
PIRP Irp = IoXxx(...);
PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
ObReferenceObject(FileObject);
stack->FileObject = FileObject;<etc.>
IoCallDriver(DeviceObject, Irp);
ObDereferenceObject(FileObject);
```

The reason you put the file object pointer in each stack location is that the target driver might be using fields in the file object to record per-handle information. The reason you take an extra reference to the file object is that you'll have code somewhere in your driver that dereferences the file object in order to release your hold on the target device. (See the next paragraph.) Should that code execute before the target driver's dispatch routine returns, the target driver might be removed from memory before its dispatch routine returns. The extra reference prevents that bad result.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 135

NOTE
Removability of devices in a Plug and Play environment is the ultimate source of the early-unload problem mentioned in the text. I discuss this problem in much greater detail in the next chapter. The upshot of that discussion is that it's your responsibility to avoid sending an IRP to a driver that might no longer be in memory and to prevent the PnP manager from unloading a driver that's still processing an IRP you've sent to that driver. One aspect of how you fulfill that responsibility is shown in the text: take an extra reference to the file object returned by *IoGetDeviceObjectPointer* around the call to *IoCallDriver*. In most drivers, you'll probably need the extra reference only when you're sending an asynchronous IRP. In that case, the code that ordinarily dereferences the file object is likely to be in some other part of your driver that runs asynchronously with the call to *IoCallDriver*—say, in the completion routine you're obliged to install for an asynchronous IRP. If you send a synchronous IRP, you're much more likely to code your driver in such a way that you don't dereference the file object until the IRP completes.

When you no longer need the device object, dereference the file object:

```
ObDereferenceObject(FileObject);
```

After making this call, don't use either of the file or device object pointers.

IoGetDeviceObjectPointer performs several steps to locate the two pointers that it returns to you:

1. It uses *ZwOpenFile* to open a kernel handle to the named device object. Internally, this will cause the Object Manager to create a file object and to send an *IRP_MJ_CREATE* to the target device. *ZwOpenFile* returns a file handle.
2. It calls *ObReferenceObjectByHandle* to get the address of the *FILE_OBJECT* that the handle represents. This address becomes the *FileObject* return value.
3. It calls *IoGetRelatedDeviceObject* to get the address of the *DEVICE_OBJECT* to which the file object refers. This address becomes the *DeviceObject* return value.
4. It calls *ZwClose* to close the handle.

Names for Device Objects
For you to use IoGetDeviceObjectPointer, a driver in the stack for the device to which you want to connect must have named a device object. We studied device object naming in Chapter 2. Recall that a driver might have specified a name in the \Device folder in its call to IoCreateDevice, and it might have created one or more symbolic links in the \DosDevices folder. If you know the name of the device object or one of the symbolic links, you can use that name in your call to IoGetDeviceObjectPointer.

Instead of naming a device object, the function driver for the target device might have registered a device interface. I showed you the user-mode code for enumerating instances of registered interfaces in Chapter 2. I'll discuss the kernel-mode equivalent of that enumeration code in Chapter 6, when I discuss Plug and Play. The upshot of that discussion is that you can obtain the symbolic link names for all the devices that expose a particular interface. With a bit of effort, you can then locate the desired device object.

The reference that *IoGetDeviceObjectPointer* claims to the file object effectively pins the device object in memory too. Releasing that reference indirectly releases the device object.

Based on this explanation of how *IoGetDeviceObjectPointer* works, you can see why it will sometimes fail with *STATUS_ACCESS_DENIED*, even though you haven't done anything wrong. If the target driver implements a "one handle only" policy, and if a handle happens to be open, the driver will cause the *IRP_MJ_CREATE* to fail. That failure causes the *ZwOpenFile* call to fail in turn. Note that you can expect this result if you try to locate a device object for a serial port or *SmartCard* reader that happens to already be open.

Sometimes driver programmers decide they don't want the clutter of two pointers to what appears to be basically the same object, so they release the file object immediately after calling *IoGetDeviceObjectPointer*, as shown here:

```
status = IoGetDeviceObjectPointer(...);
ObReferenceObject(DeviceObject);
ObDereferenceObject(FileObject);
```

Referencing the device object pins it in memory until you dereference it. Dereferencing the file object allows the I/O Manager to delete it right away.

Releasing the file object immediately might or might not be OK, depending on the target driver. Consider these fine points before you decide to do it:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 136

1. Deferencing the file object will cause the I/O Manager to send an immediate *IRP_MJ_CLEANUP* to the target driver.
2. IRPs that the target driver queues will no longer be associated with a file object. When you eventually release the device object reference, the target driver will probably not be able to cancel any IRPs you sent it that remain on its queues.
3. In many situations, the I/O Manager will also send an *IRP_MJ_CLOSE* to the target driver. (If you've opened a disk file, the file system driver's use of the system cache will probably cause the *IRP_MJ_CLOSE* to be deferred.) Many drivers, including the standard driver for serial ports, will now refuse to process IRPs that you send them.
4. Instead of claiming an extra reference to the file object around calls to *IoCallDriver*, you'll want to reference the device object instead.

NOTE
I recommend avoiding an older routine named *IoAttachDevice*, which appears superficially to be a sort-of combination of *IoGetDeviceObjectPointer* and *IoAttachDeviceToDeviceStack*. The older routine does its internal *ZwClose* call after attaching your device object. Your driver will receive the resulting *IRP_MJ_CLOSE*. To handle the IRP correctly, you must call *IoAttachDevice* in such a way that your dispatch routine has access to the location you specify for the output *DEVICE_OBJECT* pointer. It turns out that *IoAttachDevice* sets your output pointer before calling *ZwClose* and depends on you using it to forward the *IRP_MJ_CLOSE* to the target device. This is the only example I've seen in many decades of programming where you're required to use the return value from a function before the function actually returns.

IoGetAttachedDeviceReference

To send an IRP to all the drivers in your own PnP stack, use IoGetAttachedDeviceReference, as shown here:

```
PDEVICE_OBJECT tdo = IoGetAttachedDeviceReference(fdo);
:
ObDereferenceObject(tdo);
```

This function returns the address of the topmost device object in your own stack and claims a reference to that object. Because of the reference you hold, you can be sure that the pointer will remain valid until you release the reference. As discussed earlier, you might also want to take an extra reference to the topmost device object until *IoCallDriver* returns.

5.2.3 Duties of a Dispatch Routine

An archetypal IRP dispatch routine would look similar to this example:

```
NTSTATUS DispatchXxx(PDEVICE_OBJECT fdo, PIRP Irp)
   {
1
   PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
2
   PDEVICE_EXTENSION pdx =
      (PDEVICE_EXTENSION) fdo->DeviceExtension;
:
:
3
   return STATUS_Xxx;
   }
```

1. You generally need to access the current stack location to determine parameters or to examine the minor function code.
2. You also generally need to access the device extension you created and initialized during *AddDevice*.
3. You'll be returning some *NTSTATUS* code to *IoCallDriver*, which will propagate the code back to its caller.

Where I used an ellipsis in the foregoing prototypical dispatch function, a dispatch function has to choose between three courses of action. It can complete the request immediately, pass the request down to a lower-level driver in the same driver stack, or queue the request for later processing by other routines in this driver.

Completing an IRP

Someplace, sometime, someone must complete every IRP. You might want to complete an IRP in your dispatch routine in cases like these:

If the request is erroneous in some easily determined way (such as a request to rewind a printer or to eject the keyboard), the dispatch routine should cause the request to fail by completing it with an appropriate status code.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 137

If the request calls for information that the dispatch function can easily determine (such as a control request asking for the driver's version number), the dispatch routine should provide the answer and complete the request with a successful status code.

Mechanically, completing an IRP entails filling in the Status and *Information* members within the IRP's *IoStatus* block and calling *IoCompleteRequest*. The Status value is one of the codes defined by manifest constants in the DDK header file NTSTATUS.H. Refer to Table 5-3 for an abbreviated list of status codes for common situations. The *Information* value depends on what type of IRP you're completing and on whether you're causing the IRP to succeed or to fail. Most of the time, when you're causing an IRP to fail (that is, completing it with an error status of some kind), you'll set *Information* to 0. When you cause an IRP that involves data transfer to succeed, you ordinarily set the *Information* field equal to the number of bytes transferred.

Status Code	Description
STATUS_SUCCESS	Normal completion.
STATUS_UNSUCCESSFUL	Request failed, but no other status code describes the reason specifically.
STATUS_NOT_IMPLEMENTED	A function hasn't been implemented.
STATUS_INVALID_HANDLE	An invalid handle was supplied for an operation.
STATUS_INVALID_PARAMETER	A parameter is in error.
STATUS_INVALID_DEVICE_REQUEST	The request is invalid for this device.
STATUS_END_OF_FILE	End-of-file marker reached.
STATUS_DELETE_PENDING	The device is in the process of being removed from the system.
STATUS_INSUFFICIENT_RESOURCES	Not enough system resources (often memory) to perform an operation.

Table 5-3. *Some Commonly Used NTSTATUS Codes*

NOTE
Always be sure to consult the DDK documentation for the correct setting of *IoStatus.Information* for the IRP you're dealing with. In some flavors of *IRP_MJ_PNP*, for example, this field is used as a pointer to a data structure that the PnP Manager is responsible for releasing. If you were to overstore the *Information* field with 0 when causing the request to fail, you would unwittingly cause a resource leak.

Because completing a request is something you do so often, I find it useful to have a helper routine to carry out the mechanics:

```
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status,
  ULONG PTR Information)
  {
  Irp->IoStatus.Status = status;
  Irp->IoStatus.Information = Information;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return status;
  }
```

I defined this routine in such a way that it returns whatever status value you supply as its second argument. That's because I'm such a lazy typist: the return value allows me to use this helper whenever I want to complete a request and then immediately return a status code. For example:

```
NTSTATUS DispatchControl(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;
  if (code == IOCTL TOASTER BOGUS)
    return CompleteRequest(Irp, STATUS INVALID DEVICE REQUEST, 0);
  :
  }
```

You might notice that the Information argument to the *CompleteRequest* function is typed as a *ULONG_PTR*. In other words, this value can be either a *ULONG* or a pointer to something (and therefore potentially 64 bits wide).

When you call *IoCompleteRequest*, you supply a priority boost value to be applied to whichever thread is currently waiting for this request to complete. You normally choose a boost value that depends on the type of device, as suggested by the manifest constant names listed in Table 5-4. The priority adjustment improves the throughput of threads that frequently wait for I/O operations to complete. Events for which the end user is directly responsible, such as keyboard or mouse operations, result in greater priority boosts in order to give preference to interactive tasks. Consequently, you want to choose the boost value with at least some care. Don't use *IO_SOUND_INCREMENT* for absolutely every operation a sound card driver finishes, for example—it's not necessary to apply this extraordinary priority increment to a get-driver-version control request.

IPR2024-00292
KASPERSKY EX 1008 Page 138

Manifest Constant	Numeric Priority Boost
IO_NO_INCREMENT	0
IO_CD_ROM_INCREMENT	1
IO_DISK_INCREMENT	1
IO_KEYBOARD_INCREMENT	6
IO_MAILSLOT_INCREMENT	2
IO_MOUSE_INCREMENT	6
IO_NAMED_PIPE_INCREMENT	2
IO_NETWORK_INCREMENT	2
IO_PARALLEL_INCREMENT	1
IO_SERIAL_INCREMENT	2
IO_SOUND_INCREMENT	8
IO_VIDEO_INCREMENT	1

Table 5-4. *Priority Boost Values for IoCompleteRequest*

☑ Don't, by the way, complete an IRP with the special status code *STATUS_PENDING*. Dispatch routines often return *STATUS_PENDING* as their return value, but you should never set *IoStatus.Status* to this value. Just to make sure, the checked build of *IoCompleteRequest* generates an ASSERT failure if it sees *STATUS_PENDING* in the ending status. Another popular value for people to use by mistake is apparently -1, which doesn't have any meaning as an *NTSTATUS* code at all. There's a checked-build ASSERT to catch that mistake too. The Driver Verifier will complain if you try to do either of these bad things.

Before calling *IoCompleteRequest*, be sure to remove any cancel routine that you might have installed for an IRP. As you'll learn later in this chapter, you install a cancel routine while you keep an IRP in a queue. You must remove an IRP from the queue before completing it. All the queuing schemes I'll discuss in this book clear the cancel routine pointer when they dequeue an IRP. Therefore, you probably don't need to have additional code in your driver as in this sample:

```
IoSetCancelRoutine(Irp, NULL);  // <== almost certainly redundant
IoCompleteRequest(Irp, ...);
```

So far, I've just explained how to call *IoCompleteRequest*. That function performs several tasks that you need to understand:

- Calling completion routines that various drivers might have installed. I'll discuss the important topic of I/O completion routines later in this chapter.
- Unlocking any pages belonging to Memory Descriptor List (MDL) structures attached to the IRP. An MDL will be used for the buffer for an *IRP_MJ_READ* or *IRP_MJ_WRITE* for a device whose device object has the *DO_DIRECT_IO* flag set. Control operations also use an MDL if the control code's buffering method specifies one of the *METHOD_XX_DIRECT* methods. I'll discuss these issues more fully in Chapter 7 and Chapter 9, respectively.
- Scheduling a special kernel APC to perform final cleanup on the IRP. This cleanup includes copying input data back to a user buffer, copying the IRP's ending status, and signaling whichever event the originator of the IRP might be waiting on. The fact that completion processing includes an APC, and that the cleanup includes setting an event, imposes some exacting requirements on the way a driver implements a completion routine, so I'll also discuss this aspect of I/O completion in more detail later.

Passing an IRP Down the Stack

The whole goal of the layering of device objects that WDM facilitates is for you to be able to easily pass IRPs from one layer down to the next. Back in Chapter 2, I discussed how your *AddDevice* routine would contribute its portion of the effort required to create a stack of device objects with a statement like this one:

```
pdx->LowerDeviceObject = IoAttachDeviceToDeviceStack(fdo, pdo);
```

where fdo is the address of your own device object and pdo is the address of the physical device object (PDO) at the bottom of the device stack. *IoAttachDeviceToDeviceStack* returns to you the address of the device object immediately underneath yours. When you decide to forward an IRP that you received from above, this is the device object you'll specify in the eventual call to *IoCallDriver*.

☑ Before passing an IRP to another driver, be sure to remove any cancel routine that you might have installed for the IRP. As I mentioned just a few paragraphs ago, you'll probably fulfill this requirement without specifically worrying about it. Your queue management code will zero the cancel routine pointer when it dequeues an IRP. If you never queued the IRP in the first place, the driver above you will have made sure the cancel routine pointer was *NULL*. The Driver Verifier will make sure that you don't break this rule.

When you pass an IRP down, you have the additional responsibility of initializing the *IO_STACK_LOCATION* that the next

IPR2024-00292
KASPERSKY EX 1008 Page 139

driver will use to obtain its parameters. One way of doing this is to perform a physical copy, like this:

```
:
IoCopyCurrentIrpStackLocationToNext(Irp);
status = IoCallDriver(pdx->LowerDeviceObject, Irp);
:
```

IoCopyCurrentIrpStackLocationToNext is a macro in WDM.H that copies all the fields in an *IO_STACK_LOCATION*—except for the ones that pertain to the I/O completion routines—from the current stack location to the next one. In previous versions of Windows NT, kernel-mode driver writers sometimes copied the entire stack location, which would cause the caller's completion routine to be called twice. The *IoCopyCurrentIrpStackLocationToNext* macro, which is new with the WDM, avoids the problem.

If you don't care what happens to an IRP after you pass it down the stack, use the following alternative to *IoCopyCurrentIrpStackLocationToNext*:

```
NTSTATUS ForwardAndForget(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  IoSkipCurrentIrpStackLocation(Irp);
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }
```

IoSkipCurrentIrpStackLocation retards the IRP's stack pointer by one position. *IoCallDriver* will immediately advance the stack pointer. The net effect is to not change the stack pointer. When the next driver's dispatch routine calls *IoGetCurrentIrpStackLocation*, it will retrieve exactly the same *IO_STACK_LOCATION* pointer that we were working with, and it will thereby process exactly the same request (same major and minor function codes) with the same parameters.

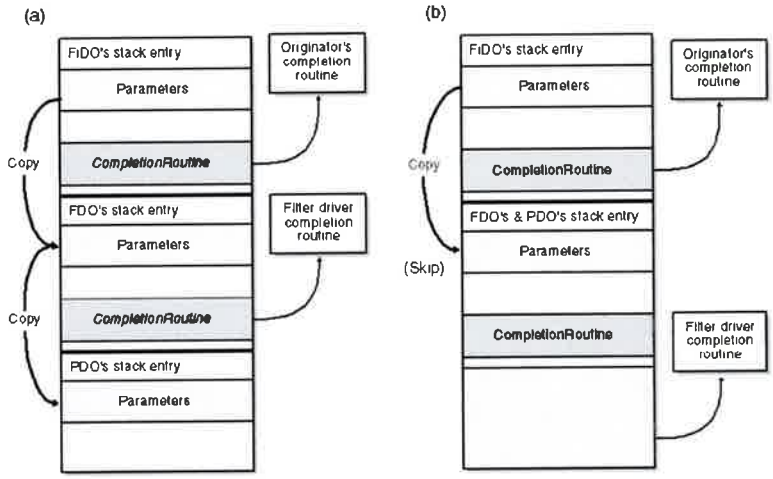


Figure 5-6. *Comparison of copying vs. skipping I/O stack locations.*

CAUTION
The version of *IoSkipCurrentIrpStackLocation* that you get when you use the Windows Me or Windows 2000 build environment in the DDK is a macro that generates two statements without surrounding braces. Therefore, you mustn't use it in a construction like this:

```
if (<expression>)
  IoSkipCurrentIrpStackLocation(Irp);  // <== don't do this!
```

The explanation of why *IoSkipCurrentIrpStackLocation* works is so tricky that I thought an illustration might help. Figure 5-6 illustrates a situation in which three drivers are in a particular stack: yours (the function device object [FDO]) and two others (an upper filter device object [FiDO] and the PDO). In the picture on the left, you see the relationship between stack locations, parameters, and completion routines when we do the copy step with *IoCopyCurrentIrpStackLocationToNext*. In the picture on the right, you see the same relationships when we use the *IoSkipCurrentIrpStackLocation* shortcut. In the right-hand picture, the third and last stack location is fallow, but nobody gets confused by that fact.

IPR2024-00292
KASPERSKY EX 1008 Page 140

Queuing an IRP for Later Processing

The third alternative action for a dispatch routine is to queue the IRP for later processing. The following code snippet assumes you're using one of my *DEVQUEUE* queue objects for IRP queuing. I'll explain the *DEVQUEUE* object later in this chapter.

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  .
  .
1
  IoMarkIrpPending(Irp);
2
  StartPacket(&pdx->dqSomething, fdo, Irp, CancelRoutine);
3
  return STATUS PENDING;
  }
```

1. Whenever we return *STATUS_PENDING* from a dispatch routine (as we're about to do here), we make this call to help the I/O Manager avoid an internal race condition. We must do this before we relinquish ownership of the IRP.
2. If our device is currently busy or stalled because of a PnP or Power event, *StartPacket* puts the request in a queue. Otherwise, *StartPacket* marks the device as busy and calls our *StartIo* routine. I'll describe the *StartIo* routine in the next section. The last argument is the address of a cancel routine. I'll discuss cancel routines later in this chapter.
3. We return *STATUS_PENDING* to tell our caller that we're not done with this IRP yet.

It's important not to touch the IRP once we call *StartPacket*. By the time that function returns, the IRP might have been completed and the memory it occupies released. The pointer we have might, therefore, now be invalid.

5.2.4 The StartIo Routine

IRP-queuing schemes often revolve around calling a *StartIo* function to process IRPs:

```
VOID StartIo(PDEVICE OBJECT device, PIRP Irp)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) device->DeviceExtension;
  .
  .
  }
```

A *StartIo* routine generally receives control at *DISPATCH_LEVEL*, meaning that it must not generate any page faults.

Your job in StartIo is to commence the IRP you've been handed. How you do this depends entirely on your device. Often you will need to access hardware registers that are also used by your interrupt service routine (ISR) and, perhaps, by other routines in your driver. In fact, sometimes the easiest way to commence a new operation is to store some state information in your device extension and then fake an interrupt. Because either of these approaches needs to be carried out under the protection of the same spin lock that protects your ISR, the correct way to proceed is to call *KeSynchronizeExecution*. For example:

```
VOID StartIo(...)
  {
  .
  .
  KeSynchronizeExecution(pdx->InterruptObject, TransferFirst, (PVOID) pdx);
  }

BOOLEAN TransferFirst(PVOID context)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) context;
  <initialize device for new operation>
  return TRUE;
  }
```

The *TransferFirst* routine shown here is an example of the generic class of *SynchCritSection* routines, so called because they're synchronized with the ISR. I'll discuss the *SynchCritSection* concept in more detail in Chapter 7.

In Windows XP and later systems, you can follow this template instead of calling *KeSynchronizeExecution*:

```
VOID StartIo(...)
  {
  KIRQL oldirql = KeAcquireInterruptSpinLock(pdx->InterruptObject);
```


IPR2024-00292
KASPERSKY EX 1008 Page 141

```
    <initialize device for new operation>
    KeReleaseInterruptSpinLock(pdx->InterruptObject, oldirql);
    }
```

Once *StartIo* gets the device busy handling the new request, it returns. You'll see the request next when your device interrupts to signal that it's done with whatever transfer you started.

5.2.5 The Interrupt Service Routine

When your device is finished transferring data, it might signal a hardware interrupt. In Chapter 7, I'll show you how to use *IoConnectInterrupt* to "hook" the interrupt. One of the arguments to *IoConnectInterrupt* is the address of your ISR. When an interrupt occurs, the system calls your ISR. The ISR runs at the device IRQL (DIRQL) of your particular device and under the protection of a spin lock associated specifically with your ISR. The ISR has the following skeleton:

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PDEVICE EXENSION pdx)
  {
  if (<my device didn't interrupt>)
    return FALSE;

  return TRUE;
  }
```

The first argument of your ISR is the address of the interrupt object created by *IoConnectInterrupt*, but you're unlikely to use this argument. The second argument is whatever context value you specified in your original call to *IoConnectInterrupt*; it will probably be the address of your device extension, as shown in this fragment.

I'll discuss the duties of your ISR in detail in Chapter 7 in connection with reading and writing data, the subject to which interrupt handling is most relevant. To carry on with this discussion of the standard model, I need to tell you that one of the likely things for the ISR to do is to schedule a deferred procedure call (DPC). The purpose of the DPC is to let you do things, such as calling *IoCompleteRequest*, that can't be done at the rarified DIRQL at which your ISR runs. So you might have a line of code like this one:

```
IoRequestDpc(pdx->DeviceObject, NULL, pdx);
```

You'll next see the IRP in the DPC routine you registered inside AddDevice with your call to IoInitializeDpcRequest. The traditional name for that routine is DpcForIsr because it's the DPC routine your ISR requests.

5.2.6 Deferred Procedure Call Routine

The DpcForIsr routine requested by your ISR receives control at DISPATCH_LEVEL. Generally, its job is to finish up the processing of the IRP that caused the most recent interrupt. Often that job entails calling IoCompleteRequest to complete this IRP and StartNextPacket to remove the next IRP from your device queue for forwarding to StartIo.

```
VOID DpcForIsr(PKDPC Dpc PDEVICE OBJECT fdo, PIRP junk, PDEVICE EXTENSION pdx)
: {
:
1
   StartNextPacket(&pdx->dqSomething, fdo);
2
   IoCompleteRequest(Irp, boost);
   }
```

StartNextPacket removes the next IRP from your queue and sends it to *StartIo*.

IoCompleteRequest completes the IRP you specify as the first argument. The second argument specifies a priority boost for the thread that has been waiting for this IRP. You'll also fill in the IoStatus block within the IRP before calling *IoCompleteRequest*, as I explained earlier, in the section "Completing an IRP."

I'm not (yet) showing you how to determine *which* IRP has just completed. You might notice that the third argument to the DPC is typed as a pointer to an IRP. This is because, once upon a time, people often specified an IRP address as one of the context parameters to *IoRequestDpc*, and that value showed up here. Trying to communicate an IRP pointer from the function that queues a DPC is unwise, though, because it's possible for there to be just one call to the DPC routine for any number of requests to queue that DPC. Accordingly, the DPC routine should develop the current IRP pointer based on whatever scheme you happen to be using for IRP queuing.

The call to *IoCompleteRequest* is the end of this standard way of handling an I/O request. After that call, the I/O Manager (or whichever entity created the IRP in the first place) owns the IRP once more. That entity will destroy the IRP and might unblock a thread that has been waiting for the request to complete.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 142

5.3 Completion Routines

You often need to know the results of I/O requests that you pass down to lower levels of the driver hierarchy or that you originate. To find out what happened to a request, you install *a completion routine* by calling *IoSetCompletionRoutine*:

```
IoSetCompletionRoutine(Irp, CompletionRoutine, context,
  InvokeOnSuccess, InvokeOnError, InvokeOnCancel);
```

Irp is the request whose completion you want to know about. *CompletionRoutine* is the address of the completion routine you want called, and *context* is an arbitrary pointer-size value you want passed as an argument to the completion routine. The *InvokeOnXxx* arguments are Boolean values indicating whether you want the completion routine called in three different circumstances:

- *InvokeOnSuccess* means you want the completion routine called when somebody completes the IRP with a status code that passes the *NT_SUCCESS* test.
- *InvokeOnError* means you want the completion routine called when somebody completes the IRP with a status code that does not pass the *NT_SUCCESS* test.
- *InvokeOnCancel* means you want the completion routine called when somebody calls *IoCancelIrp* before completing the IRP. I worded this quite delicately: *IoCancelIrp* will set the Cancel flag in the IRP, and that's the condition that gets tested if you specify this argument. A cancelled IRP might end up being completed with *STATUS_CANCELLED* (which would cause the *NT_SUCCESS* test to fail) or with any other status at all. If the IRP gets completed with an error and you specified *InvokeOnError*, *InvokeOnError* by itself will cause your completion routine to be called. Conversely, if the IRP gets completed without error and you specified *InvokeOnSuccess*, *InvokeOnSuccess* by itself will cause your completion routine to be called. In these cases, *InvokeOnCancel* will be redundant. But if you left out one or the other (or both) of *InvokeOnSuccess* or *InvokeOnError*, the *InvokeOnCancel* flag will let you see the eventual completion of an IRP whose Cancel flag has been set, no matter which status is used for the completion.

At least one of these three flags must be TRUE. Note that *IoSetCompletionRoutine* is a macro, so you want to avoid arguments that generate side effects. The three flag arguments and the function pointer, in particular, are each referenced twice by the macro.

IoSetCompletionRoutine installs the completion routine address and context argument in the next *IO_STACK_LOCATION*—that is, in the stack location in which the next lower driver will find its parameters. Consequently, the lowest-level driver in a particular stack of drivers doesn't dare attempt to install a completion routine. Doing so would be pretty futile, of course, because—by definition of lowest-level driver—there's no driver left to pass the request on to.

CAUTION
Recall that you are responsible for initializing the next I/O stack location before you call *IoCallDriver*. Do this initialization before you install a completion routine. This step is especially important if you use *IoCopyCurrentIrpStackLocationToNext* to initialize the next stack location because that function clears some flags that *IoSetCompletionRoutine* sets.

A completion routine looks like this:

```
NTSTATUS CompletionRoutine(PDEVICE_OBJECT fdo, PIRP Irp, PVOID context)
{
  :
  return <some status code>;
  }
```

It receives pointers to the device object and the IRP, and it also receives whichever context value you specified in the call to *IoSetCompletionRoutine*. Completion routines can be called at *DISPATCH_LEVEL* in an arbitrary thread context but can also be called at *PASSIVE_LEVEL* or *APC_LEVEL*. To accommodate the worst case (*DISPATCH_LEVEL*), completion routines therefore need to be in nonpaged memory and must call only service functions that are callable at or below *DISPATCH_LEVEL*. To accommodate the possibility of being called at a lower IRQL, however, a completion routine shouldn't call functions such as *KeAcquireSpinLockAtDpcLevel* that assume they're at *DISPATCH_LEVEL* to start with.

There are really just two possible return values from a completion routine:

- *STATUS_MORE_PROCESSING_REQUIRED*, which aborts the completion process immediately. The spelling of this status code obscures its actual purpose, which is to short-circuit the completion of an IRP. Sometimes, a driver actually does some additional processing on the same IRP. Other times, the flag just means, "Yo, *IoCompleteRequest*! Like, don't touch this IRP no more, dude!" Future versions of the DDK will therefore define an enumeration constant, *StopCompletion*, that is numerically the same as *STATUS_MORE_PROCESSING_REQUIRED* but more evocatively named. (Future printings of this book may also employ better grammar in describing the meaning to be ascribed the constant, at least if my editors get their way.)

IPR2024-00292
KASPERSKY EX 1008 Page 143

- Anything else, which allows the completion process to continue. Because any value besides *STATUS_MORE_PROCESSING_REQUIRED* has the same meaning as any other, I usually just code *STATUS_SUCCESS*. Future versions of the DDK will define *STATUS_CONTINUE_COMPLETION* and an enumeration constant, *ContinueCompletion*, that are numerically the same as *STATUS_SUCCESS*.

I'll have more to say about these return codes a bit further on in this chapter.

NOTE
The device object pointer argument to a completion routine is the value left in the I/O stack location's *DeviceObject* pointer. *IoCallDriver* ordinarily sets this value. People sometimes create an IRP with an extra stack location so that they can pass parameters to a completion routine without creating an extra context structure. Such a completion routine gets a NULL device object pointer unless the creator sets the *DeviceObject* field.

How Completion Routines Get Called

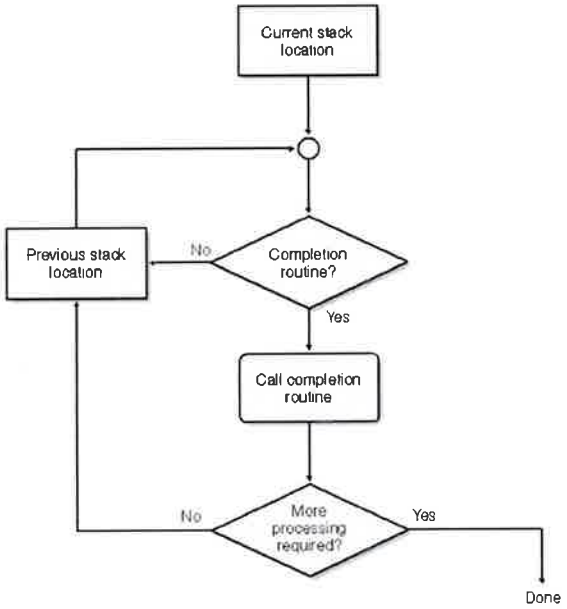


Figure 5-7. *Logic of IoCompleteRequest.*

IoCompleteRequest is responsible for calling all of the completion routines that drivers installed in their respective stack locations. The way the process works, as shown in the flowchart in Figure 5-7, is this: Somebody calls *IoCompleteRequest* to signal the end of processing for the IRP. *IoCompleteRequest* then consults the current stack location to see whether the driver above the current level installed a completion routine. If not, it moves the stack pointer up one level and repeats the test. This process repeats until a stack location is found that does specify a completion routine or until *IoCompleteRequest* reaches the top of the stack. Then *IoCompleteRequest* takes steps that eventually result in somebody releasing the memory occupied by the IRP (among other things).

When *IoCompleteRequest* finds a stack frame with a completion routine pointer, it calls that routine and examines the return code. If the return code is anything other than *STATUS_MORE_PROCESSING_REQUIRED*, *IoCompleteRequest* moves the stack pointer up one level and continues as before. If the return code is *STATUS_MORE_PROCESSING_REQUIRED*, however, *IoCompleteRequest* stops dead in its tracks and returns to its caller. The IRP will then be in a sort of limbo state. The driver whose completion routine halted the stack unwinding process is expected to do more work with the IRP and call *IoCompleteRequest* to resume the completion process.

Within a completion routine, a call to *IoGetCurrentIrpStackLocation* will retrieve the same stack pointer that was current when somebody called *IoSetCompletionRoutine*. You shouldn't rely in a completion routine on the contents of any lower stack location. To reinforce this rule, *IoCompleteRequest* zeroes most of the next location just before calling a completion routine.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 144

Actual Question from Who Wants to Be a Gazillionaire Driver Tycoon:
Suppose you install a completion routine and then immediately call *IoCompleteRequest*. What do you suppose happens?

A. Your computer implodes, creating a gravitational singularity into which the universe instantaneously collapses.

B. You receive the blue screen of death because you're supposed to know better than to install a completion routine in this situation.

C. *IoCompleteRequest* calls your completion routine. Unless the completion routine returns *STATUS_MORE_PROCESSING_REQUIRED*, *IoCompleteRequest* then completes the IRP normally.

D. *IoCompleteRequest* doesn't call your completion routine. It completes the IRP normally.

A. Answer A may be correct, but, if so, my interest in the question will be greatly attenuated. Answer B is incorrect unless there are no more stack locations for this IRP. Answer C is plausible but wrong. Answer D is correct: the pointer to your completion routine is in the *next* stack location. *IoCompleteRequest* starts its processing in the *current* location and never sees the pointer to your completion routine.

The Problem of *IoMarkIrpPending*

Completion routines have one more detail to attend to. You can learn this the easy way or the hard way, as they say in the movies. First the easy way—just follow this rule:

Execute the following code in any completion routine that does not return STATUS_MORE_PROCESSING_REQUIRED:

```
if (Irp->PendingReturned) IoMarkIrpPending(Irp);
```



Now we'll explore the hard way to learn about *IoMarkIrpPending*. Some I/O Manager routines manage an IRP with code that functions much as does this example:

```
KEVENT event;
IO STATUS BLOCK iosb;
KeInitializeEvent(&event, ...);
PIRP Irp = IoBuildDeviceIoControlRequest(..., &event, &iosb);
NTSTATUS status = IoCallDriver(SomeDeviceObject, Irp);
if (status == STATUS PENDING)
  {
  KeWaitForSingleObject(&event, ...);
  status = iosb.Status;
  }
else
  <cleanup IRP>
```

The key here is that, if the returned status is *STATUS_PENDING*, the entity that creates this IRP will wait on the event that was specified in the call to *IoBuildDeviceIoControlRequest*. This discussion could also be about an IRP built by *IoBuildSynchronousFsdRequest* too—the important factor is the conditional wait on the event.

So who, you might well wonder, signals that event? *IoCompleteRequest* does this signaling indirectly by scheduling an APC to the same routine that performs the *<cleanup IRP>* step in the preceding pseudocode. That cleanup code will do many tasks, including calling *IoFreeIrp* to release the IRP and *KeSetEvent* to set the event on which the creator might be waiting. For some types of IRP, *IoCompleteRequest* will always schedule the APC. For other types of IRP, though, *IoCompleteRequest* will schedule the APC only if the *SL_PENDING_RETURNED* flag is set in the topmost stack location. You don't need to know which types of IRP fall into these two categories because Microsoft might change the way this function works and invalidate the deductions you might make if you knew. You do need to know, though, that *IoMarkPending* is a macro whose only purpose is to set *SL_PENDING_RETURNED* in the current stack location. Thus, if the dispatch routine in the topmost driver on the stack does this:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
. IoMarkIrpPending(Irp);
:
  return STATUS_PENDING;
```


IPR2024-00292
KASPERSKY EX 1008 Page 145

```
}
```

things will work out nicely. (I'm violating my naming convention here to emphasize where this dispatch function lives.) Because this dispatch routine returns *STATUS_PENDING*, the originator of the IRP will call *KeWaitForSingleObject*. Because the dispatch routine sets the *SL_PENDING_RETURNED* flag, *IoCompleteRequest* will know to set the event on which the originator waits.

But suppose the topmost driver merely passed the request down the stack, and the second driver pended the IRP:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  IoCopyCurrentIrpStackLocationToNext(Irp);
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }

NTSTATUS SecondDriverDispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
. IoMarkIrpPending(Irp);
:
  return STATUS PENDING;
  }
```

Apparently, the second driver's stack location contains the *SL_PENDING_RETURNED* flag, but the first driver's does not. *IoCompleteRequest* anticipates this situation, however, by propagating the *SL_PENDING_RETURNED* flag whenever it unwinds a stack location that doesn't have a completion routine associated with it. Because the top driver didn't install a completion routine, therefore, *IoCompleteRequest* will have set the flag in the topmost location, and it will have caused the completion event to be signaled.

In another scenario, the topmost driver uses *IoSkipCurrentIrpStackLocation* instead of *IoCopyCurrentIrpStackLocationToNext*. Here, everything works out by default. This is because the *IoMarkIrpPending* call in *SecondDriverDispatchSomething* sets the flag in the topmost stack location to begin with.

Things get sticky if the topmost driver installs a completion routine:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, TopDriverCompletionRoutine, ...);
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }

NTSTATUS SecondDriverDispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
. IoMarkIrpPending(Irp);
:
  return STATUS PENDING;
  }
```

Here *IoCompleteRequest* won't propagate *SL_PENDING_RETURNED* into the topmost stack location. I'm not exactly sure why the Windows NT designers decided not to do this propagation, but it's a fact that they did so decide. Instead, just before calling the completion routine, *IoCompleteRequest* sets the *PendingReturned* flag in the IRP to whichever value *SL_PENDING_RETURNED* had in the immediately lower stack location. The completion routine must then take over the job of setting *SL_PENDING_RETURNED* in its own location:

```
NTSTATUS TopDriverCompletionRoutine(PDEVICE OBJECT fido,  PIRP Irp, ...)
  {
  if (Irp->PendingReturned)
  . IoMarkIrpPending(Irp);
  :
  return STATUS SUCCESS;
  }
```

If you omit this step, you'll find that threads deadlock waiting for someone to signal an event that's destined never to be signaled. So don't omit this step.

Given the importance of the call to *IoMarkIrpPending*, driver programmers through the ages have tried to find other ways of dealing with the problem. Here is a smattering of bad ideas.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 146

Bad Idea # 1—Conditionally Call *IoMarkIrpPending* in the Dispatch Routine

The first bad idea is to try to deal with the pending flag solely in the dispatch routine, thereby keeping the completion routine pristine and understandable in some vague way:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, TopDriverCompletionRoutine, ...);
  NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  if (status == STATUS PENDING)
    IoMarkIrpPending(Irp); // <== Argh! Don't do this!
  return status;
  }
```

The reason this is a bad idea is that the IRP might already be complete, and someone might already have called *IoFreeIrp*, by the time *IoCallDriver* returns. You must treat the pointer as poison as soon as you give it away to a function that might complete the IRP.

Bad idea # 2—Always Call *IoMarkIrpPending* in the Dispatch Routine

Here the dispatch routine unconditionally calls *IoMarkIrpPending* and then returns whichever value *IoCallDriver* returns:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  IoMarkIrpPending(Irp); // <== Don't do this either!
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, TopDriverCompletionRoutine, ...);
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }
```

This is a bad idea if the next driver happens to complete the IRP in its dispatch routine and returns a nonpending status. In this situation, *IoCompleteRequest* will cause all the completion cleanup to happen. When you return a nonpending status, the I/O Manager routine that originated the IRP might call the same completion cleanup routine a second time. This leads to a double-completion bug check.

Remember always to pair the call to *IoMarkIrpPending* with returning *STATUS_PENDING*. That is, do both or neither, but never one without the other.

Bad Idea # 3—Call *IoMarkPending* Regardless of the Return Code from the Completion Routine

In this example, the programmer forgot the qualification of the rule about when to make the call to *IoMarkIrpPending* from a completion routine:

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  KEVENT event;
  KeInitializeEvent(&event, NotificationEvent, FALSE);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, TopDriverCompletionRoutine, &event,
    TRUE, TRUE, TRUE);
  IoCallDriver(pdx->LowerDeviceObject, Irp);
  KeWaitForSingleObject(&event, ...);

  Irp->IoStatus.Status = status;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return status;
  }

NTSTATUS TopDriverCompletionRoutine(PDEVICE OBJECT fido, PIRP Irp, PVOID pev)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp); // <== oops
  KeSetEvent((PKEVENT) pev, IO_NO_INCREMENT, FALSE);
```


IPR2024-00292
KASPERSKY EX 1008 Page 147

```
  return STATUS MORE PROCESSING REQUIRED;
  }
```

What's probably going on here is that the programmer wants to forward the IRP synchronously and then resume processing the IRP after the lower driver finishes with it. (See IRP-handling scenario 7 at the end of this chapter.) That's how you're supposed to handle certain PnP IRPs, in fact. This example can cause a double-completion bug check, though, if the lower driver happens to return *STATUS_PENDING*. This is actually the same scenario as in the previous bad idea: your dispatch routine is returning a nonpending status, but your stack frame has the pending flag set. People often get away with this bad idea, which existed in the *IRP_MJ_PNP* handlers of many early Windows 2000 DDK samples, because no one ever posts a Plug and Play IRP. (Therefore, *PendingReturned* is never set, and the incorrect call to *IoMarkIrpPending* never happens.)

A variation on this idea occurs when you create an asynchronous IRP of some kind. You're supposed to provide a completion routine to free the IRP, and you'll necessarily return *STATUS_MORE_PROCESSING_REQUIRED* from that completion routine to prevent *IoCompleteRequest* from attempting to do any more work on an IRP that has disappeared:

```
SOMETYPE SomeFunction()
  {
  PIRP Irp = IoBuildAsynchronousFsdRequest(...);
  IoSetCompletionRoutine(Irp, MyCompletionRoutine, ...);
  IoCallDriver(...);
  }

NTSTATUS MyCompletionRoutine(PDEVICE OBJECT junk, PIRP Irp,
  PVOID context)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp); // <== oops!
  IoFreeIrp(Irp);
  return STATUS MORE PROCESSING REQUIRED;
  }
```

The problem here is that there is no current stack location inside this completion routine! Consequently, *IoMarkIrpPending* modifies a random piece of storage. Besides, it's fundamentally silly to worry about setting a flag that *IoCompleteRequest* will never inspect: you're returning *STATUS_MORE_PROCESSING_REQUIRED*, which is going to cause *IoCompleteRequest* to immediately return to its own caller without doing another single thing with your IRP.

Avoid both of these problems by remembering not to call *IoMarkIrpPending* from a completion routine that returns *STATUS_MORE_PROCESSING_REQUIRED*.

Bad Idea # 4—Always Pend the IRP

Here the programmer gives up trying to understand and just always pends the IRP. This strategy avoids needing to do anything special in the completion routine.

```
NTSTATUS TopDriverDispatchSomething(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  IoMarkIrpPending(Irp);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, TopDriverCompletionRoutine, ...);
  IoCallDriver(pdx->LowerDeviceObject, Irp);
  return STATUS PENDING;
  }

NTSTATUS TopDriverCompletionRoutine(PDEVICE OBJECT fido, PIRP Irp, ...)
  {

  return STATUS SUCCESS;
  }
```

This strategy isn't so much bad as inefficient. If *SL_PENDING_RETURNED* is set in the topmost stack location, *IoCompleteRequest* schedules a special kernel APC to do the work in the context of the originating thread. Generally speaking, if a dispatch routine posts an IRP, the IRP will end up being completed in some other thread. An APC is needed to get back into the original context in order to do some buffer copying. But scheduling an APC is relatively expensive, and it would be nice to avoid the overhead if you're still in the original thread. Thus, if your dispatch routine doesn't actually return *STATUS_PENDING*, you shouldn't mark your stack frame pending.

But nothing really awful will happen if you implement this bad idea, in the sense that the system will keep working normally. Note also that Microsoft might someday change the way completion cleanup happens, so don't write your driver on the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 148

assumption that an APC is always going to occur.

A Plug and Play Complication

The PnP Manager might conceivably decide to unload your driver before one of your completion routines has a chance to return to the I/O Manager. Anyone who sends you an IRP is supposed to prevent this unhappy occurrence by making sure you can't be unloaded until you've finished handling that IRP. When you create an IRP, however, you have to protect yourself. Part of the protection involves a so-called remove lock object, discussed in Chapter 6, which gates PnP removal until drivers under you finish handling all outstanding IRPs. Another part of the protection is the following function, available in XP and later releases of Windows:

```
IoSetCompletionRoutineEx(DeviceObject, Irp, CompletionRoutine,
  context, InvokeOnSuccess, InvokeOnError, InvokeOnCancel);
```

NOTE
The DDK documentation for IoSetCompletionRoutineEx suggests that it's useful only for non-PnP drivers. As discussed here, however, on many occasions a PnP driver might need to use this function to achieve full protection from early unloading.

The *DeviceObject* parameter is a pointer to your own device object. *IoSetCompletionRoutineEx* takes an extra reference to this object just before calling your completion routine, and it releases the reference when your completion routine returns. The extra reference pins the device object and, more important, your driver, in memory. But because this function doesn't exist in Windows versions prior to XP, you need to consider carefully whether you want to go to the trouble of calling *MmGetSystemRoutineAddress* (and loading a Windows 98/Me implementation of the same function) to dynamically link to this routine if it happens to be available. It seems to me that there are five discrete situations to consider:



Situation 1: Synchronous Subsidiary IRP

The first situation to consider occurs when you create a synchronous IRP to help you process an IRP that someone else has sent you. You intend to complete the main IRP after the subsidiary IRP completes.

You wouldn't ordinarily use a completion routine with a synchronous IRP, but you might want to if you were going to implement the safe cancel logic discussed later in this chapter. If you follow that example, your completion routine will safely return before you completely finish handling the subsidiary IRP and, therefore, comfortably before you complete the main IRP. The sender of the main IRP is keeping you in memory until then. Consequently, you won't need to use *IoSetCompletionRoutineEx*.

Situation 2: Asynchronous Subsidiary IRP

In this situation, you use an asynchronous subsidiary IRP to help you implement a main IRP that someone sends you. You complete the main IRP in the completion routine that you're obliged to install for the subsidiary IRP.

Here you should use *IoSetCompletionRoutineEx* if it's available because the main IRP sender's protection expires as soon as you complete the main IRP. Your completion routine still has to return to the I/O Manager and therefore needs the protection offered by this new routine.

Situation 3: IRP Issued from Your Own System Thread

The third situation in our analysis of completion routines occurs when a system thread you've created (see Chapter 14 for a discussion of system threads) installs completion routines for IRPs it sends to other drivers. If you create a truly asynchronous IRP in this situation, use *IoSetCompletionRoutineEx* to install the obligatory completion routine and make sure that your driver can't unload before the completion routine is actually called. You could, for example, claim an *IO_REMOVE_LOCK* that you release in the completion routine. If you use scenario 8 from the cookbook at the end of this chapter to send a nominally asynchronous IRP in a synchronous way, however, or if you use synchronous IRPs in the first place, there's no particular reason to use *IoSetCompletionRoutineEx* because you'll presumably wait for these IRPs to finish before calling *PsTerminateSystemThread* to end the thread. Some other function in your driver will be waiting for the thread to terminate before allowing the operating system to finally unload your driver. This combination of protections makes it safe to use an ordinary completion routine.

Situation 4: IRP Issued from a Work Item

Here I hope you'll be using *IoAllocateWorkItem* and *IoQueueWorkItem*, which protect your driver from being unloaded until the work item callback routine returns. As in the previous situation, you'll want to use *IoSetCompletionRoutineEx* if you issue an asynchronous IRP and don't wait (as in scenario 8) for it to finish. Otherwise, you don't need the new routine unless you somehow return before the IRP completes, which would be against all the rules for IRP handling and not just the rules for completion routines.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 149

Situation 5: Synchronous or Asynchronous IRP for Some Other Purpose

Maybe you have some reason for issuing a synchronous IRP that is not in aid of an IRP that someone else has sent you and is not issued from the context of your own system thread or a work item. I confess that I can't think of a circumstance in which you'd actually want to do this, but I think you'd basically be toast if you tried. Protecting your completion routine, if any, probably helps a bit, but there's no bulletproof way for you to guarantee that you'll still be there when *IoCallDriver* returns. If you think of a way, you'll simply move the problem to after you do whatever it is you think of, at which point there has to be at least a return instruction that will get executed without protection from outside your driver.

So don't do this.

5.4 Queuing I/O Requests

Sometimes your driver receives an IRP that it can't handle right away. Rather than reject the IRP by causing it to fail with an error status, your dispatch routine places the IRP on a queue. In another part of your driver, you provide logic that removes one IRP from the queue and passes it to a *StartIo* routine.

Queuing an IRP is conceptually very simple. You can provide a list anchor in your device extension, which you initialize in your *AddDevice* function:

```
typedef struct  DEVICE EXTENSION {
:
  LIST ENTRY IrpQueue;
  BOOLEAN DeviceBusy;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS AddDevice(...)
: {
:
. InitializeListHead(&pdx->IrpQueue);
:
  }
```

Then you can write two naive routines for queuing and dequeuing IRPs:

```
VOID NaiveStartPacket(PDEVICE EXTENSION pdx, PIRP Irp)
  {
  if (pdx->DeviceBusy)
    InsertTailList(&pdx->IrpQueue, &Irp->Tail.Overlay.ListEntry);
  else
    {
    pdx->DeviceBusy = TRUE;
    StartIo(pdx->DeviceObject, Irp);
    }
  }

VOID NaiveStartNextPacket(PDEVICE EXTENSION pdx, PIRP Irp)
  {
  if (IsListEmpty(&pdx->IrpQueue))
    pdx->DeviceBusy = FALSE;
  else
    {
    PLIST ENTRY foo = RemoveHeadList(&pdx->IrpQueue);
    PIRP Irp = CONTAINING RECORD(foo, IRP,
      Tail.Overlay.ListEntry);
    StartIo(pdx->DeviceObject, Irp);
    }
  }
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 150

Microsoft Queuing Routines|
Apart from this sidebar, I'm omitting discussion of the functions *IoStartPacket* and *IoStartNextPacket*, which have been part of Windows NT since the beginning. These functions implement a queuing model that's inappropriate for WDM drivers. In that model, a device is in one of three states: idle, busy with an empty queue, or busy with a nonempty queue. If you call *IoStartPacket* at a time when the device is idle, it unconditionally sends the IRP to your *StartIo* routine. Unfortunately, many times a WDM driver needs to queue an IRP even though the device is idle. These functions also rely heavily on a global spin lock whose overuse has created a serious performance bottleneck.

Just in case you happen to be working on an old driver that uses these obsolete routines, however, here's how they work. A dispatch routine would queue an IRP like this:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  IoMarkIrpPending(Irp);
  IoStartPacket(fdo, Irp, NULL, CancelRoutine);
  return STATUS_PENDING;
  }
```

Your driver would have a single *StartIo* routine. Your *DriverEntry* routine would set the *DriverStartIo* field of the driver object to point to this routine. If your *StartIo* routine completes IRPs, you would also call *IoSetStart-IoAttributes* (in Windows XP or later) to help prevent excessive recursion into *StartIo*. *IoStartPacket* and *IoStartNextPacket* call *StartIo* to process one IRP at a time. In other words, *StartIo* is the place where the I/O manager serializes access to your hardware.

A DPC routine (see the later discussion of how DPC routines work) would complete the previous IRP and start the next one using this code:

```
VOID DpcForIsr(PKDPC junk, PDEVICE OBJECT fdo, PIRP Irp,  PVOID morejunk)
  {
  IoCompleteRequest(Irp, STATUS NO INCREMENT);
  IoStartNextPacket(fdo, TRUE);
  }
```

To provide for canceling a queued IRP, you would need to write a cancel routine. Illustrating that and the cancel logic in StartIo is beyond the scope of this book.

In addition, you can rely on the *CurrentIrp* field of a *DEVICE_OBJECT* to always contain NULL or the address of the IRP most recently sent (by *IoStartPacket* or *IoStartNextPacket*) to your *StartIo* routine.

Then your dispatch routine calls *NaiveStartPacket*, and your DPC routine calls *NaiveStartNextPacket* in the manner discussed earlier in connection with the standard model.

There are many problems with this scheme, which is why I called it naive. The most basic problem is that your DPC routine and multiple instances of your dispatch routine could all be simultaneously active on different CPUs. They would likely conflict in trying to access the queue and the busy flag. You could address that problem by creating a spin lock and using it to guard against the obvious races, as follows:

```
typedef struct  DEVICE EXTENSION {
  LIST ENTRY IrpQueue;
  KSPIN LOCK IrpQueueLock;
  BOOLEAN DeviceBusy;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS AddDevice(...)
  {
  InitializeListHead(&pdx->IrpQueue);
  KeInitializeSpinLock(&pdx->IrpQueueLock);

  }

VOID LessNaiveStartPacket(PDEVICE EXTENSION pdx, PIRP Irp)
  {
  KIRQL oldirql;
  KeAcquireSpinLock(&pdx->IrpQueueLock, &oldirql);
  if (pdx->DeviceBusy)
    {
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 151

```
    InsertTailList(&pdx->IrpQueue, &Irp->Tail.Overlay.ListEntry);
    KeReleaseSpinLock(&pdx->IrpQueueLock, oldirql);
    }
  else
    {
    pdx->DeviceBusy = TRUE;
    KeReleaseSpinLock(&pdx->IrpQueueLock, DISPATCH_LEVEL);
    StartIo(pdx->DeviceObject, Irp);
    KeLowerIrql(oldirql);
    }
  }

VOID LessNaiveStartNextPacket(PDEVICE_EXTENSION pdx, PIRP Irp)
  {
  KIRQL oldirql;
  KeAcquireSpinLock(&pdx->IrpQueueLock, &oldirql);
  if (IsListEmpty(&pdx->IrpQueue)
   {
    pdx->DeviceBusy = FALSE;
    KeReleaseSpinLock(&pdx->IrpQueueLock, oldirql);
  else
    {
    PLIST_ENTRY foo = RemoveHeadList(&pdx->IrpQueue);
    KeReleaseSpinLock(&pdx->IrpQueueLock, DISPATCH_LEVEL);
    PIRP Irp = CONTAINING_RECORD(foo, IRP, Tail.Overlay.ListEntry);
    StartIo(pdx->DeviceObject, Irp);
    KeLowerIrql(oldirql);
    }
  }
```

Incidentally, we always want to call StartIo at a single IRQL. Because DPC routines are among the callers of *LessNaiveStartNextPacket*, and they run at *DISPATCH_LEVEL*, we pick *DISPATCH_LEVEL*. That means we want to stay at *DISPATCH_LEVEL* when we release the spin lock.

(You did remember that these two queue management routines need to be in nonpaged memory because they run at *DISPATCH_LEVEL*, right?)

These queuing routines are actually almost OK, but they have one more defect and a shortcoming. The shortcoming is that we need a way to stall a queue for the duration of certain PnP and Power states. IRPs accumulate in a stalled queue until someone unstalls the queue, whereupon the queue manager can resume sending IRPs to a StartIo routine. The defect in the "less naive" set of routines is that someone could decide to cancel an IRP at essentially any time. IRP cancellation complicates IRP queuing logic so much that I've devoted the next major section to discussing it. Before we get to that, though, let me explain how to use the queuing routines that I crafted to deal with all the problems.

5.4.1 Using the DEVQUEUE Object

To solve a variety of IRP queuing problems, I created a package of subroutines for managing a queue object that I call a *DEVQUEUE*. I'll show you first the basic usage of a *DEVQUEUE*. Later in this chapter, I'll explain how the major *DEVQUEUE* service routines work. I'll discuss in later chapters how your PnP and power management code interacts with the *DEVQUEUE* object or objects you define.

You define a DEVQUEUE object for each queue of requests you'll manage in the driver. For example, if your device manages reads and writes in a single queue, you define one DEVQUEUE:

```
typedef struct  DEVICE_EXTENSION {
.
.
.  DEVQUEUE dqReadWrite;
.
.
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

On the CD Code for the *DEVQUEUE* is part of GENERIC.SYS. In addition, if you use my WDMWIZ to create a skeleton driver and don't ask for GENERIC.SYS support, your skeleton project will include the files DEVQUEUE.CPP and DEVQUEUE.H, which fully implement exactly the same object. I don't recommend trying to type this code from the book because the code from the companion content will contain even more features than I can describe in the book. I also recommend checking my Web site (www.oneysoft.com) for updates and corrections.

Figure 5-8 illustrates the IRP processing logic for a typical driver using DEVQUEUE objects. Each DEVQUEUE has its own StartIo routine, which you specify when you initialize the object in AddDevice:

IPR2024-00292
KASPERSKY EX 1008 Page 152

```
NTSTATUS AddDevice(...)
  {

  PDEVICE EXTENSION pdx = ...;
  InitializeQueue(&pdx->dqReadWrite, StartIo);

  }
```

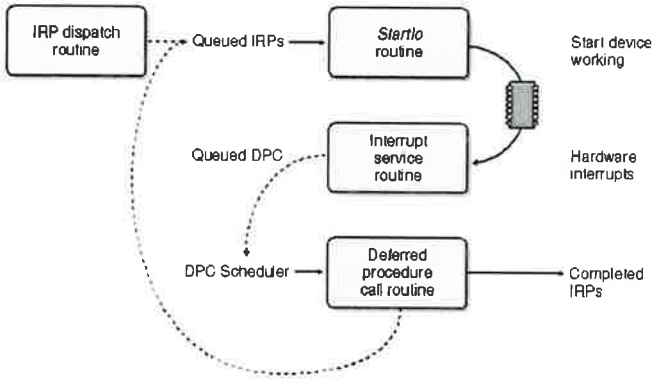


Figure 5-8. *IRP flow with a DEVQUEUE and a StartIo routine.*

You can specify a common dispatch function for both *IRP_MJ_READ* and *IRP_MJ_WRITE*:

```
NTSTATUS DriverEntry(PDRIVER OBJECT DriverObject, PUNICODE STRING RegistryPath)
  {

  DriverObject->MajorFunction[IRP MJ READ] = DispatchReadWrite;
  DriverObject->MajorFunction[IRP MJ WRITE] = DispatchReadWrite;

  }

#pragma PAGEDCODE

NTSTATUS DispatchReadWrite(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PAGED CODE();
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  IoMarkIrpPending(Irp);
  StartPacket(&pdx->dqReadWrite, fdo, Irp, CancelRoutine);
  return STATUS PENDING;
  }

#pragma LOCKEDCODE

VOID CancelRoutine(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  CancelRequest(&pdx->dqReadWrite, Irp);
  }
```

Note that the cancel argument to *StartPacket* is not optional: you must supply a cancel routine, but you can see how simple that routine will be.

If you complete IRPs in a DPC routine, you'll also call *StartNextPacket*:

```
VOID DpcForIsr(PKPDC junk1, PDEVICE OBJECT fdo, PIRP junk2,
  PDEVICE EXTENSION pdx)
  {

  StartNextPacket(&pdx->dqReadWrite, fdo);
  }
```

If you complete IRPs in your *StartIo* routine, schedule a DPC to make the call to *StartNextPacket* in order to avoid excessive

IPR2024-00292
KASPERSKY EX 1008 Page 153

recursion. For example:

```
typedef struct  DEVICE EXTENSION {
  KDPC StartNextDpc;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS AddDevice(...)
  {
  KeInitializeDpc(&pdx->StartNextDpc,
    (PKDEFERRED ROUTINE) StartNextDpcRoutine, pdx);
  }

VOID StartIo(...)
  {
  IoCompleteRequest(...);
  KeInsertQueueDpc(&pdx->StartNextDpc, NULL, NULL);
  }

VOID StartNextDpcRoutine(PKDPC junk1, PDEVICE EXTENSION pdx,
  PVOID junk2, PVOID junk3)
  {
  StartNextPacket(&pdx->dqReadWrite, pdx->DeviceObject);
  }
```

In this example, *StartIo* calls *IoCompleteRequest* to complete the IRP it has just handled. Calling *StartNextPacket* directly might lead to a recursive call to *StartIo*. After enough recursive calls, we'll run out of stack. To avoid the potential stack overflow, we queue the *StartNextDpc* DPC object and return. Because *StartIo* runs at *DISPATCH_LEVEL*, it won't be possible for the DPC routine to be called before *StartIo* returns. Therefore, *StartNextDpcRoutine* can call *StartNextPacket* without worrying about recursion.

NOTE
If you were using the Microsoft queue routines *IoStartPacket* and *IoStartNextPacket*, you'd have a single *StartIo* routine. Your *DriverEntry* routine would set the *DriverStartIo* pointer in the driver object to the address of this routine. To avoid the recursion problem discussed in the text in Windows XP or later, you could call *IoSetStartIoAttributes*.

5.4.2 Using Cancel-Safe Queues

Some drivers work better if they operate with a separate I/O thread. Such a thread wakes up each time there is an IRP to be processed, processes IRPs until a queue is empty, and then goes back to sleep. I'll discuss the details of how such a thread routine works in Chapter 14, but this is the appropriate time to talk about how you can queue IRPs in such a driver. See Figure 5-9.

A *DEVQUEUE* isn't appropriate for this situation because the *DEVQUEUE* wants to call a *StartIo* routine to process IRPs. When you have a separate I/O thread, you want to be responsible in that thread for fetching IRPs. Microsoft provides a set of routines for cancel-safe queue operations that provide most of the functionality you need. These routines don't work automatically with your PnP and Power logic, but I predict it won't be hard to add such support. The Cancel sample in the DDK shows how to work with a cancel-safe queue in exactly this situation, but I'll go over the mechanics here as well.

NOTE
In their original incarnation, the cancel-safe queue functions weren't appropriate when you wanted to use a StartIo routine for actual I/O because they didn't provide a way to set a CurrentIrp pointer and do a queue operation inside one invocation of the queue lock. They were modified while I was writing this book to support StartIo usage, but we didn't have time to include an explanation of how to use the new features. I commend you, therefore, to the DDK documentation.

Note also that the cancel-safe queue functions were first described in an XP release of the DDK. They are implemented in a static library, however, and are therefore available for use on all prior platforms.

IPR2024-00292
KASPERSKY EX 1008 Page 154

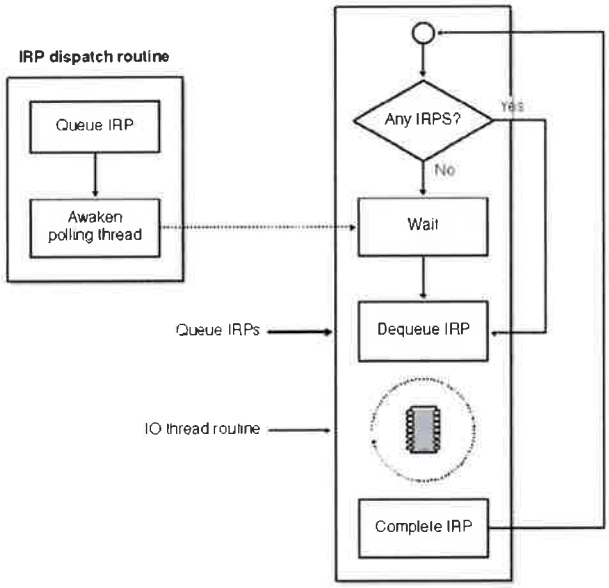


Figure 5-9. *IRP flow with an I/O thread.*

Initialization for Cancel-Safe Queue

To take advantage of the cancel-safe queue functions, first declare six helper functions (see Table 5-5) that the I/O Manager can call to perform operations on your queue. Declare an instance of the IO_CSQ structure in your device extension structure. Also declare an anchor for your IRP queue and whichever synchronization object you want to use. You initialize these objects in your AddDevice function. For example:

```
typedef struct  DEVICE EXTENSION {
  :
  IO CSQ IrpQueue;
  LIST ENTRY IrpQueueAnchor;
  KSPIN LOCK IrpQueueLock;
  :
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  :
  {
  :
  KeInitializeSpinLock(&pdx->IrpQueueLock);
  InitializeListHead(&pdx->IrpQueueAnchor);
  IoCsqInitialize(&pdx->IrpQueue, InsertIrp, RemoveIrp,
  :   PeekNextIrp, AcquireLock, ReleaseLock, CompleteCanceledIrp);
  :
  }
```

Callback Routine	Purpose
AcquireLock	Acquire lock on the queue
CompleteCanceledIrp	Complete an IRP that has been cancelled
InsertIrp	Insert IRP into queue
PeekNextIrp	Retrieve pointer to next IRP in queue without removing it
ReleaseLock	Release queue lock
RemoveIrp	Remove IRP from queue

Table 5-5. *Cancel-Safe Queue Callback Routines*

Using the Queue

You queue an IRP in a dispatch routine like this:

IPR2024-00292
KASPERSKY EX 1008 Page 155

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  IoCsqInsertIrp(&pdx->IrpQueue, Irp, NULL);
  return STATUS PENDING;
  }
```

It's unnecessary and incorrect to call *IoMarkIrpPending* yourself because *IoCsqInsertIrp* does so automatically. As is true with other queuing schemes, the IRP might be complete by the time *IoCsqInsertIrp* returns, so don't touch the pointer afterwards.

To remove an IRP from the queue (say, in your I/O thread), use this code:

```
PIRP Irp = IoCsqRemoveNextIrp(&pdx->IrpQueue, PeekContext);
```

I'll describe the *PeekContext* argument a bit further on. Note that the return value is NULL if no IRPs on are on the queue. The IRP you get back hasn't been cancelled, and any future call to *IoCancelIrp* is guaranteed to do nothing more than set the Cancel flag in the IRP.

You'll also want to provide a dispatch routine for *IRP_MJ_CLEANUP* that will interact with the queue. I'll show you code for that purpose a bit later in this chapter.

Cancel-Safe Queue Callback Routines

The I/O Manager calls your cancel-safe queue callback routines with the address of the queue object as one of the arguments. To recover the address of your device extension structure, use the *CONTAINING_RECORD* macro:

```
#define GET_DEVICE_EXTENSION(csq) \
  CONTAINING_RECORD(csq, DEVICE_EXTENSION, IrpQueue)
```

You supply callback routines for acquiring and releasing the lock you've decided to use for your queue. For example, if you had settled on using a spin lock, you'd write these two routines:

```
VOID AcquireLock(PIO CSQ csq, PKIRQL Irql)
  {
  PDEVICE EXTENSION pdx = GET DEVICE EXTENSION(csq);
  KeAcquireSpinLock(&pdx->IrpQueueLock, Irql);
  }

VOID ReleaseLock(PIO CSQ csq, KIRQL Irql)
  {
  PDEVICE EXTENSION pdx = GET DEVICE EXTENSION(csq);
  KeReleaseSpinLock(&pdx->IrpQueueLock, Irql);
  }
```

You don't have to use a spin lock for synchronization, though. You can use a mutex, a fast mutex, or any other object that suits your fancy.

When you call *IoCsqInsertIrp*, the I/O Manager locks your queue by calling your *AcquireLock* routine and then calls your *InsertIrp* routine:

```
VOID InsertIrp(PIO CSQ csq, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = GET DEVICE EXTENSION(csq);
  InsertTailList(&pdx->IrpQueueAnchor, &Irp->Tail.Overlay.ListEntry);
  }
```

When you call IoCsqRemoveNextIrp, the I/O Manager locks your queue and calls your PeekNextIrp and RemoveIrp functions:

```
PIRP PeekNextIrp(PIO CSQ csq, PIRP Irp, PVOID PeekContext)
  {
  PDEVICE EXTENSION pdx = GET DEVICE EXTENSION(csq);
  PLIST ENTRY next = Irp
    ? Irp->Tail.Overlay.ListEntry.Flink
    : pdx->IrpQueueAnchor.Flink;
  while (next != &pdx->IrpQueueAnchor)
    {
    PIRP NextIrp = CONTAINING_RECORD(next, IRP,
```


IPR2024-00292
KASPERSKY EX 1008 Page 156

```
      Tail.Overlay.ListEntry);
    if (PeekContext && <NextIrp matches PeekContext>)
      return NextIrp;
    if (!PeekContext)
      return NextIrp;
    next = next->Flink;
    }
  return NULL;
  }

VOID RemoveIrp(PIO CSQ csq, PIRP Irp)
  {
  RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
  }
```

The parameters to *PeekNextIrp* require a bit of explanation. Irp, if not *NULL*, is the predecessor of the first IRP you should look at. If Irp is NULL, you should look at the IRP at the front of the list. *PeekContext* is an arbitrary parameter that you can use for any purpose you want as a way for the caller of *IoCsqRemoveNextIrp* to communicate with *PeekNextIrp*. A common convention is to use this argument to point to a *FILE_OBJECT* that's the current subject of an *IRP_MJ_CLEANUP*. I wrote this function so that a NULL value for *PeekContext* means, "Return the next IRP, period." A *non-NULL* value means, "Return the next value that matches *PeekContext*." You define what it means to "match" the peek context.

The sixth and last callback function is this one, which the I/O Manager calls when an IRP needs to be cancelled:

```
VOID CompleteCanceledIrp(PIO CSQ csq, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = GET DEVICE EXTENSION(csq);
  Irp->IoStatus.Status = STATUS CANCELLED;
  Irp->IoStatus.Information = 0;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  }
```

That is, all you do is complete the IRP with *STATUS_CANCELLED*.

To reiterate, the advantage you gain by using the cancel-safe queue functions is that you don't need to write a cancel routine, and you don't need to include any code in your driver (apart from the *CompleteCanceledIrp* function, that is) that relates to cancelling queued IRPs. The I/O Manager installs its own cancel routine, and it promises never to deliver a cancelled IRP back from *IoCsqRemoveNextIrp*.

Parking an IRP on a Cancel-Safe Queue

The preceding sections described how you can use a cancel-safe queue to serialize I/O processing in a kernel thread. Another way to use the cancel-safe queue functions is for parking IRPs while you process them. The idea is that you would place the IRP into the queue when you first received it. Then, when it comes time to complete the IRP, you remove that specific IRP from the queue. You're not using the queue as a real queue in this scenario, because you don't pay any attention to the order of the IRPs in the queue.

To park an IRP, define a persistent context structure for use by the cancel-safe queue package. You need one such structure for each separate IRP that you plan to park. Suppose, for example, that your driver processes "red" requests and "blue" requests (fanciful names to avoid the baggage that real examples sometimes bring along with them).

```
typedef struct  DEVICE EXTENSION {

  IO CSQ IRP CONTEXT RedContext;
  IO CSQ IRP CONTEXT BlueContext;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

When you receive a "red" IRP, you specify the context structure in your call to *IoCsqInsertIrp*:

```
IoCsqInsertIrp(&pdx->IrpQueue, RedIrp, &pdx->RedContext);
```

How to park a "blue" IRP should be pretty obvious.

When you later decide you want to complete a parked IRP, you write code like this:

```
PIRP RedIrp = IoCsqRemoveIrp(&pdx->IrpQueue, &pdx->RedContext);
if (RedIrp)
  {
```


IPR2024-00292
KASPERSKY EX 1008 Page 157

```
    RedIrp->IoStatus.Status = STATUS XXX;
    RedIrp->IoStatus.Information = YYY;
    IoCompleteRequest(RedIrp, IO NO INCREMENT);
    }
```

IoCsqRemoveIrp will return NULL if the IRP associated with the context structure has already been cancelled.

Bear in mind the following caveats when using this mechanism:

- It's up to you to make sure that you haven't previously parked an IRP using a particular context structure. *IoCsqInsertIrp* is a VOID function and therefore has no way to tell you when you violate this rule.
- You mustn't touch an I/O buffer associated with a parked IRP because the IRP can be cancelled (and the I/O buffer released!) at any time while it's parked. You should remove the IRP from the queue before trying to use a buffer.

5.5 Cancelling I/O Requests

Just as happens with people in real life, programs sometimes change their mind about the I/O requests they've asked you to perform for them. We're not talking about simple fickleness here. Applications might terminate after issuing requests that will take a long time to complete, leaving requests outstanding. Such an occurrence is especially likely in the WDM world, where the insertion of new hardware might require you to stall requests while the Configuration Manager rebalances resources or where you might be told at any moment to power down your device.

To cancel a request in kernel mode, someone calls *IoCancelIrp*. The operating system automatically calls *IoCancelIrp* for every IRP that belongs to a thread that's terminating with requests still outstanding. A user-mode application can call CancelIo to cancel all outstanding asynchronous operations issued by a given thread on a file handle. *IoCancelIrp* would like to simply complete the IRP it's given with *STATUS_CANCELLED*, but there's a hitch: *IoCancelIrp* doesn't know where you have salted away pointers to the IRP, and it doesn't know for sure whether you're currently processing the IRP. So it relies on a cancel routine you provide to do most of the work of cancelling an IRP.

It turns out that a call to *IoCancelIrp* is more of a suggestion than a demand. It would be nice if every IRP that somebody tried to cancel really got completed with *STATUS_CANCELLED*. But it's OK if a driver wants to go ahead and finish the IRP normally if that can be done relatively quickly. You should provide a way to cancel I/O requests that might spend significant time waiting in a queue between a dispatch routine and a *StartIo* routine. How long is significant is a matter for your own sound judgment; my advice is to err on the side of providing for cancellation because it's not that hard to do and makes your driver fit better into the operating system.

5.5.1 If It Weren't for Multitasking...

An intricate synchronization problem is associated with cancelling IRPs. Before I explain the problem and the solution, I want to describe the way cancellation would work in a world where there was no multitasking and no concern with multiprocessor computers. In that utopia, several pieces of the I/O Manager would fit together with your StartIo routine and with a cancel routine you'd provide, as follows:

- When you queue an IRP, you set the *CancelRoutine* pointer in the IRP to the address of your cancel routine. When you dequeue the IRP, you set *CancelRoutine* to *NULL*.
- *IoCancelIrp* unconditionally sets the Cancel flag in the IRP. Then it checks to see whether the *CancelRoutine* pointer in the IRP is NULL. While the IRP is in your queue, *CancelRoutine* will be *non-NULL*. In this case, *IoCancelIrp* calls your cancel routine. Your cancel routine removes the IRP from the queue where it currently resides and completes the IRP with *STATUS_CANCELLED*.
- Once you dequeue the IRP, *IoCancelIrp* finds the *CancelRoutine* pointer set to *NULL*, so it doesn't call your cancel routine. You process the IRP to completion with reasonable promptness (a concept that calls for engineering judgment), and it doesn't matter to anyone that you didn't actually cancel the IRP.

5.5.2 Synchronizing Cancellation

Unfortunately for us as programmers, we write code for a multiprocessing, multitasking environment in which effects can sometimes appear to precede causes. There are many possible race conditions between the queue insertion, queue removal, and cancel routines in the naive scenario I just described. For example, what would happen if *IoCancelIrp* called your cancel routine to cancel an IRP that happened to be at the head of your queue? If you were simultaneously removing an IRP from the queue on another CPU, you can see that your cancel routine would probably conflict with your queue removal logic. But this is just the simplest of the possible races.

In earlier times, driver programmers dealt with the cancel races by using a global spin lock—the cancel spin lock. Because you shouldn't use this spin lock for synchronization in your own driver, I've explained it briefly in the sidebar. Read the sidebar for its historical perspective, but don't plan to use this lock.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 158

The Global Cancel Spin Lock
The original Microsoft scheme for synchronizing IRP cancellation revolved around a global cancel spin lock. Routines named *IoAcquireCancelSpinLock* and *IoReleaseCancelSpinLock* acquire and release this lock. The Microsoft queuing routines *IoStartPacket* and *IoStartNextPacket* acquire and release the lock to guard their access to the cancel fields in an IRP and to the *CurrentIrp* field of the device object. *IoCancelIrp* acquires the lock before calling your cancel routine but doesn't release the lock. Your cancel routine runs briefly under the protection of the lock and must call *IoReleaseCancelSpinLock* before returning.

In this scheme, your own *StartIo* routine must also acquire and release the cancel spin lock to safely test the Cancel flag in the IRP and to reset the *CancelRoutine* pointer to NULL.

Hardly anyone was able to craft queuing and cancel logic that approached being bulletproof using this original scheme. Even the best algorithms actually have a residual flaw arising from a coincidence in IRP pointer values. In addition, the fact that every driver in the system needed to use a single spin lock two or three times in the normal execution path created a measurable performance problem. Consequently, Microsoft now recommends that drivers either use the cancel-safe queue routines or else copy someone else's proven queue logic. Neither Microsoft nor I would recommend that you try to design your own queue logic with cancellation because getting it right is very hard.

Nowadays, we handle the cancel races in one of two ways. We can implement our own IRP queue (or, more probably, cut and paste someone else's). Or, in certain kinds of drivers, we can use the *IoCsqXxx* family of functions. You don't need to understand how the *IoCsqXxx* functions handle IRP cancellation because Microsoft intends these functions to be a black box. I'll discuss in detail how my own DEVQUEUE handles cancellation, but I first need to tell you a bit more about the internal workings of *IoCancelIrp*.

5.5.3 Some Details of IRP Cancellation

Here is a sketch of *IoCancelIrp*. You need to know this to correctly write IRP-handling code. (This isn't a copy of the Windows XP source code—it's an abridged excerpt.)

```
BOOLEAN IoCancelIrp(PIRP Irp)
  {
1
  IoAcquireCancelSpinLock(&Irp->CancelIrql);
2
  Irp->Cancel = TRUE;
3
  PDRIVER_CANCEL CancelRoutine = IoSetCancelRoutine(Irp, NULL);
  if (CancelRoutine)
    {
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
4
    (*CancelRoutine)(stack->DeviceObject, Irp);
    return TRUE;
    }
  else
    {
5
    IoReleaseCancelSpinLock(Irp->CancelIrql);
    return FALSE;
    }
  }
```

1. *IoCancelIrp* first acquires the global cancel spin lock. As you know if you read the sidebar earlier, lots of old drivers contend for the use of this lock in their normal IRP-handling path. New drivers hold this lock only briefly while handling the cancellation of an IRP.
2. Setting the Cancel flag to TRUE alerts any interested party that *IoCancelIrp* has been called for this IRP.
3. *IoSetCancelRoutine* performs an interlocked exchange to simultaneously retrieve the existing *CancelRoutine* pointer and set the field to NULL in one atomic operation.
4. *IoCancelIrp* calls the cancel routine, if there is one, without first releasing the global cancel spin lock. The cancel routine must release the lock! Note also that the device object argument to the cancel routine comes from the current stack location, where *IoCallDriver* is supposed to have left it.
5. If there is no cancel routine, *IoCancelIrp* itself releases the global cancel spin lock. Good idea, huh?

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 159

5.5.4 How the DEVQUEUE Handles Cancellation

As I promised, I'll now show you how the major *DEVQUEUE* routines work so you can see how they safely cope with IRP cancellation.

DEVQUEUE Internals—Initialization

The DEVQUEUE object has this declaration in my *DEVQUEUE*.H and GENERIC.H header files:

```
typedef struct  DEVQUEUE {
1
  LIST ENTRY head;
2
  KSPIN LOCK lock;
3
  PDRIVER START StartIo;
4
  LONG stallcount;
5
  PIRP CurrentIrp;
6
  KEVENT evStop;
7
  NTSTATUS abortstatus;
  } DEVQUEUE, *PDEVQUEUE;
```

InitializeQueue initializes one of these objects like this:

```
VOID NTAPI InitializeQueue(PDEVQUEUE pdq,
  PDRIVER STARTIO StartIo)
  {
  InitializeListHead(&pdq->head);
  KeInitializeSpinLock(&pdq->lock);
  pdq->StartIo = StartIo;
  pdq->stallcount = 1;
  pdq->CurrentIrp = NULL;
  KeInitializeEvent(&pdq->evStop, NotificationEvent, FALSE);
  pdq->abortstatus = (NTSTATUS) 0;
  }
```

1. We use an ordinary (noninterlocked) doubly-linked list to queue IRPs. We don't need to use an interlocked list because we'll always access it within the protection of our own spin lock.
2. This spin lock guards access to the queue and other fields in the *DEVQUEUE* structure. It also takes the place of the global cancel spin lock for guarding nearly all of the cancellation process, thereby improving system performance.
3. Each queue has its own associated *StartIo* function that we call automatically in the appropriate places.
4. The stall counter indicates how many times somebody has requested that IRP delivery to *StartIo* be stalled. Initializing the counter to 1 means that the *IRP_MN_START_DEVICE* handler must call *RestartRequests* to release an IRP. I'll discuss this issue more fully in Chapter 6.
5. The *CurrentIrp* field records the IRP most recently sent to the *StartIo* routine. Initializing this field to *NULL* indicates that the device is initially idle.
6. We use this event when necessary to block *WaitForCurrentIrp*, one of the *DEVQUEUE* routines involved in handling PnP requests. We'll set the event inside *StartNextPacket*, which should always be called when the current IRP completes.
7. We reject incoming IRPs in two situations. The first situation occurs after we irrevocably commit to removing the device, when we must start causing new IRPs to fail with *STATUS_DELETE_PENDING*. The second situation occurs during a period of low power, when, depending on the type of device we're managing, we might choose to cause new IRPs to fail with the *STATUS_DEVICE_POWERED_OFF* code. The *abortstatus* field records the status code we should use in rejecting IRPs in these situations.

In the steady state after all PnP initialization finishes, each *DEVQUEUE* will have a zero *stallcount* and *abortstatus*.

DEVQUEUE Internals—Queuing and Cancellation

Here is the complete implementation of the three *DEVQUEUE* routines whose usage I just showed you. I cut and pasted the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 160

source code directly from GENERIC.SYS and did some minor formatting for the sake of readability on the printed page. I also removed some power management code from *StartNextPacket* because it would just confuse this presentation.

```
VOID StartPacket(PDEVQUEUE pdq, PDEVICE OBJECT fdo, PIRP Irp,
  PDRIVER CANCEL cancel)
  {
  KIRQL oldirql;
1
  KeAcquireSpinLock(&pdq->lock, &oldirql);
  NTSTATUS abortstatus = pdq->abortstatus;
2
  if (abortstatus)
    {
    KeReleaseSpinLock(&pdq->lock, oldirql);
    Irp->IoStatus.Status = abortstatus;
    IoCompleteRequest(Irp, IO NO INCREMENT);
    }
3
  else if (pdq->CurrentIrp || pdq->stallcount)
    {
4
    IoSetCancelRoutine(Irp, cancel);
5
    if (Irp->Cancel && IoSetCancelRoutine(Irp, NULL))
      {
      KeReleaseSpinLock(&pdq->lock, oldirql);
      Irp->IoStatus.Status = STATUS CANCELLED;
      IoCompleteRequest(Irp, IO NO INCREMENT);
      }
6
    else
      {
      InsertTailList(&pdq->head, &Irp->Tail.Overlay.ListEntry);
      KeReleaseSpinLock(&pdq->lock, oldirql);
      }
    }
7
  else
    {
    pdq->CurrentIrp = Irp;
    KeReleaseSpinLockFromDpcLevel(&pdq->lock);
    (*pdq->StartIo)(fdo, Irp);
    KeLowerIrql(oldirql);
    }
  }

VOID StartNextPacket(PDEVQUEUE pdq, PDEVICE OBJECT fdo)
  {
  KIRQL oldirql;
8
  KeAcquireSpinLock(&pdq->lock, &oldirql);
9
  pdq->CurrentIrp = NULL;
10
  while (!pdq->stallcount && !pdq->abortstatus
    && !IsListEmpty(&pdq->head))
    {
11
    PLIST ENTRY next = RemoveHeadList(&pdq->head);
    PIRP Irp = CONTAINING RECORD(next, IRP, Tail.Overlay.ListEntry);
12
    if (!IoSetCancelRoutine(Irp, NULL))
      {
      InitializeListHead(&Irp->Tail.Overlay.ListEntry);
      continue;
      }
12
    pdq->CurrentIrp = Irp;
    KeReleaseSpinLockFromDpcLevel(&pdq->lock);
```


IPR2024-00292
KASPERSKY EX 1008 Page 161

```
    (*pdq->StartIo)(fdo, Irp);
    KeLowerIrql(oldirql);
    }
  KeReleaseSpinLock(&pdq->lock, oldirql);
  }
13
VOID CancelRequest(PDEVQUEUE pdq, PIRP Irp)
  {
  KIRQL oldirql = Irp->CancelIrql;
14
  IoReleaseCancelSpinLock(DISPATCH LEVEL);
15
  KeAcquireSpinLockAtDpcLevel(&pdq->lock);
16
  RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
  KeReleaseSpinLock(&pdq->lock, oldirql);
17
  Irp->IoStatus.Status = STATUS CANCELLED;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  }
```

Now I'll explain in detail how these functions work together to provide cancel-safe queuing. I'll do this by describing a series of scenarios that involve all of the code paths.

1. The Normal Case for *StartPacket*

The normal case for *StartPacket* occurs in the steady state when an IRP, which we assume has not been cancelled, arrives after all PnP processing has taken place and at a time when the device was fully powered. In this situation, *stallcount* and *abortstatus* will both be 0. The path through *StartPacket* depends on whether the device is busy, as follows:

- We first acquire the spin lock associated with the queue. (See point 1.) Nearly all the *DEVQUEUE* routines acquire this lock (see points 8 and 15), so we can be sure that no other code on any other CPU can do anything to the queue that would invalidate the decisions we're about to make.
- If the device is busy, the *if* statement at point 3 will find *CurrentIrp* not set to *NULL*. The *if* statement at point 5 will also fail (I'll explain later exactly why), so we'll get to point 6 to put the IRP in the queue. Releasing the spin lock is the last thing we do in this code path.
- If the device is idle, the *if* statement at point 3 will find *CurrentIrp* set to *NULL*. I've already assumed that *stallcount* is 0, so we'll get to point 7 in order to process this IRP. Note how we manage to call *StartIo* at *DISPATCH_LEVEL* after releasing the spin lock.

2. The Normal Case for StartNextPacket

The normal case for *StartNextPacket* is similar to that for *StartPacket*. The *stallcount* and *abortstatus* members are 0, and the IRP at the head of the queue hasn't been cancelled. *StartNextPacket* executes these steps:

- Acquires the queue spin lock (point 8). This protects the queue from simultaneous access by other CPUs trying to execute *StartPacket* or *CancelRequest*. No other CPU can be trying to execute *StartNextPacket* because the only caller of *StartNextPacket* is someone who has just finished processing some other IRP. We allow only one IRP to be active at a time, so there should never be more than one such entity.
- If the list is empty, we just release the spin lock and return. If *StartPacket* had been waiting for the lock, it will now find that the device isn't busy and will call *StartIo*.
- If the list isn't empty, the *if* test at point 10 will succeed, and we'll enter a loop looking for the next uncancelled IRP.
- The first step in the loop (point 11) is to remove the next IRP from the list. Note that *RemoveHeadList* returns the address of a LIST_ENTRY built into the IRP. We use *CONTAINING_RECORD* to get the address of the IRP.
- *IoSetCancelRoutine* (point 12) will return the non-NULL address of the cancel routine originally supplied to *StartPacket*. This is because nothing, least of all *IoCancelIrp*, has changed this pointer since *StartPacket* set it. Consequently, we'll get to point 13, where we'll send this IRP to the *StartIo* routine at *DISPATCH_LEVEL*.

3. IRP Cancelled Prior to StartPacket; Device Idle

Suppose *StartPacket* receives an IRP that was cancelled some time ago. At the time *IoCancelIrp* executed, there wouldn't have been a cancel routine for the IRP. (If there had been, it would have belonged to a driver higher up the stack than us. That other driver would have completed the IRP instead of sending it down to us.) All that *IoCancelIrp* would have done, therefore, is to set the *Cancel* flag in the IRP.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 162

If the device is idle, the *if* test at point 3 fails and we once again go directly to point 7, where we send the IRP to *StartIo*. In effect, we're going to ignore the *Cancel* flag. This is fine so long as we process the IRP "relatively quickly," which is an engineering judgment. If we won't process the IRP with reasonable dispatch, *StartIo* and the downstream logic for handling the IRP should have code to detect the *Cancel* flag and to complete the IRP early.

4. IRP Cancelled During *StartPacket*; Device Idle

In this scenario, someone calls *IoCancelIrp* while *StartPacket* is running. Just as in scenario 3, *IoCancelIrp* will set the *Cancel* flag and return. We'll ignore the flag and send the IRP to *StartIo*.

5. IRP Cancelled Prior to *StartPacket*; Device Busy

The initial conditions are the same as in scenario 3 except that now the device is busy and the if test at point 3 succeeds. We'll set the cancel routine (point 4) and then test the *Cancel* flag (point 5). Because the *Cancel* flag is *TRUE*, we'll go on to call *IoSetCancelRoutine* a second time. The function will return the non-NULL address we just installed, whereupon we'll complete the IRP with *STATUS_CANCELLED*.

6. IRP Cancelled During *StartPacket*; Device Busy

This is the first sticky wicket we encounter in the analysis. Assume the same initial conditions as scenario 3, but now the device is busy and someone calls *IoCancelIrp* at about the same time *StartPacket* is running. There are several possible situations now:

- Suppose we test the Cancel flag (point 5) before *IoCancelIrp* manages to set that flag. Since we find the flag set to *FALSE*, we go to point 6 and queue the IRP. What happens next depends on how *IoCancelIrp*, *CancelRequest*, and *StartNextPacket* interact. *StartPacket* is in a not-my-problem field at this point, however, and needn't worry about this IRP any more.
- Suppose we test the Cancel flag (point 5) after *IoCancelIrp* sets the flag. We have already set the cancel pointer (point 4). What happens next depends on whether *IoCancelIrp* or we are first to execute the *IoSetCancelRoutine* call that changes the cancel pointer back to *NULL*. Recall that *IoSetCancelRoutine* is an atomic operation based on an *InterlockedExchangePointer*. If we execute our call first, we get back a non-*NULL* value and complete the IRP. *IoCancelIrp* gets back *NULL* and therefore doesn't call any cancel routine.
- On the other hand, if *IoCancelIrp* executes its *IoSetCancelRoutine* first, we will get back *NULL* from our call. We'll go on to queue the IRP (point 6) and to enter that not-my-problem field I just referred to. *IoCancelIrp* will call our cancel routine, which will block (point 15) until we release the queue spin lock. Our cancel routine will eventually complete the IRP.

7. Normal IRP Cancellation

IRPs don't get cancelled very often, so I'm not sure it's really right to use the word normal in this context. But if there were a normal scenario for IRP cancellation, this would be it: someone calls *IoCancelIrp* to cancel an IRP that's in our queue, but the cancel process runs to conclusion before *StartNextPacket* can possibly try to reach it. The potential race between *StartNextPacket* and *CancelRequest* therefore can't materialize. Events will unfold this way:

- *IoCancelIrp* acquires the global cancel spin lock, sets the Cancel flag, and executes *IoSetCancelRoutine* to simultaneously retrieve the address of our cancel routine and set the cancel pointer in the IRP to *NULL*. (Refer to the earlier sketch of *IoCancelIrp*.)
- *IoCancelIrp* calls our cancel routine without releasing the lock. The cancel routine locates the correct *DEVQUEUE* and calls *CancelRequest*. *CancelRequest* immediately releases the global cancel spin lock (point 14).
- *CancelRequest* acquires the queue spin lock (point 15). Past this point, there can be no more races with other *DEVQUEUE* routines.
- *CancelRequest* removes the IRP from the queue (point 16) and then releases the spin lock. If *StartNextPacket* were to run now, it wouldn't find this IRP on the queue.
- *CancelRequest* completes the IRP with *STATUS_CANCELLED* (point 17).

8. Pathological IRP Cancellation

The most difficult IRP cancellation scenario to handle occurs when *IoCancelIrp* tries to cancel the IRP at the head of our queue while *StartNextPacket* is active. At point 12, *StartNextPacket* will nullify the cancel pointer. If the return value from *IoSetCancelRoutine* is not *NULL*, we've beaten *IoCancelIrp* to the punch and can go on to process the IRP (point 13).

If the return value from *IoSetCancelRoutine* is *NULL*, however, it means that *IoCancelIrp* has gotten there first. *CancelRequest* is probably waiting *right now* on another CPU for us to release the queue spin lock, whereupon it will dequeue the IRP and complete it. The trouble is, we've already removed the IRP from the queue. I'm a bit proud of the trick I devised for coping

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 163

with the situation: we simply initialize the linking field of the IRP as if it were the anchor of a list! The call to *RemoveEntryList* at point 16 in *CancelRequest* will perform several motions with no net result to "remove" the IRP from the degenerate list it now inhabits.

9. Things That Can't Happen or Won't Matter

The preceding list exhausts the possibilities for conflict between these *DEVQUEUE* routines and *IoCancelIrp*. (There is still a race between *IRP_MJ_CLEANUP* and IRP cancellation, but I'll discuss that a bit later in this chapter.) Here is a list of things that might be causing you needless worry:

- Could *CancelRoutine* be non-*NULL* when *StartPacket* gets control? It better not be, because a driver is supposed to remove its cancel routine from an IRP before sending the IRP to another driver. *StartPacket* contains an *ASSERT* to this effect. If you engage the Driver Verifier for your driver, it will verify that you nullify the cancel routine pointer in IRPs that you pass down the stack, but it will not verify that the drivers above you have done this for IRPs they pass to you.
- Could the cancel argument to *StartPacket* be *NULL*? It better not be: you might have noticed that much of the cancel logic I described hinges on whether the IRP's *CancelRoutine* pointer is *NULL*. *StartPacket* contains an *ASSERT* to test this assumption.
- Could someone call *IoCancelIrp* twice? The thing to think about is that the *Cancel* flag might be set in an IRP because of some number of primeval calls to *IoCancelIrp* and that someone might call *IoCancelIrp* one more time (getting a little impatient, are we?) while *StartPacket* is active. This wouldn't matter because our first test of the Cancel flag occurs after we install our cancel pointer. We would find the flag set to *TRUE* in this hypothetical situation and would therefore execute the second call to *IoSetCancelRoutine*. Either *IoCancelIrp* or we win the race to reset the cancel pointer to *NULL*, and whoever wins ends up completing the IRP. The residue from the primeval calls is simply irrelevant.

5.5.5 Cancelling IRPs You Create or Handle

Sometimes you'll want to cancel an IRP that you've created or passed to another driver. Great care is required to avoid an obscure, low-probability problem. Just for the sake of illustration, suppose you want to impose an overall 5-second timeout on a synchronous I/O operation. If the time period elapses, you want to cancel the operation. Here is some naive code that, you might suppose, would execute this plan:

```
SomeFunction()
  {
  KEVENT event;
  IO STATUS BLOCK iosb;
  KeInitializeEvent(&event, ...);
  PIRP Irp = IoBuildSynchronousFsdRequest(..., &event, &iosb);
  NTSTATUS status = IoCallDriver(DeviceObject, Irp);
  if (status == STATUS PENDING)
    {
    LARGE INTEGER timeout;
    timeout.QuadPart = -5 * 10000000;
A   if (KeWaitForSingleObject(&event, Executive, KernelMode,
      FALSE, &timeout) == STATUS TIMEOUT)
      {
      IoCancelIrp(Irp);  // <== don't do this!
B
      KeWaitForSingleObject(&event, Executive, KernelMode,
        FALSE, NULL);
      }
    }
  }
```

The first call (A) to *KeWaitForSingleObject* waits until one of two things happens. First, someone might complete the IRP, and the I/O Manager's cleanup code will then run and signal event.

Alternatively, the timeout might expire before anyone completes the IRP. In this case, *KeWaitForSingleObject* will return *STATUS_TIMEOUT*. The IRP should now be completed quite soon in one of two paths. The first completion path is taken when whoever was processing the IRP was really just about done when the timeout happened and has, therefore, already called (or will shortly call) *IoCompleteRequest*. The other completion path is through the cancel routine that, we must assume, the lower driver has installed. That cancel routine should complete the IRP. Recall that we have to trust other kernel-mode components to do their jobs, so we have to rely on whomever we sent the IRP to complete it soon. Whichever path is taken, the I/O Manager's completion logic will set event and store the IRP's ending status in iosb. The second call (B) to *KeWaitForSingleObject* makes sure that the event and iosb objects don't pass out of scope too soon. Without that second call, we might return from this function, thereby effectively deleting event and iosb. The I/O Manager might then end up walking on memory that belongs to some other subroutine.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 164

The problem with the preceding code is truly minuscule. Imagine that someone manages to call *IoCompleteRequest* for this IRP right around the same time we decide to cancel it by calling *IoCancelIrp*. Maybe the operation finishes shortly after the 5 - second timeout terminates the first KeWaitForSingleObject, for example. *IoCompleteRequest* initiates a process that finishes with a call to *IoFreeIrp*. If the call to *IoFreeIrp* were to happen before *IoCancelIrp* was done mucking about with the IRP, you can see that *IoCancelIrp* could inadvertently corrupt memory when it touched the CancelIrql, Cancel, and *CancelRoutine* fields of the IRP. It's also possible, depending on the exact sequence of events, for *IoCancelIrp* to call a cancel routine, just before someone clears the *CancelRoutine* pointer in preparation for completing the IRP, and for the cancel routine to be in a race with the completion process.

It's very unlikely that the scenario I just described will happen. But, as someone (James Thurber?) once said in connection with the chances of being eaten by a tiger on Main Street (one in a million, as I recall), "Once is enough." This kind of bug is almost impossible to find, so you want to prevent it if you can. I'll show you two ways of cancelling your own IRPs. One way is appropriate for synchronous IRPs, the other for asynchronous IRPs.

Don't Do This...
A once common but now deprecated technique for avoiding the tiger-on-main-street bug described in the text relies on the fact that, in earlier versions of Windows, the call to IoFreeIrp happened in the context of an APC in the thread that originates the IRP. You could make sure you were in that same thread, raise IRQL to APC_LEVEL, check whether the IRP had been completed yet, and (if not) call IoCancelIrp. You could be sure of blocking the APC and the problematic call to IoFreeIrp.

You shouldn't rely on future releases of Windows always using an APC to perform the cleanup for a synchronous IRP. Consequently, you shouldn't rely on boosting IRQL to APC_LEVEL as a way to avoid a race between IoCancelIrp and IoFreeIrp.

Cancelling Your Own Synchronous IRP

Refer to the example in the preceding section, which illustrates a function that creates a synchronous IRP, sends it to another driver, and then wants to wait no longer than 5 seconds for the IRP to complete. The key thing we need to accomplish in a solution to the race between *IoFreeIrp* and *IoCancelIrp* is to prevent the call to *IoFreeIrp* from happening until after any possible call to *IoCancelIrp*. We do this by means of a completion routine that returns *STATUS_MORE_PROCESSING_REQUIRED*, as follows:

```
SomeFunction()
  {
  KEVENT event;
  IO STATUS BLOCK iosb;
  KeInitializeEvent(&event, ...);
  PIRP Irp = IoBuildSynchronousFsdRequest(..., &event, &iosb);
  IoSetCompletionRoutine(Irp, OnComplete, (PVOID) &event, TRUE, TRUE, TRUE);
  NTSTATUS status = IoCallDriver(...);
  if (status == STATUS PENDING)
    {
    LARGE INTEGER timeout;
    timeout.QuadPart = -5 * 10000000;
A   if (KeWaitForSingleObject(&event, Executive, KernelMode,
      FALSE, &timeout) == STATUS TIMEOUT)
      {
      IoCancelIrp(Irp);  // <== okay in this context
B
      KeWaitForSingleObject(&event, Executive, KernelMode,
        FALSE, NULL);
      }
    }
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  }

NTSTATUS OnComplete(PDEVICE_OBJECT junk, PIRP Irp, PVOID pev)
  {
  if (Irp->PendingReturned)
    KeSetEvent((PKEVENT) pev, IO_NO_INCREMENT, FALSE);
  return STATUS_MORE_PROCESSING_REQUIRED;
  }
```

The new code in boldface prevents the race. Suppose *IoCallDriver* returns *STATUS_PENDING*. In a normal case, the operation will complete normally, and a lower-level driver will call *IoCompleteRequest*. Our completion routine gains control and signals the event on which our mainline is waiting. Because the completion routine returns *STATUS_MORE_PROCESSING_REQUIRED*, *IoCompleteRequest* will then stop working on this IRP. We eventually regain

IPR2024-00292
KASPERSKY EX 1008 Page 165

control in our SomeFunction and notice that our wait (the one labeled A) terminated normally. The IRP hasn't yet been cleaned up, though, so we need to call *IoCompleteRequest a second time* to trigger the normal cleanup mechanism.

Now suppose we decide we want to cancel the IRP and that Thurber's tiger is loose so we have to worry about a call to *IoFreeIrp* releasing the IRP out from under us. Our first wait (labeled A) finishes with *STATUS_TIMEOUT*, so we perform a second wait (labeled B). Our completion routine sets the event on which we're waiting. It will also prevent the cleanup mechanism from running by returning *STATUS_MORE_PROCESSING_REQUIRED*. *IoCancelIrp* can stomp away to its heart's content on our hapless IRP without causing any harm. The IRP can't be released until the second call to *IoCompleteRequest* from our mainline, and that can't happen until *IoCancelIrp* has safely returned.

Notice that the completion routine in this example calls KeSetEvent only when the IRP's *PendingReturned* flag is set to indicate that the lower driver's dispatch routine returned *STATUS_PENDING*. Making this step conditional is an optimization that avoids the potentially expensive step of setting the event when *SomeFunction* won't be waiting on the event in the first place.

I want to mention one last fine point in connection with the preceding code. The call to *IoCompleteRequest* at the very end of the subroutine will trigger a process that includes setting event and iosb so long as the IRP originally completed with a success status. In the first edition, I had an additional call to *KeWaitForSingleObject* at this point to make sure that event and iosb could not pass out of scope before the I/O Manager was done touching them. A reviewer pointed out that the routine that references event and iosb will already have run by the time *IoCompleteRequest* returns; consequently, the additional wait is not needed.

Cancelling Your Own Asynchronous IRP

To safely cancel an IRP that you've created with *IoAllocateIrp* or *IoBuildAsynchronousFsdRequest*, you can follow this general plan. First define a couple of extra fields in your device extension structure:

```
typedef struct  DEVICE EXTENSION {
  PIRP TheIrp;
  ULONG CancelFlag;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Initialize these fields just before you call *IoCallDriver* to launch the IRP:

```
pdx->TheIrp = IRP;
pdx->CancelFlag = 0;
IoSetCompletionRoutine(Irp,
  (PIO COMPLETION ROUTINE) CompletionRoutine,
  (PVOID) pdx, TRUE, TRUE, TRUE);
IoCallDriver(..., Irp);
```

If you decide later on that you want to cancel this IRP, do something like the following:

```
VOID CancelTheIrp(PDEVICE EXENSION pdx)
  {
1
  PIRP Irp =(PIRP) InterlockedExchangePointer((PVOID*)&pdx->TheIrp, NULL);
  if (Irp)
    {
    IoCancelIrp(Irp);
2
    if (InterlockedExchange(&pdx->CancelFlag, 1)
3
      IoFreeIrp(Irp);
    }
  }
```

This function dovetails with the completion routine you install for the IRP:

```
NTSTATUS CompletionRoutine(PDEVICE OBJECT junk, PIRP Irp,
  PDEVICE EXTENSION pdx)
  {
4
  if (InterlockedExchangePointer(&pdx->TheIrp, NULL)
5
  || InterlockedExchange(&pdx->CancelFlag, 1))
6
    IoFreeIrp(Irp);
```


IPR2024-00292
KASPERSKY EX 1008 Page 166

```
    return STATUS MORE PROCESSING REQUIRED;
    }
```

The basic idea underlying this deceptively simple code is that whichever routine sees the IRP last (either *CompletionRoutine* or *CancelTheIrp*) will make the requisite call to *IoFreeIrp*, at point 3 or 6. Here's how it works:

- The normal case occurs when you don't ever try to cancel the IRP. Whoever you sent the IRP to eventually completes it, and your completion routine gets control. The first *InterlockedExchangePointer* (point 4) returns the non-*NULL* address of the IRP. Since this is not 0, the compiler short-circuits the evaluation of the Boolean expression and executes the call to *IoFreeIrp*. Any subsequent call to *CancelTheIrp* will find the IRP pointer set to *NULL* at point 1 and won't do anything else.

- Another easy case to analyze occurs when *CancelTheIrp* is called long before anyone gets around to completing this IRP, which means that we don't have any actual race. At point 1, we nullify the *TheIrp* pointer. Because the IRP pointer was previously not *NULL*, we go ahead and call *IoCancelIrp*. In this situation, our call to *IoCancelIrp* will cause somebody to complete the IRP reasonably soon, and our completion routine runs. It sees *TheIrp* as *NULL* and goes on to evaluate the second half of the Boolean expression. Whoever executes the *InterlockedExchange* on *CancelFlag* first will get back 0 and skip calling *IoFreeIrp*. Whoever executes it second will get back 1 and will call *IoFreeIrp*.

- Now for the case we were worried about: suppose someone is completing the IRP right about the time *CancelTheIrp* wants to cancel it. The worst that can happen is that our completion routine runs before we manage to call *IoCancelIrp*. The completion routine sees *TheIrp* as *NULL* and therefore exchanges *CancelFlag* with 1. Just as in the previous case, the routine will get 0 as the return value and skip the *IoFreeIrp* call. *IoCancelIrp* can safely operate on the IRP. (It will presumably just return without calling a cancel routine because whoever completed this IRP will undoubtedly have set the *CancelRoutine* pointer to *NULL* first.)

The appealing thing about the technique I just showed you is its elegance: we rely solely on interlocked operations and therefore don't need any potentially expensive synchronization primitives.

Cancelling Someone Else's IRP

To round out our discussion of IRP cancellation, suppose someone sends you an IRP that you then forward to another driver. Situations might arise where you'd like to cancel that IRP. For example, perhaps you need that IRP out of the way so you can proceed with a power-down operation. Or perhaps you're waiting synchronously for the IRP to finish and you'd like to impose a timeout as in the first example of this section.

To avoid the *IoCancelIrp*/*IoFreeIrp* race, you need to have your own completion routine in place. The details of the coding then depend on whether you're waiting for the IRP.

Canceling Someone Else's IRP on Which You're Waiting

Suppose your dispatch function passes down an IRP and waits synchronously for it to complete. (See usage scenario 7 at the end of this chapter for the cookbook version.) Use code like this to cancel the IRP if it doesn't finish quickly enough to suit you:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  KEVENT event;
  KeInitializeEvent(&event, NotificationEvent, FALSE);
  IoSetCompletionRoutine(Irp, OnComplete, (PVOID) &event, TRUE, TRUE, TRUE);
  NTSTATUS status = IoCallDriver(...);
  if (status == STATUS PENDING)
    {
    LARGE INTEGER timeout;
    timeout.QuadPart = -5 * 10000000;
    if (KeWaitForSingleObject(&event, Executive, KernelMode,
      FALSE, &timeout) == STATUS TIMEOUT)
      {
      IoCancelIrp(Irp);
      KeWaitForSingleObject(&event, Executive, KernelMode,
        FALSE, NULL);
      }
    }
  status = Irp->IoStatus.Status;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return status;
  }
```


IPR2024-00292
KASPERSKY EX 1008 Page 167

```
NTSTATUS OnComplete(PDEVICE OBJECT junk, PIRP Irp, PVOID pev)
  {
  if (Irp->PendingReturned)
    KeSetEvent((PKEVENT) pev, IO NO INCREMENT, FALSE);
  return STATUS MORE PROCESSING REQUIRED;
  }
```

This code is almost the same as what I showed earlier for canceling your own synchronous IRP. The only difference is that this example involves a dispatch routine, which must return a status code. As in the earlier example, we install our own completion routine to prevent the completion process from running to its ultimate conclusion before we get past the point where we might call *IoCancelIrp*.

You might notice that I didn't say anything about whether the IRP itself was synchronous or asynchronous. This is because the difference between the two types of IRP only matters to the driver that creates them in the first place. File system drivers must make distinctions between synchronous and asynchronous IRPs with respect to how they call the system cache manager, but device drivers don't typically have this complication. What matters to a lower-level driver is whether it's appropriate to block a thread in order to handle an IRP synchronously, and that depends on the current IRQL and whether you're in an arbitrary or a nonarbitrary thread.

Canceling Someone Else's IRP on Which You're Not Waiting

Suppose you've forwarded somebody else's IRP to another driver, but you weren't planning to wait for it to complete. For whatever reason, you decide later on that you'd like to cancel that IRP.

```
typedef struct  DEVICE EXTENSION {
  PIRP TheIrp;
  ULONG CancelFlag;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx =
    (PDEVICE EXTENSION) fdo->DeviceExtension;
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE) OnComplete,
    (PVOID) pdx,
    TRUE, TRUE, TRUE);
  pdx->CancelFlag = 0;
  pdx->TheIrp = Irp;
  IoMarkIrpPending(Irp);
  IoCallDriver(pdx->LowerDeviceObject, Irp);
  return STATUS PENDING;
  }

VOID CancelTheIrp(PDEVICE EXTENSION pdx)
  {
  PIRP Irp = (PIRP) InterlockedExchangePointer(
    (PVOID*) &pdx->TheIrp, NULL);
  if (Irp)
    {
    IoCancelIrp(Irp);
    if (InterlockedExchange(&pdx->CancelFlag, 1))
      IoCompleteRequest(Irp, IO NO INCREMENT);
    }
  }

NTSTATUS OnComplete(PDEVICE OBJECT fdo, PIRP Irp,
  PDEVICE EXTENSION pdx)
  {
  if (InterlockedExchangePointer((PVOID*) &pdx->TheIrp, NULL)
    || InterlockedExchange(&pdx->CancelFlag, 1))
    return STATUS SUCCESS;
  return STATUS MORE PROCESSING REQUIRED;
  }
```

This code is similar to the code I showed earlier for cancelling your own asynchronous IRP. Here, however, allowing *IoCompleteRequest* to finish completing the IRP takes the place of the call to *IoFreeIrp* we made when we were dealing with our own IRP. If the completion routine is last on the scene, it returns *STATUS_SUCCESS* to allow *IoCompleteRequest* to finish completing the IRP. If *CancelTheIrp* is last on the scene, it calls *IoCompleteRequest* to resume the completion processing that

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 168

the completion routine short-circuited by returning *STATUS_MORE_PROCESSING_REQUIRED*.

One extremely subtle point regarding this example is the call to *IoMarkIrpPending* in the dispatch routine. Ordinarily, it would be safe to just do this step conditionally in the completion routine, but not this time. If we should happen to call *CancelTheIrp* in the context of some thread other than the one in which the dispatch routine runs, the pending flag is needed so that *IoCompleteRequest* will schedule an APC to clean up the IRP in the proper thread. The easiest way to make that true is simple—always mark the IRP pending.

5.5.6 Handling *IRP_MJ_CLEANUP*

Closely allied to the subject of IRP cancellation is the I/O request with the major function code *IRP_MJ_CLEANUP*. To explain how you should process this request, I need to give you a little additional background.

When applications and other drivers want to access your device, they first open a handle to the device. Applications call *CreateFile* to do this; drivers call *ZwCreateFile*. Internally, these functions create a kernel file object and send it to your driver in an *IRP_MJ_CREATE* request. When the entity that opened the handle is done accessing your driver, it will call another function, such as *CloseHandle* or *ZwClose*. Internally, these functions send your driver an *IRP_MJ_CLOSE* request. Just before sending you the *IRP_MJ_CLOSE*, however, the I/O Manager sends you an *IRP_MJ_CLEANUP* so that you can cancel any IRPs that belong to the same file object but that are still sitting in one of your queues. From the perspective of your driver, the one thing all the requests have in common is that the stack location you receive points to the same file object in every instance.

Figure 5-10 illustrates your responsibility when you receive *IRP_MJ_CLEANUP*. You should run through your queues of IRPs, removing those that are tagged as belonging to the same file object. You should complete those IRPs with *STATUS_CANCELLED*.

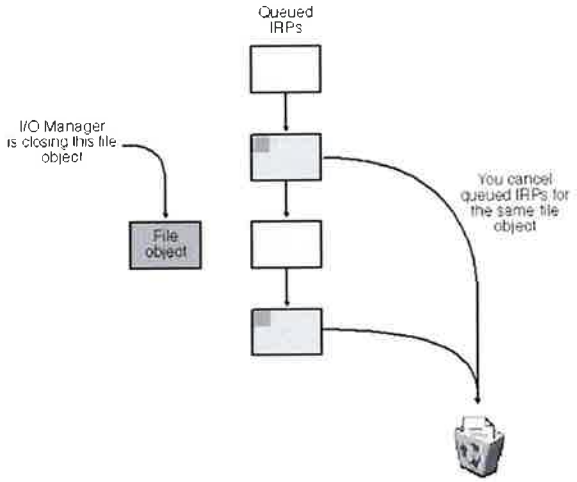


Figure 5-10. *Driver responsibility for IRP_MJ_CLEANUP.*

File Objects
Ordinarily, just one driver (the function driver, in fact) in a device stack implements all three of the following requests: *IRP_MJ_CREATE*, *IRP_MJ_CLOSE*, and *IRP_MJ_CLEANUP*. The I/O Manager creates a file object (a regular kernel object) and passes it in the I/O stack to the dispatch routines for all three of these IRPs. Anybody who sends an IRP to a device should have a pointer to the same file object and should insert that pointer into the I/O stack as well. The driver that handles these three IRPs acts as the owner of the file object in some sense, in that it's the driver that's entitled to use the *FsContext* and *FsContext2* fields of the object. So your *DispatchCreate* routine can put something into one of these context fields for use by other dispatch routines and for eventual cleanup by your *DispatchClose* routine.

It's easy to get confused about *IRP_MJ_CLEANUP*. In fact, programmers who have a hard time understanding IRP cancellation sometimes decide (incorrectly) to just ignore this IRP. You need both cancel and cleanup logic in your driver, though:

- *IRP_MJ_CLEANUP* means a handle is being closed. You should purge all the IRPs that pertain to that handle.
- The I/O Manager and other drivers cancel individual IRPs for a variety of reasons that have nothing to do with closing handles.
- One of the times the I/O Manager cancels IRPs is when a thread terminates. Threads often terminate because their parent process is terminating, and the I/O Manager will also automatically close all handles that are still open when a process

IPR2024-00292
KASPERSKY EX 1008 Page 169

terminates. The coincidence between this kind of cancellation and the automatic handle closing contributes to the incorrect idea that a driver can get by with support for just one concept.

In this book, I'll show you two ways of painlessly implementing support for *IRP_MJ_CLEANUP*, depending on whether you're using one of my *DEVQUEUE* objects or one of Microsoft's cancel-safe queues.

5.5.7 Cleanup with a *DEVQUEUE*

If you've used a *DEVQUEUE* to queue IRPs, your *IRP_MJ_CLEANUP* routine will be astonishingly simple:

```
NTSTATUS DispatchCleanup(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PFILE_OBJECT fop = stack->FileObject;
  CleanupRequests(&pdx->dqReadWrite, fop, STATUS_CANCELLED);
  return CompleteRequest(Irp, STATUS_SUCCESS, 0);
  }
```

CleanupRequests will remove all IRPs from the queue that belong to the same file object and will complete those IRPs with *STATUS_CANCELLED*. Note that you complete the *IRP_MJ_CLEANUP* request itself with *STATUS_SUCCESS*.

CleanupRequests contains a wealth of detail:

```
VOID CleanupRequests(PDEVQUEUE pdq, PFILE_OBJECT fop, NTSTATUS status)
  {
  LIST_ENTRY cancellist;
1
  InitializeListHead(&cancellist);
  KIRQL oldirql;
  KeAcquireSpinLock(&pdq->lock, &oldirql);
  PLIST_ENTRY first = &pdq->head;
  PLIST_ENTRY next;
2
  for (next = first->Flink; next != first; )
    {
    PIRP Irp = CONTAINING_RECORD(next, IRP,
      Tail.Overlay.ListEntry);
3
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
4
    PLIST_ENTRY current = next;
    next = next->Flink;
    if (fop && stack->FileObject != fop)
      continue;
5
    if (!IoSetCancelRoutine(Irp, NULL))
      continue;
6
    RemoveEntryList(current);
    InsertTailList(&cancellist, current);
    }
7
  KeReleaseSpinLock(&pdq->lock, oldirql);
  while (!IsListEmpty(&cancellist))
    {
    next = RemoveHeadList(&cancellist);
    PIRP Irp = CONTAINING_RECORD(next, IRP,
      Tail.Overlay.ListEntry);
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
  }
```

1. Our strategy will be to move the IRPs that need to be cancelled into a private queue under protection of the queue's spin lock. Hence, we initialize the private queue and acquire the spin lock before doing anything else.

2. This loop traverses the entire queue until we return to the list head. Notice the absence of a loop increment step—the third clause in the for statement. I'll explain in a moment why it's desirable to have no loop increment.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 170

3. If we're being called to help out with *IRP_MJ_CLEANUP*, the fop argument is the address of a file object that's about to be closed. We're supposed to isolate the IRPs that pertain to the same file object, which requires us to first find the stack location.

4. If we decide to remove this IRP from the queue, we won't thereafter have an easy way to find the next IRP in the main queue. We therefore perform the loop increment step here.

5. This especially clever statement comes to us courtesy of Jamie Hanrahan. We need to worry that someone might be trying to cancel the IRP that we're currently looking at during this iteration. They could get only as far as the point where *CancelRequest* tries to acquire the spin lock. Before getting that far, however, they necessarily had to execute the statement inside *IoCancelIrp* that nullifies the cancel routine pointer. If we find that pointer set to *NULL* when we call *IoSetCancelRoutine*, therefore, we can be sure that someone really is trying to cancel this IRP. By simply skipping the IRP during this iteration, we allow the cancel routine to complete it later on.

6. Here's where we take the IRP out of the main queue and put it in the private queue instead.

7. Once we finish moving IRPs into the private queue, we can release our spin lock. Then we cancel all the IRPs we moved.

5.5.8 Cleanup with a Cancel-Safe Queue

To easily clean up IRPs that you've queued by calling *IoCsqInsertIrp*, simply adopt the convention that the peek context parameter you use with *IoCsqRemoveNextIrp*, if not NULL, will be the address of a *FILE_OBJECT*. Your *IRP_MJ_CANCEL* routine will look like this (compare with the Cancel sample in the DDK):

```
NTSTATUS DispatchCleanup(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PFILE_OBJECT fop = stack->FileObject;
  PIRP qirp;
  while ((qirp = IoCsqRemoveNextIrp(&pdx->csq, fop)))
    CompleteRequest(qirp, STATUS_CANCELLED, 0);
  return CompleteRequest(Irp, STATUS_SUCCESS, 0);
  }
```

Implement your *PeekNextIrp* callback routine this way:

```
PIRP PeekNextIrp(PIO_CSQ csq, PIRP Irp, PVOID PeekContext)
  {
  PDEVICE_EXTENSION pdx = GET_DEVICE_EXTENSION(csq);
  PLIST_ENTRY next = Irp ? Irp->Tail.Overlay.ListEntry.Flink
    : pdx->IrpQueueAnchor.Flink;
  while (next != &pdx->IrpQueueAnchor)
    {
    PIRP NextIrp = CONTAINING_RECORD(next, IRP,
      Tail.Overlay.ListEntry);
    PIO_STACK_LOCATION stack =
      IoGetCurrentIrpStackLocation(NextIrp);
    if (!PeekContext || (PFILE_OBJECT) PeekContext == stack->FileObject)
      return NextIrp;
    next = next->Flink;
    }
  return NULL;
  }
```

5.6 Summary—Eight IRP-Handling Scenarios

Notwithstanding the length of the preceding explanations, IRP handling is actually quite easy. By my reckoning, only eight significantly different scenarios are in common use, and the code required to handle those scenarios is pretty simple. In this final section of this chapter, I've assembled some pictures and code samples to help you sort out all the theoretical knowledge.

Because this section is intended as a cookbook that you can use without completely understanding every last nuance, I've included calls to the remove lock functions that I'll discuss in detail in Chapter 6. I've also used the shorthand *IoSetCompletionRoutine[Ex]* to indicate places where you ought to call *IoSetCompletionRoutineEx*, in a system where it's available, to install a completion routine. I've also used an overloaded version of my *CompleteRequest* helper routine that doesn't change *IoStatus.Information* in these examples because that would be correct for *IRP_MJ_PNP* and not incorrect for other types of IRP.

IPR2024-00292
KASPERSKY EX 1008 Page 171

5.6.1 Scenario 1—Pass Down with Completion Routine

In this scenario, someone sends you an IRP. You'll forward this IRP to the lower driver in your PnP stack, and you'll do some postprocessing in a completion routine. See Figure 5-11. Adopt this strategy when all of the following are true:

- Someone is sending you an IRP (as opposed to you creating the IRP yourself).
- The IRP might arrive at *DISPATCH_LEVEL* or in an arbitrary thread (so you can't block while the lower drivers handle the IRP).
- Your postprocessing can be done at *DISPATCH_LEVEL* if need be (because completion routines might be called at *DISPATCH_LEVEL*).

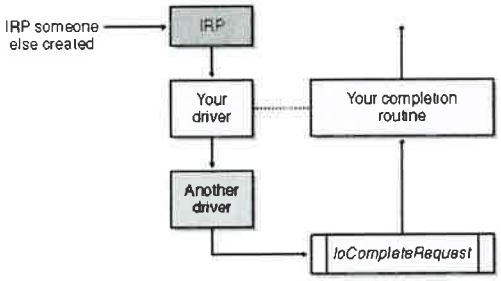


Figure 5-11. *Pass down with completion routine.*

Your dispatch and completion routines will have this skeletal form:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp,
    (PIO COMPLETION ROUTINE) CompletionRoutine, pdx, TRUE, TRUE, TRUE);
  return IoCallDriver(fdo, Irp);
  }

NTSTATUS CompletionRoutine(PDEVICE OBJECT fdo, PIRP Irp,PDEVICE EXTENSION pdx)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
  <whatever post processing you wanted to do>
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return STATUS SUCCESS;
  }
```

5.6.2 Scenario 2—Pass Down Without Completion Routine

In this scenario, someone sends you an IRP. You'll forward the IRP to the lower driver in your PnP stack, but you don't need to do anything with the IRP. See Figure 5-12. Adopt this strategy, which can also be called the "Let Mikey try it" approach, when both of the following are true:

- Someone is sending you an IRP (as opposed to you creating the IRP yourself).
- You don't process this IRP, but a driver below you might want to.

IPR2024-00292
KASPERSKY EX 1008 Page 172

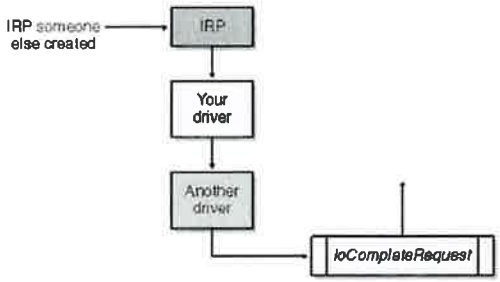


Figure 5-12. *Pass down without completion routine.*

This scenario is often used in a filter driver, which should act as a simple conduit for every IRP that it doesn't specifically need to filter.

I recommend writing the following helper routine, which you can use whenever you need to employ this strategy.

```
NTSTATUS ForwardAndForget(PDEVICE EXTENSION pdx, PIRP Irp)
  {
  //PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status);
  IoSkipCurrentIrpStackLocation (Irp);
  status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
  }
```

5.6.3 Scenario 3—Complete in the Dispatch Routine

In this scenario, you immediately complete an IRP that someone sends you. See Figure 5-13. Adopt this strategy when:

- Someone is sending you an IRP (as opposed to you creating the IRP yourself), and
- You can process the IRP immediately. This would be the case for many kinds of I/O control (IOCTL) requests. Or
- Something is obviously wrong with the IRP, in which case causing it to fail immediately might be the kindest thing to do.

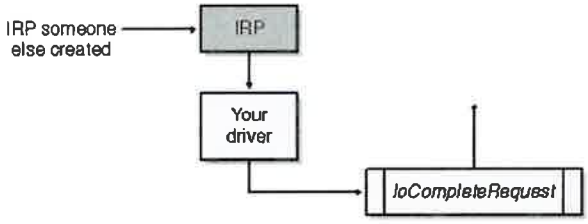


Figure 5-13. *Complete in the dispatch routine.*

Your dispatch routine has this skeletal form:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  <process the IRP>
  Irp->IoStatus.Status = STATUS XXX;
  Irp->IoStatus.Information = YYY;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return STATUS XXX;
  }
```


IPR2024-00292
KASPERSKY EX 1008 Page 173

5.6.4 Scenario 4—Queue for Later Processing

In this scenario, someone sends you an IRP that you can't handle right away. You put the IRP on a queue for later processing in a *StartIo* routine. See Figure 5-14. Adopt this strategy when both of the following are true:

- Someone is sending you an IRP (as opposed to you creating the IRP yourself).
- You don't know that you can process the IRP right away. This would frequently be the case for IRPs that require serialized hardware access, such as reads and writes.

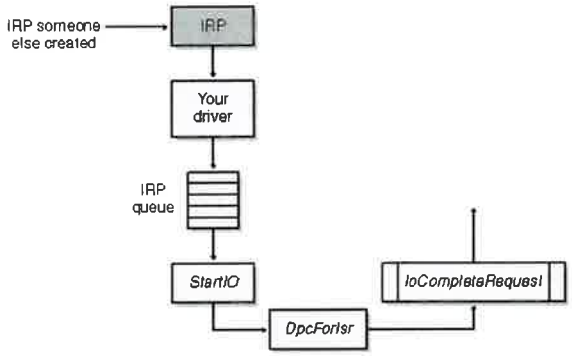


Figure 5-14. *Queue for later processing.*

Although you have many choices, a typical way of implementing this scenario involves using a DEVQUEUE to manage the IRP queue. The following fragments show how various parts of a driver for a programmed I/O interrupt-driven device would interact. Only the parts shown in boldface pertain specifically to IRP handling.

```
typedef struct  DEVICE EXTENSION {
:
  DEVQUEUE dqReadWrite;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
:
  InitializeQueue(&pdx->dqReadWrite, StartIo);
  IoInitializeDpcRequest(fdo, (PIO DPC ROUTINE) DpcForIsr);
:
  }

NTSTATUS DispatchReadWrite(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  IoMarkIrpPending(Irp);
  StartPacket(&pdx->dqReadWrite, fdo, Irp, CancelRoutine);
  return STATUS PENDING;
  }

VOID CancelRoutine(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  CancelRequest(&pdx->dqReadWrite, Irp);
  }
VOID StartIo(PDEVICE OBJECT fdo, PIRP Irp)
  {
:
  }

BOOLEAN OnInterrupt(PKINTERRUPT junk, PDEVICE EXTENSION pdx)
  {
:
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  Irp->IoStatus.Status = STATUS XXX;
  Irp->IoStatus.Information = YYY;
```


IPR2024-00292
KASPERSKY EX 1008 Page 174

```
  IoRequestDpc(pdx->DeviceObject, NULL, pdx);
  :
  }

VOID DpcForIsr(PKDPC junk1, PDEVICE OBJECT fdo, PIRP junk2,
  PDEVICE EXTENSION pdx)
  {
  :
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  StartNextPacket(&pdx->dqReadWrite, fdo);
  IoCompleteRequest(Irp, IO NO INCREMENT);
  }
```

5.6.5 Scenario 5—Your Own Asynchronous IRP

In this scenario, you create an asynchronous IRP, which you forward to another driver. See Figure 5-15. Adopt this strategy when the following conditions are true:

You need another driver to perform an operation on your behalf.

Either you're in an arbitrary thread (which you shouldn't block) or you're running at DISPATCH_LEVEL (in which case you can't block).

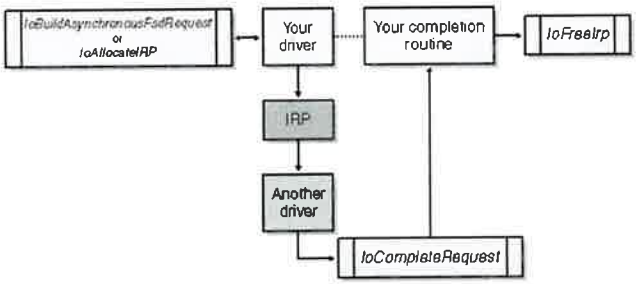


Figure 5-15. *Your own asynchronous IRP.*

You'll have code like the following in your driver. This won't necessarily be in an IRP dispatch routine, and the target device object won't necessarily be the next lower one in your PnP stack. Look in the DDK documentation for full details about how to call *IoBuildAsynchronousFsdRequest* and *IoAllocateIrp*.

```
SOMETYPE SomeFunction(PDEVICE EXTENSION pdx, PDEVICE OBJECT DeviceObject)
  {
A
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, (PVOID) 42);
A
  if (!NT SUCCESS(status))
A
    return <status>;
  PIRP Irp;
  Irp = IoBuildAsynchronousFsdRequest(IRP_MJ_XXX, DeviceObject, ...);

    -or-

  Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
  PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
  stack->MajorFunction = IRP MJ XXX;
  <additional initialization)
  IoSetCompletionRoutine[Ex]([pdx->DeviceObject,] Irp,
    (PIO COMPLETION ROUTINE) CompletionRoutine, pdx,
    TRUE, TRUE, TRUE);
B
  ObReferenceObject(DeviceObject);
  IoCallDriver(DeviceObject, Irp);
B
  ObDereferenceObject(DeviceObject);
  }

NTSTATUS CompletionRoutine(PDEVICE_OBJECT junk, PIRP Irp, PDEVICE_EXTENSION pdx)
```


IPR2024-00292
KASPERSKY EX 1008 Page 175

```
{
<IRP cleanup -- see below>
IoFreeIrp(Irp);
IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42);
return STATUS_MORE_PROCESSING_REQUIRED;
}
```

The calls to *IoAcquireRemoveLock* and *IoReleaseRemoveLock* (the points labeled A) are necessary only if the device to which you're sending this IRP is the *LowerDeviceObject* in your PnP stack. The 42 is an arbitrary tag—it's simply too complicated to try to acquire the remove lock after the IRP exists just so we can use the IRP pointer as a tag in the debug build.

The calls to *ObReferenceObject* and *ObDereferenceObject* that precede and follow the call to *IoCallDriver* (the points labeled B) are necessary only when you've used *IoGetDeviceObjectPointer* to obtain the *DeviceObject* pointer and when the completion routine (or something it calls) will release the resulting reference to a device or file object.

You do not have both the A code and the B code—you have one set or neither.

If you use *IoBuildAsynchronousFsdRequest* to build an *IRP_MJ_READ* or *IRP_MJ_WRITE*, you have some relatively complex cleanup to perform in the completion routine.

Cleanup for DO_DIRECT_IO Target

If the target device object indicates the *DO_DIRECT_IO* buffering method, you'll have to release the memory descriptor lists that the I/O Manager allocated for your data buffer:

```
NTSTATUS CompletionRoutine(...)
  {
  PMDL mdl;
  while ((mdl = Irp->MdlAddress))
    {
    Irp->MdlAddress = mdl->Next;
    MmUnlockPages(mdl); // <== only if you earlier
                        // called MmProbeAndLockPages
    IoFreeMdl(mdl);
    }
  IoFreeIrp(Irp);
  <optional release of remove lock>
  return STATUS_MORE_PROCESSING_REQUIRED;
  }
```

Cleanup for DO_BUFFERED_IO Target

If the target device object indicates *DO_BUFFERED_IO*, the I/O Manager will create a system buffer. Your completion routine should theoretically copy data from the system buffer to your own buffer and then release the system buffer. Unfortunately, the flag bits and fields needed to do this are not documented in the DDK. My advice is to simply not send reads and writes directly to a driver that uses buffered I/O. Instead, call *ZwReadFile* or *ZwWriteFile*.

Cleanup for Other Targets

If the target device indicates neither DO_DIRECT_IO nor DO_BUFFERED_IO, there is no additional cleanup. Phew!

5.6.6 Scenario 6—Your Own Synchronous IRP

In this scenario, you create a synchronous IRP, which you forward to another driver. See Figure 5-16. Adopt this strategy when all of the following are true:

- You need another driver to perform an operation on your behalf.
- You must wait for the operation to complete before proceeding.
- You're running at PASSIVE_LEVEL in a nonarbitrary thread.

IPR2024-00292
KASPERSKY EX 1008 Page 176

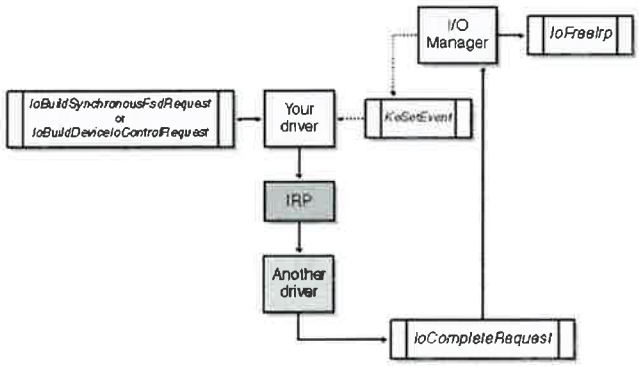


Figure 5-16. *Your own synchronous IRP.*

You'll have code like the following in your driver. This won't necessarily be in an IRP dispatch routine, and the target device object won't necessarily be the next lower one in your PnP stack. Look in the DDK documentation for full details about how to call *IoBuildSynchronousFsdRequest* and *IoBuildDeviceIoControlRequest*.

```
SOMETYPE SomeFunction(PDEVICE EXTENSION pdx, PDEVICE OBJECT DeviceObject)
{
A
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock,
    (PVOID) 42);
A
  if (!NT SUCCESS(status))
A
    return <status>;
  PIRP Irp;
  KEVENT event;
  IO STATUS BLOCK iosb;
  KeInitializeEvent(&event, NotificationEvent, FALSE);
  Irp = IoBuildSynchronousFsdRequest(IRP MJ XXX,
    DeviceObject, ..., &event, &iosb);

    -or-

  Irp = IoBuildDeviceIoControlRequest(IOCTL XXX, DeviceObject,
    ..., &event, &iosb);
  status = IoCallDriver(DeviceObject, Irp);
  if (status == STATUS PENDING)
    {
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
    status = iosb.Status;
    }
A
  IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42);
  .
  .
  .

}
```

As in scenario 5, the calls to *IoAcquireRemoveLock* and *IoReleaseRemoveLock* (the points labeled A) are necessary only if the device to which you're sending this IRP is the *LowerDeviceObject* in your PnP stack. The 42 is an arbitrary tag—it's simply too complicated to try to acquire the remove lock after the IRP exists just so we can use the IRP pointer as a tag in the debug build.

We'll use this scenario frequently in Chapter 12 to send USB Request Blocks (URBs) synchronously down the stack. In the examples we'll study there, we'll usually be doing this in the context of an IRP dispatch routine that independently acquires the remove lock. Therefore, you won't see the extra remove lock code in those examples.

You do not clean up after this IRP! The I/O Manager does it automatically.

5.6.7 Scenario 7—Synchronous Pass Down

In this scenario, someone sends you an IRP. You pass the IRP down synchronously in your PnP stack and then continue processing. See Figure 5-17. Adopt this strategy when all of the following are true:

IPR2024-00292
KASPERSKY EX 1008 Page 177

- Someone is sending you an IRP (as opposed to you creating the IRP yourself).
- You're running at *PASSIVE_LEVEL* in a nonarbitrary thread.
- Your postprocessing for the IRP must be done at *PASSIVE_LEVEL*.

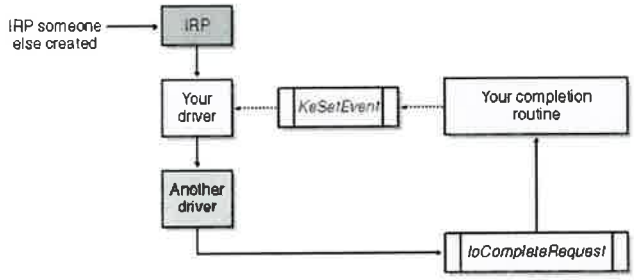


Figure 5-17. *Synchronous pass down.*

A good example of when you would need to use this strategy is while processing an *IRP_MN_START_DEVICE* flavor of PnP request.

I recommend writing two helper routines to make it easy to perform this synchronous pass-down operation:

```
NTSTATUS ForwardAndWait(PDEVICE EXTENSION pdx, PIRP Irp)
  {
  KEVENT event;
  KeInitialize(&event, NotificationRoutine, FALSE);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE)
    ForwardAndWaitCompletionRoutine, &event, TRUE, TRUE, TRUE);
  NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  if (status == STATUS PENDING)
    {
    KeWaitForSingleObject(&event, Executive, KernelMode,
      FALSE, NULL);
    status = Irp->IoStatus.Status;
    }
  return status;
  }

NTSTATUS ForwardAndWaitCompletionRoutine(PDEVICE OBJECT fdo,
  PIRP Irp, PKEVENT pev)
  {
  if (Irp->PendingReturned)
    KeSetEvent(pev, IO NO INCREMENT, FALSE);
  return STATUS MORE PROCESSING REQUIRED;
  }
```

The caller of this routine needs to call *IoCompleteRequest* for this IRP and to acquire and release the remove lock. It's inappropriate for *ForwardAndWait* to contain the remove lock logic because the caller might not want to release the lock so soon.

Note that the Windows XP DDK function *IoForwardIrpSynchronously* encapsulates these same steps.

5.6.8 Scenario 8—Asynchronous IRP Handled Synchronously

In this scenario, you create an asynchronous IRP, which you forward to another driver. Then you wait for the IRP to complete. See Figure 5-18. Adopt this strategy when all of the following are true:

- You need another driver to perform an operation on your behalf.
- You need to wait for the operation to finish before you can go on.
- You're running at *APC_LEVEL* in a nonarbitrary thread.

IPR2024-00292
KASPERSKY EX 1008 Page 178

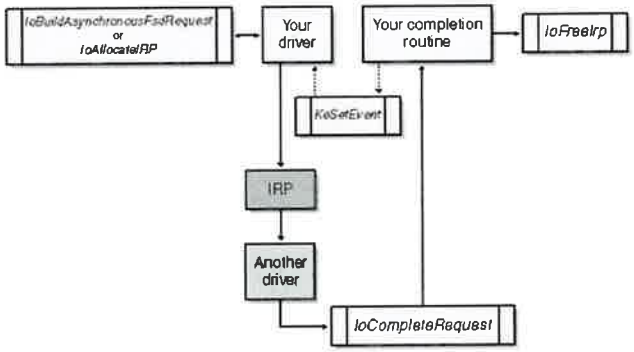


Figure 5-18. *Asynchronous IRP handled synchronously.*

I use this technique when I've acquired an executive fast mutex and need to perform a synchronous operation. Your code combines elements you've seen before (compare with scenarios 5 and 7):

```
SOMETYPE SomeFunction(PDEVICE EXTENSION pdx, PDEVICE OBJECT DeviceObject)
  {
A
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, (PVOID) 42);
A
  if (!NT SUCCESS(status))
A
    return <status>;
  PIRP Irp;
  Irp = IoBuildAsynchronousFsdRequest(IRP_MJ_XXX, DeviceObject, ...);

    -or-

  Irp = IoAllocateIrp(DeviceObject->StackSize, FALSE);
  PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
  Stack->MajorFunction = IRP MJ XXX;
  <additional initialization)
  KEVENT event;
  KeInitializeEvent(&event, NotificationEvent, FALSE);
  IoSetCompletionRoutine[Ex]([pdx->DeviceObject], Irp,
    (PIO COMPLETION ROUTINE) CompletionRoutine,
    &event, TRUE, TRUE, TRUE);
  status = IoCallDriver(DeviceObject, Irp);
  if (status == STATUS PENDING)
    KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
A
  IoReleaseRemoveLock(&pdx->RemoveLock, (PVOID) 42);
  }

NTSTATUS CompletionRoutine(PDEVICE OBJECT junk, PIRP Irp, PKEVENT pev)
  {
  if (Irp->PendingReturned)
    KeSetEvent(pev, EVENT INCREMENT, FALSE);
  <IRP cleanup -- see above>
  IoFreeIrp(Irp);
  return STATUS MORE PROCESSING REQUIRED;
  }
```

The portions that differ from scenario 5 are in boldface.

As in earlier scenarios, the calls to *IoAcquireRemoveLock* and *IoReleaseRemoveLock* (the points labeled A) are necessary only if the device to which you're sending this IRP is the *LowerDeviceObject* in your PnP stack. The 42 is an arbitrary tag—it's simply too complicated to try to acquire the remove lock after the IRP exists just so we can use the IRP pointer as a tag in the debug build.

Note that you must still perform all the same cleanup discussed earlier because the I/O Manager doesn't clean up after an asynchronous IRP. You might also need to provide for cancelling this IRP, in which case you should use the technique

IPR2024-00292
KASPERSKY EX 1008 Page 179

Chapter 6

Plug and Play for Function Drivers

The Plug and Play (PnP) Manager communicates information and requests to device drivers via I/O request packets (IRPs) with the major function code *IRP_MJ_PNP*. This type of request was new with Microsoft Windows 2000 and the Windows Driver Model (WDM): previous versions of Microsoft Windows NT required device drivers to do most of the work of detecting and configuring their devices. Happily, WDM drivers can let the PnP Manager do that work. To work with the PnP Manager, driver authors will have to understand a few relatively complicated IRPs.

Plug and Play requests play two roles in the WDM. In their first role, these requests instruct the driver when and how to configure or deconfigure itself and the hardware. Table 6-1 lists the roughly two dozen minor functions that a PnP request can designate. Only a bus driver handles the nine minor functions shown with an asterisk; a filter driver or function driver would simply pass these IRPs down the stack. Of the remaining minor functions, three have special importance to a typical filter driver or function driver. The PnP Manager uses *IRP_MN_START_DEVICE* to inform the function driver which I/O resources it has assigned to the hardware and to instruct the function driver to do any necessary hardware and software setup so that the device can function. *IRP_MN_STOP_DEVICE* tells the function driver to shut down the device. *IRP_MN_REMOVE_DEVICE* tells the function driver to shut down the device and release the associated device object. I'll discuss these three minor functions in detail in this chapter and the next; along the way, I'll also describe the purpose of the other unstarred minor functions that a filter driver or function driver might need to handle.

IRP Minor Function Code	Description
IRP_MN_START_DEVICE	Configure and initialize device
IRP_MN_QUERY_REMOVE_DEVICE	Can device be removed safely?
IRP_MN_REMOVE_DEVICE	Shut down and remove device
IRP_MN_CANCEL_REMOVE_DEVICE	Ignore previous QUERY_REMOVE
IRP_MN_STOP_DEVICE	Shut down device
IRP_MN_QUERY_STOP_DEVICE	Can device be shut down safely?
IRP_MN_CANCEL_STOP_DEVICE	Ignore previous QUERY_STOP
IRP_MN_QUERY_DEVICE_RELATIONS	Get list of devices that are related in some specified way
IRP_MN_QUERY_INTERFACE	Obtain direct-call function addresses
IRP_MN_QUERY_CAPABILITIES	Determine capabilities of device
*IRP_MN_QUERY_RESOURCES**	Determine boot configuration
*IRP_MN_QUERY_RESOURCE_REQUIREMENTS**	Determine I/O resource requirements
*IRP_MN_QUERY_DEVICE_TEXT**	Obtain description or location string
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	Modify I/O resource requirements list
*IRP_MN_READ_CONFIG**	Read configuration space
*IRP_MN_WRITE_CONFIG**	Write configuration space
*IRP_MN_EJECT**	Eject the device
*IRP_MN_SET_LOCK**	Lock/unlock device against ejection
*IRP_MN_QUERY_ID**	Determine hardware ID of device
IRP_MN_QUERY_PNP_DEVICE_STATE	Determine state of device
*IRP_MN_QUERY_BUS_INFORMATION**	Determine parent bus type
IRP_MN_DEVICE_USAGE_NOTIFICATION	Note creation or deletion of paging, dump, or hibernate file
IRP_MN_SURPRISE_REMOVAL	Note fact that device has been removed

Table 6-1. *Minor Function Codes for IRP_MJ_PNP (* Indicates Handled Only by Bus Drivers)*

A second and more complicated purpose of PnP requests is to guide the driver through a series of state transitions, as illustrated in Figure 6-1. *WORKING* and *STOPPED* are the two fundamental states of the device. The *STOPPED* state is the initial state of a device immediately after you create the device object. The *WORKING* state indicates that the device is fully operational. Two of the intermediate states—*PENDINGSTOP* and *PENDINGREMOVE*—arise because of queries that all drivers for a device must process before making the transition from *WORKING*. *SURPRISEREMOVED* occurs after the sudden and unexpected removal of the physical hardware.

I introduced my *DEVQUEUE* queue management routines in the preceding chapter. The main reason for needing a custom queuing scheme in the first place is to facilitate the PnP state transitions shown in Figure 6-1 and the power state transitions I'll discuss in Chapter 8. I'll describe the *DEVQUEUE* routines that support these transitions in this chapter.

IPR2024-00292
KASPERSKY EX 1008 Page 181

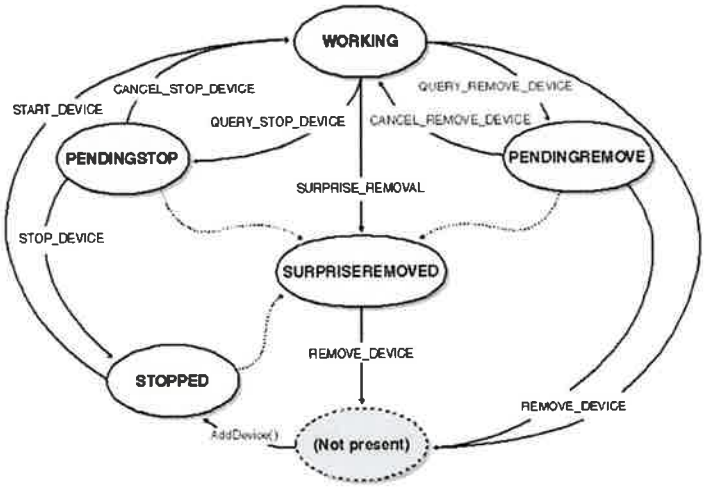


Figure 6-1. *State diagram for a device.*

This chapter also discusses PnP notifications, which provide a way for drivers and user-mode programs to learn asynchronously about the arrival and departure of devices. Properly handling these notifications is important for applications that work with devices that can be hot plugged and unplugged.

I've devoted a separate chapter (Chapter 11) to bus and multifunction drivers.

TIP
You can save yourself a lot of work by copying and using my GENERIC.SYS library. Instead of writing your own elaborate dispatch function for *IRP_MJ_PNP*, simply delegate this IRP to *GenericDispatchPnp*. See the Introduction for a table that lists the callback functions your driver supplies to perform device-specific operations. I've used the same callback function names in this chapter. In addition, I'm basically using GENERIC's PnP handling code for all of the examples.

6.1 *IRP_MJ_PNP* Dispatch Function

A simplified version of the dispatch function for *IRP_MJ_PNP* might look like the following:

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fdo, PIRP Irp)
{
1
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
2
  ULONG fcn = stack->MinorFunction;

3
  static NTSTATUS (*fcntab[])(PDEVICE_OBJECT, PIRP) = {
    HandleStartDevice,          // IRP_MN_START_DEVICE
    HandleQueryRemove,          // IRP_MN_QUERY_REMOVE_DEVICE
    <etc.>,
    };

4
  if (fcn >= arraysize(fcntab))
    return DefaultPnpHandler(fdo, Irp);
5
  return (*fcntab[fcn])(fdo, Irp);
  }

NTSTATUS DefaultPnpHandler(PDEVICE_OBJECT fdo, PIRP Irp)
{
6
  IoSkipCurrentIrpStackLocation(Irp);
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 182

```
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
}
```

1. All the parameters for the IRP, including the all-important minor function code, are in the stack location. Hence, we obtain a pointer to the stack location by calling *IoGetCurrentIrpStackLocation*.

2. We expect the IRP's minor function code to be one of those listed in Table 6 - 1.

3. A method of handling the two dozen possible minor function codes is to write a subdispatch function for each one we're going to handle and then to define a table of pointers to those subdispatch functions. Many of the entries in the table will be *DefaultPnpHandler*. Subdispatch functions such as *HandleStartDevice* will take pointers to a device object and an IRP as parameters and will return an *NTSTATUS* code.

4. If we get a minor function code we don't recognize, it's probably because Microsoft defined a new one in a release of the DDK after the DDK with which we built our driver. The right thing to do is to pass the minor function code down the stack by calling the default handler. By the way, *arraysize* is a macro in one of my own header files that returns the number of elements in an array. It's defined as #define *arraysize(p) (sizeof(p)/sizeof(p)[0])*.

5. This is the operative statement in the dispatch routine, in which we index the table of subdispatch functions and call the right one.

6. The *DefaultPnpHandler* routine is essentially the *ForwardAndForget* function I showed in connection with IPR-handling scenario 2 in the preceding chapter. We're passing the IRP down without a completion routine and therefore use *IoSkipCurrentIrpStackLocation* to retard the IRP stack pointer in anticipation that *IoCallDriver* will immediately advance it.

Using a Function Pointer Table
Using a table of function pointers to dispatch handlers for minor function codes as I'm showing you in *DispatchPnp* entails some slight danger. A future version of the operating system might change the meaning of some of the codes. That's not a practical worry except during the beta test phase of a system, though, because a later change would invalidate an unknown number of existing drivers. I like using a table of pointers to subdispatch functions because having separate functions for the minor function codes seems like the right engineering solution to me. If I were designing a C++ class library, for instance, I'd define a base class that used virtual functions for each of the minor function codes.

Most programmers would probably place a switch statement in their *DispatchPnp* routine. You can simply recompile your driver to conform to any reassignment of minor function codes. Recompilation will also highlight—by producing compilation errors!—name changes that might signal functionality shifts. That happened a time or two during the Microsoft Windows 98 and Windows 2000 betas, in fact. Furthermore, an optimizing compiler should be able to use a jump table to produce slightly faster code for a switch statement than for calls to subdispatch functions.

I think the choice between a switch statement and a table of function pointers is mostly a matter of taste, with readability and modularity winning over efficiency in my own evaluation. You can avoid uncertainty during a beta test by placing appropriate assertions in your code. For example, the *HandleStartDevice* function can assert that *stack->MinorFunction == IRP_MN_START_DEVICE*. If you recompile your driver with each new beta DDK, you'll catch any number reassignments or name changes.

6.2 Starting and Stopping Your Device

Working with the bus driver, the PnP Manager automatically detects hardware and assigns I/O resources in Windows XP and Windows 98/Me. Most modern devices have PnP features that allow system software to detect them automatically and to electronically determine which I/O resources they require. In the case of legacy devices that have no electronic means of identifying themselves to the operating system or of expressing their resource requirements, the registry database contains the information needed for the detection and assignment operations.

NOTE
I find it hard to give an abstract definition of the term *I/O resource* that isn't circular (for example, a resource used for I/O), so I'll give a concrete one instead. The WDM encompasses four standard I/O resource types: I/O ports, memory registers, direct memory access (DMA) channels, and interrupt requests.

When the PnP Manager detects hardware, it consults the registry to learn which filter drivers and function drivers will manage the hardware. As I discussed in Chapter 2, the PnP Manager loads these drivers (if necessary—one or more of them might already be present, having been called into memory on behalf of some other hardware) and calls their *AddDevice* functions. The *AddDevice* functions, in turn, create device objects and link them into a stack. At this point, the stage is set for the PnP Manager, working with all of the device drivers, to assign I/O resources.

The PnP Manager initially creates a list of resource requirements for each device and allows the drivers to filter that list. I'm going to ignore the filtering step for now because not every driver will need to participate in this step. Given a list of

IPR2024-00292
KASPERSKY EX 1008 Page 183

requirements. the PnP Manager can then assign resources so as to harmonize the potentially conflicting requirements of all the hardware present on the system. Figure 6-2 illustrates how the PnP Manager can arbitrate between two different devices that have overlapping requirements for an interrupt request number, for example.

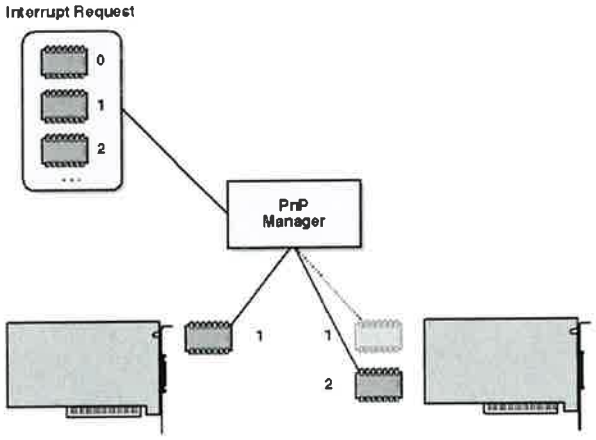


Figure 6-2. *Arbitration of conflicting I/O resource requirements.*

6.2.1 IRP_MN_START_DEVICE

Once the resource assignments are known, the PnP Manager notifies each device by sending it a PnP request with the minor function code *IRP_MN_START_DEVICE*. Filter drivers are typically not interested in this IRP, so they usually pass the request down the stack by using the *DefaultPnpHandler* technique I showed you earlier in "*IRP_MJ_PNP* Dispatch Function." Function drivers, on the other hand, need to do a great deal of work on the IRP to allocate and configure additional software resources and to prepare the device for operation. This work needs to be done, furthermore, at *PASSIVE_LEVELafter* the lower layers in the device hierarchy have processed this IRP.

You might implement *IRP_MN_START_DEVICE* in a subdispatch routine—reached from the *DispatchPnp* dispatch routine shown earlier—that has the following skeletal form:

```
NTSTATUS HandleStartDevice(PDEVICE OBJECT fdo, PIRP Irp)
  {
1
  Irp->IoStatus.Status = STATUS SUCCESS;
2
  NTSTATUS status = ForwardAndWait(fdo, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status);
3
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
4
  status = StartDevice(fdo, <additional args>);
5
  EnableAllInterfaces (pdx, True);
6
  return CompleteRequest(Irp, status);
  }
```

1. The bus driver uses the incoming setting of *IoStatus.Status* to determine whether upper-level drivers have handled this IRP. The bus driver makes a similar determination for several other minor functions of *IRP_MJ_PNP*. We therefore need to initialize the Status field of the IRP to *STATUS_SUCCESS* before passing it down.

2. *ForwardAndWait* is the function I showed you in Chapter 5 in connection with IRP-handling scenario 7 (synchronous pass down). The function returns a status code. If the status code denotes some sort of failure in the lower layers, we propagate the code back to our own caller. Because our completion routine returned *STATUS_MORE_PROCESSING_REQUIRED*, we halted the completion process inside *IoCompleteRequest*. Therefore, we have to complete the request all over again, as shown here.

3. Our configuration information is buried inside the stack parameters. I'll show you where a bit further on.

4. *StartDevice* is a helper routine you write to handle the details of extracting and dealing with configuration information. In my sample drivers, I've placed it in a separate source module named READWRITE.CPP. I'll explain shortly what

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 184

arguments you would pass to this routine besides the address of the device object.

5. *EnableAllInterfaces* enables all the device interfaces that you registered in your *AddDevice* routine. This step allows applications to find your device when they use *SetupDiXxx* functions to enumerate instances of your registered interfaces.

6. Since *ForwardAndWait* short-circuited the completion process for the *START_DEVICE* request, we need to complete the IRP a second time. In this example, I'm using an overloaded version of *CompleteRequest* that doesn't change *IoStatus.Information*, in accordance with the DDK rules for handling PnP requests.

You might guess (correctly!) that the *IRP_MN_START_DEVICE* handler has work to do that concerns the transition from the initial *STOPPED* state to the *WORKING* state. I can't explain that yet because I need to first explain the ramifications of other PnP requests on state transitions, IRP queuing, and IRP cancellation. So I'm going to concentrate for a while on the configuration aspects of the PnP requests.

The I/O stack location's Parameters union has a substructure named *StartDevice* that contains the configuration information you pass to the *StartDevice* helper function. See Table 6-2.

Field Name	Description
AllocatedResources	Contains raw resource assignments
AllocatedResourcesTranslated	Contains translated resource assignments

Table 6-2. *Fields in the Parameters.StartDevice Substructure of an I/O Stack Location*

Both *AllocatedResources* and *AllocatedResourcesTranslated* are instances of the same kind of data structure, called a *CM_RESOURCE_LIST*. This seems like a very complicated data structure if you judge only by its declaration in WDM.H. As used in a start device IRP, however, all that remains of the complication is a great deal of typing. The "lists" will have just one entry, a *CM_PARTIAL_RESOURCE_LIST* that describes all of the I/O resources assigned to the device. You can use statements like the following to access the two lists:

```
PCM_PARTIAL_RESOURCE_LIST raw, translated;
raw = &stack->Parameters.StartDevice
  .AllocatedResources->List[0].PartialResourceList;
translated = &stack->Parameters.StartDevice
  .AllocatedResourcesTranslated->List[0].PartialResourceList;
```

The only difference between the last two statements is the reference to either the *AllocatedResources* or *AllocatedResourcesTranslated* member of the parameters structure.

The raw and translated resource lists are the logical arguments to send to the StartDevice helper function, by the way:

```
status = StartDevice(fdo, raw, translated);
```

There are two different lists of resources because I/O buses and the CPU can address the same physical hardware in different ways. The raw resources contain numbers that are bus-relative, whereas the translated resources contain numbers that are system-relative. Prior to the WDM, a kernel-mode driver might expect to retrieve raw resource values from the registry, the Peripheral Component Interconnect (PCI) configuration space, or some other source, and to translate them by calling routines such as *HalTranslateBusAddress* and *HalGetInterruptVector*. See, for example, Art Baker's The Windows NT Device Driver Book: *A Guide for Programmers* (Prentice Hall, 1997), pages 122-62. Both the retrieval and translation steps are done by the PnP Manager now, and all a WDM driver needs to do is access the parameters of a start device IRP as I'm now describing.

What you actually do with the resource descriptions inside your *StartDevice* function is a subject for Chapter 7.

6.2.2 IRP_MN_STOP_DEVICE

The stop device request tells you to shut your device down so that the PnP Manager can reassign I/O resources. At the hardware level, shutting down involves pausing or halting current activity and preventing further interrupts. At the software level, it involves releasing the I/O resources you configured at start device time. Within the framework of the dispatch/subdispatch architecture I've been illustrating, you might have a subdispatch function like this one:

```
NTSTATUS HandleStopDevice(PDEVICE_OBJECT fdo, PIRP Irp)
{
1
  <complicated stuff>
2
  StopDevice(fdo, oktouch);
  Irp->IoStatus.Status = STATUS_SUCCESS;
3
  return DefaultPnpHandler(fdo, Irp);
}
```


IPR2024-00292
KASPERSKY EX 1008 Page 185

1. Right about here, you need to insert some more or less complicated code that concerns IRP queuing and cancellation. I'll show you the code that belongs in this spot further on in this chapter in "While the Device Is Stopped."

2. In contrast with the start device case, in which we passed the request down and then did device-dependent work, here we do our device-dependent stuff first and then pass the request down. The idea is that our hardware will be quiescent by the time the lower layers see this request. I wrote a helper function named *StopDevice* to do the shutdown work. The second argument indicates whether it will be OK for *StopDevice* to touch the hardware if it needs to. Refer to the sidebar "Touching the Hardware When Stopping the Device" for an explanation of how to set this argument.

3. We always pass PnP requests down the stack. In this case, we don't care what the lower layers do with the request, so we can simply use the *DefaultPnpHandler* code to perform the mechanics.

The *StopDevice* helper function called in the preceding example is code you write that essentially reverses the configuration steps you took in *StartDevice*. I'll show you that function in the next chapter. One important fact about the function is that you should code it in such a way that it can be called more than once for a single call to *StartDevice*. It's not always easy for a PnP IRP handler to know whether you've already called *StopDevice*, but it is easy to make *StopDevice* proof against duplicative calls.

Touching the Hardware When Stopping the Device

In the skeleton of *HandleStopDevice*, I used an *oktouch* variable that I didn't show you how to initialize. In the scheme I'm teaching you in this book for writing a driver, the *StopDevice* function gets a *BOOLEAN* argument that indicates whether it should be safe to address actual I/O operations to the hardware. The idea behind this argument is that you might want to send certain instructions to your device as part of your shutdown protocol, but there might be some reason why you can't. You might want to tell your Personal Computer Memory Card International Association (PCMCIA) modem to hang up the phone, for example, but there's no point in trying if the end user has already removed the modem card from the computer.

There's no certain way to know whether your hardware is physically connected to the computer except by trying to access it. Microsoft recommends, however, that if you succeeded in processing a *START_DEVICE* request, you should go ahead and try to access your hardware when you process *STOP_DEVICE* and certain other PnP requests. When I discuss how you track PnP state changes later in this chapter, I'll honor this recommendation by setting the *oktouch* argument to *TRUE* if we believe that the device is currently working and FALSE otherwise.

6.2.3 IRP_MN_REMOVE_DEVICE

Recall that the PnP Manager calls the *AddDevice* function in your driver to notify you about an instance of the hardware you manage and to give you an opportunity to create a device object. Instead of calling a function to do the complementary operation, however, the PnP Manager sends you a PnP IRP with the minor function code *IRP_MN_REMOVE_DEVICE*. In response to that, you'll do the same things you did for *IRP_MN_STOP_DEVICE* to shut down your device, and then you'll delete the device object:

```
NTSTATUS HandleRemoveDevice(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  <complicated stuff>
  DeregisterAllInterfaces(pdx);
  StopDevice(fdo, oktouch);
  Irp->IoStatus.Status = STATUS_SUCCESS;
  NTSTATUS status = DefaultPnpHandler(fdo, Irp);
  RemoveDevice(fdo);
  return status;
  }
```

This fragment looks similar to *HandleStopDevice*, with a couple of additions. *DeregisterAllInterfaces* will disable any device interfaces you registered (probably in *AddDevice*) and enabled (probably in *StartDevice*), and it will release the memory occupied by their symbolic link names. *RemoveDevice* will undo all the work you did inside *AddDevice*. For example:

```
VOID RemoveDevice(PDEVICE_OBJECT fdo)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
1
  IoDetachDevice(pdx->LowerDeviceObject);
2
  IoDeleteDevice(fdo);
  }
```

1. This call to *IoDetachDevice* balances the call *AddDevice* made to *IoAttachDeviceToDeviceStack*.

IPR2024-00292
KASPERSKY EX 1008 Page 186

2. This call to *IoDeleteDevice* balances the call *AddDevice* made to *IoCreateDevice*. Once this function returns, you should act as if the device object no longer exists. If your driver isn't managing any other devices, it will shortly be unloaded from memory too.

Note, by the way, that you don't get a stop device request followed by a remove device request. The remove device request implies a shutdown, so you do both pieces of work in reply.

6.2.4 IRP_MN_SURPRISE_REMOVAL

Sometimes the end user has the physical ability to remove a device without going through any user interface elements first. If the system detects that such a surprise removal has occurred, or that the device appears to be broken, it sends the driver a PnP request with the minor function code *IRP_MN_SURPRISE_REMOVAL*. It will later send an *IRP_MN_REMOVE_DEVICE*. Unless you previously set the *SurpriseRemovalOK* flag while processing *IRP_MN_QUERY_CAPABILITIES* (as I'll discuss in Chapter 8), some platforms also post a dialog box to inform the user that it's potentially dangerous to yank hardware out of the computer.

In response to the surprise removal request, a device driver should disable any registered interfaces. This will give applications a chance to close handles to your device if they're on the lookout for the notifications I discuss later in "PnP Notifications." Then the driver should release I/O resources and pass the request down:

```
NTSTATUS HandleSurpriseRemoval(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  <complicated stuff>
  EnableAllInterfaces(pdx, FALSE);
  StopDevice(fdo, oktouch);
  Irp->IoStatus.Status = STATUS SUCCESS;
  return DefaultPnpHandler(fdo, Irp);
  }
```

Whence IRP_MN_SURPRISE_REMOVAL?
The surprise removal PnP notification doesn't happen as a simple and direct result of the end user yanking the device from the computer. Some bus drivers can know when a device disappears. For example, removing a universal serial bus (USB) device generates an electronic signal that the bus driver notices. For many other buses, however, there isn't any signal to alert the bus driver. The PnP Manager therefore relies on other methods to decide that a device has disappeared.

A function driver can signal the disappearance of its device (if it knows) by calling IoInvalidateDeviceState and then returning any of the values PNP_DEVICE_FAILED, PNP_DEVICE_REMOVED, or PNP_DEVICE_DISABLED from the ensuing IRP_MN_QUERY_PNP_DEVICE_STATE. You might want to do this in your own driver if—to give one example of many—your interrupt service routines (ISRs) read all 1 bits from a status port that normally returns a mixture of 1s and 0s. More commonly, a bus driver calls IoInvalidateDeviceRelations to trigger a re-enumeration and then fails to report the newly missing device. It's worth knowing that when the end user removes a device while the system is hibernating or in another low-power state, when power is restored, the driver receives a series of power management IRPs before it receives the IRP_MN_SURPRISE_REMOVAL request.

What these facts mean, practically speaking, is that your driver should be able to cope with errors that might arise from having your device suddenly not present.

6.3 Managing PnP State Transitions

As I said at the outset of this chapter, WDM drivers need to track their devices through the state transitions diagrammed in Figure 6-1. This state tracking also ties in with how you queue and cancel I/O requests. Cancellation in turn implicates the global cancel spin lock, which is a performance bottleneck in a multi-CPU system. The standard model of IRP processing with Microsoft queuing functions can't solve all these interrelated problems. In this section, therefore, I'll describe how my *DEVQUEUE* object helps you cope with the complications Plug and Play creates.

Figure 6-3 illustrates the states of a *DEVQUEUE*. In the *READY* state, the queue accepts and forwards requests to your *StartIo* routine in such a way that the device stays busy. In the *STALLED* state, however, the queue doesn't forward IRPs to *StartIo*, even when the device is idle. In the *REJECTING* state, the queue doesn't even accept new IRPs. Figure 6-4 illustrates the flow of IRPs through the queue.

IPR2024-00292
KASPERSKY EX 1008 Page 187

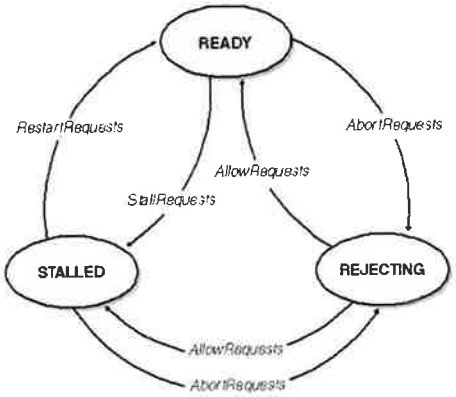


Figure 6-3. *States of a DEVQUEUE object.*

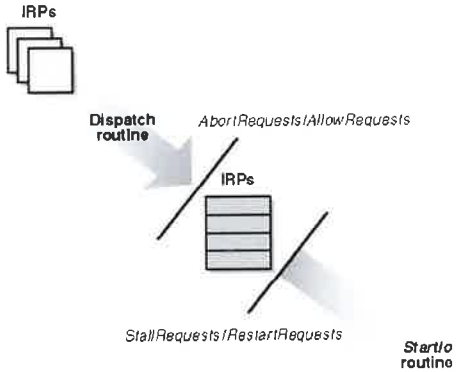


Figure 6-4. *Flow of IRPs through a DEVQUEUE.*

Table 6-3 lists the support functions you can use with a *DEVQUEUE*. I discussed how to use *InitializeQueue*, *StartPacket*, *StartNextPacket*, and *CancelRequest* in the preceding chapter. Now it's time to discuss all the other functions.

Support Function	Description
AbortRequests	Aborts current and future requests
AllowRequests	Undoes effect of previous *AbortRequests*
AreRequestsBeingAborted	Are we currently aborting new requests?
CancelRequest	Generic cancel routine
CheckBusyAndStall	Checks for idle device and stalls requests in one atomic operation
CleanupRequests	Cancels all requests for a given file object in order to service *IRP_MJ_CLEANUP*
GetCurrentIrp	Determines which IRP is currently being processed by associated *StartIo* routine
InitializeQueue	Initializes *DEVQUEUE* object
RestartRequests	Restarts a stalled queue
StallRequests	Stalls the queue
StartNextPacket	Dequeues and starts the next request
StartPacket	Starts or queues a new request
WaitForCurrentIrp	Waits for current IRP to finish

Table 6-3. *DEVQUEUE Service Routines*

The real point of using a *DEVQUEUE* instead of one of the queue objects defined in the DDK is that a *DEVQUEUE* makes it easier to manage the transitions between PnP states. In all of my sample drivers, the device extension contains a state variable with the imaginative name *state*. I also define an enumeration named *DEVSTATE* whose values correspond to the PnP states. When you initialize your device object in *AddDevice*, you'll call *InitializeQueue* for each of your device queues and also indicate that the device is in the *STOPPED* state:

```
NTSTATUS AddDevice(...)
  {
```


IPR2024-00292
KASPERSKY EX 1008 Page 188

```
  :
  PDEVICE_EXTENSION pdx = ...;
  InitializeQueue(&pdx->dqReadWrite, StartIo);
  pdx->state = STOPPED;
  :
  }
```

After *AddDevice* returns, the system sends *IRP_MJ_PNP* requests to direct you through the various PnP states the device can assume.

NOTE
If your driver uses GENERIC.SYS, GENERIC will initialize your DEVQUEUE object or objects for you. Just be sure to give GENERIC the addresses of those objects in your call to InitializeGenericExtension.

6.3.1 Starting the Device

A newly initialized *DEVQUEUE* is in a *STALLED* state, such that a call to *StartPacket* will queue a request even when the device is idle. You'll keep the queue (or queues) in the *STALLED* state until you successfully process *IRP_MN_START_DEVICE*, whereupon you'll execute code like the following:

```
NTSTATUS HandleStartDevice(...)
  {
  status = StartDevice(...);
  if (NT_SUCCESS(status))
    {
    pdx->state = WORKING;
    RestartRequests(&pdx->dqReadWrite, fdo);
    }
  }
```

You record *WORKING* as the current state of your device, and you call *RestartRequests* for each of your queues to release any IRPs that might have arrived between the time *AddDevice* ran and the time you received the *IRP_MN_START_DEVICE* request.

6.3.2 Is It OK to Stop the Device?

The PnP Manager always asks your permission before sending you an *IRP_MN_STOP_DEVICE*. The query takes the form of an *IRP_MN_QUERY_STOP_DEVICE* request that you can cause to succeed or fail as you choose. The query basically means, "Would you be able to immediately stop your device if the system were to send you an *IRP_MN_STOP_DEVICE* in a few nanoseconds?" You can handle this query in two slightly different ways. Here's the first way, which is appropriate when your device might be busy with an IRP that either finishes quickly or can be easily terminated in the middle:

```
NTSTATUS HandleQueryStop(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS_SUCCESS;
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
1 if (pdx->state != WORKING)
    return DefaultPnpHandler(fdo, Irp);
2 if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
3 StallRequests(&pdx->dqReadWrite);
  WaitForCurrentIrp(&pdx->dqReadWrite);
4 pdx->state = PENDINGSTOP;
  return DefaultPnpHandler(fdo, Irp);
  }
```

1. This statement handles a peculiar situation that can arise for a boot device: the PnP Manager might send you a *QUERY_STOP* when you haven't initialized yet. You want to ignore such a query, which is tantamount to saying yes.
2. At this point, you perform some sort of investigation to see whether it will be OK to revert to the *STOPPED* state. I'll discuss factors bearing on the investigation next.
3. *StallRequests* puts the *DEVQUEUE* in the *STALLED* state so that any new IRP just goes into the queue. *WaitForCurrentIrp* waits until the current request, if there is one, finishes on the device. These two steps make the device

IPR2024-00292
KASPERSKY EX 1008 Page 189

quiescent until we know whether the device is really going to stop or not. If the current IRP won't finish quickly of its own accord, you'll do something (such as calling *IoCancelIrp* to force a lower-level driver to finish the current IRP) to "encourage" it to finish; otherwise, *WaitForCurrentIrp* won't return.

4. At this point, we have no reason to demur. We therefore record our state as *PENDINGSTOP*. Then we pass the request down the stack so that other drivers can have a chance to accept or decline this query.

The other basic way of handling *QUERY_STOP* is appropriate when your device might be busy with a request that will take a long time and can't be stopped in the middle, such as a tape retension operation that can't be stopped without potentially breaking the tape. In this case, you can use the *DEVQUEUE* object's *CheckBusyAndStall* function. That function returns *TRUE* if the device is busy, whereupon you cause the *QUERY_STOP* to fail with *STATUS_UNSUCCESSFUL*. The function returns *FALSE* if the device is idle, in which case it also stalls the queue. (The operations of checking the state of the device and stalling the queue need to be protected by a spin lock, which is why I wrote this function in the first place.)

You can cause a stop query to fail for many reasons. Disk devices that are used for paging, for example, cannot be stopped. Neither can devices that are used for storing hibernation or crash dump files. (You'll know about these characteristics as a result of an *IRP_MN_DEVICE_USAGE_NOTIFICATION* request, which I'll discuss later in "Other Configuration Functionality.") Other reasons may also apply to your device.

Even if you have the query succeed, one of the drivers underneath you might cause it to fail for some reason. Even if all the drivers have the query succeed, the PnP Manager might decide not to shut you down. In any of these cases, you'll receive another PnP request with the minor code *IRP_MN_CANCEL_STOP_DEVICE* to tell you that your device won't be shut down. You should then clear whatever state you set during the initial query:

```
NTSTATUS HandleCancelStop(PDEVICE OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  if (pdx->state != PENDINGSTOP)
    return DefaultPnpHandler(fdo, Irp);
  NTSTATUS status = ForwardAndWait(fdo, Irp);
  pdx->state = WORKING;
  RestartRequests(&pdx->dqReadWrite, fdo);
  return CompleteRequest(Irp, status);
  }
```

We first check to see whether a stop operation is even pending. Some higher-level driver might have vetoed a query that we never saw, so we'd still be in the *WORKING* state. If we're not in the *PENDINGSTOP* state, we simply forward the IRP. Otherwise, we send the *CANCEL_STOP IRP* synchronously to the lower-level drivers. That is, we use our *ForwardAndWait* helper function to send the IRP down the stack and await its completion. We wait for low-level drivers because we're about to resume processing IRPs, and the drivers might have work to do before we send them an IRP. We then change our state variable to indicate that we're back in the WORKING state, and we call *RestartRequests* to unstall the queues we stalled when we caused the query to succeed.

6.3.3 While the Device Is Stopped

If, on the other hand, all device drivers have the query succeed and the PnP Manager decides to go ahead with the shutdown, you'll get an IRP_MN_STOP_DEVICE next. Your subdispatch function will look like this one:

```
NTSTATUS HandleStopDevice(PDEVICE OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
  if (pdx->state != PENDINGSTOP);
    {
    <complicated stuff>
    }
2
  StopDevice(fdo, pdx->state == WORKING);
3
  pdx->state = STOPPED;
4
  return DefaultPnpHandler(fdo, Irp);
  }
```

1. We expect the system to send us a *QUERY_STOP* before it sends us a *STOP*, so we should already be in the *PENDINGSTOP* state with all of our queues stalled. There is, however, a bug in Windows 98 such that we can sometimes get a *STOP* (without a *QUERY_STOP*) instead of a *REMOVE*. You need to take some action at this point that causes you

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 190

to reject any new IRPs, but you mustn't really remove your device object or do the other things you do when you really receive a *REMOVE* request.

2. *StopDevice* is the helper function I've already discussed that deconfigures the device.

3. We now enter the *STOPPED* state. We're in almost the same situation as we were when *AddDevice* was done. That is, all queues are stalled, and the device has no I/O resources. The only difference is that we've left our registered interfaces enabled, which means that applications won't have received removal notifications and will leave their handles open. Applications can also open new handles in this situation. Both aspects are just as they should be because the stop condition won't last long.

4. As I previously discussed, the last thing we do to handle *IRP_MN_STOP_DEVICE* is pass the request down to the lower layers of the driver hierarchy.

6.3.4 Is It OK to Remove the Device?

Just as the PnP Manager asks your permission before shutting your device down with a stop device request, it also might ask your permission before removing your device. This query takes the form of an *IRP_MN_QUERY_REMOVE_DEVICE* request that you can, once again, cause to succeed or fail as you choose. And, just as with the stop query, the PnP Manager will use an *IRP_MN_CANCEL_REMOVE_DEVICE* request if it changes its mind about removing the device.

```
NTSTATUS HandleQueryRemove(PDEVICE OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
  if (OkayToRemove(fdo))
    {
2
    StallRequests(&pdx->dqReadWrite);
    WaitForCurrentIrp(&pdx->dqReadWrite);
3
    pdx->prevstate = pdx->state;
    pdx->state = PENDINGREMOVE;
    return DefaultPnpHandler(fdo, Irp);
    }
  return CompleteRequest(Irp, STATUS UNSUCCESSFUL, 0);
  }

NTSTATUS HandleCancelRemove(PDEVICE OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
4
  if (pdx->state != PENDINGREMOVE)
    return DefaultPnpHandler(fdo, Irp);
  NTSTATUS status = ForwardAndWait(fdo, Irp);
5
  pdx->state = pdx->prevstate;
  RestartRequests(&pdx->dqReadWrite, fdo);
  return CompleteRequest(Irp, status);
  }
```

1. This *OkayToRemove* helper function provides the answer to the question, "Is it OK to remove this device?" In general, this answer includes some device-specific ingredients, such as whether the device holds a paging or hibernation file, and so on.

2. Just as I showed you for *IRP_MN_QUERY_STOP_DEVICE*, you want to stall the request queue and wait for a short period, if necessary, until the current request finishes.

3. If you look at Figure 6-1 carefully, you'll notice that it's possible to get a *QUERY_REMOVE* when you're in either the *WORKING* or the *STOPPED* state. The right thing to do if the current query is later cancelled is to return to the original state. Hence, I have a prevstate variable in the device extension to record the prequery state.

4. We get the CANCEL_REMOVE request when someone either above or below us vetoes a *QUERY_REMOVE*. If we never saw the query, we'll still be in the *WORKING* state and don't need to do anything with this IRP. Otherwise, we need to forward it to the lower levels before we process it because we want the lower levels to be ready to process the IRPs we're about to release from our queues.

5. Here we undo the steps we took when we succeeded the *QUERY_REMOVE*. We revert to the previous state. We stalled the queues when we handled the query and need to unstall them now.

IPR2024-00292
KASPERSKY EX 1008 Page 191

6.3.5 Synchronizing Removal

It turns out that the I/O Manager can send you PnP requests simultaneously with other substantive I/O requests, such as requests that involve reading or writing. It's entirely possible, therefore, for you to receive an *IRP_MN_REMOVE_DEVICE* at a time when you're still processing another IRP. It's up to you to prevent untoward consequences, and the standard way to do that involves using an *IO_REMOVE_LOCK* object and several associated kernel-mode support routines.

The basic idea behind the standard scheme for preventing premature removal is that you acquire the remove lock each time you start processing a request that you will pass down the PnP stack, and you release the lock when you're done. Before you remove your device object, you make sure that the lock is free. If not, you wait until all references to the lock are released. Figure 6-5 illustrates the process.

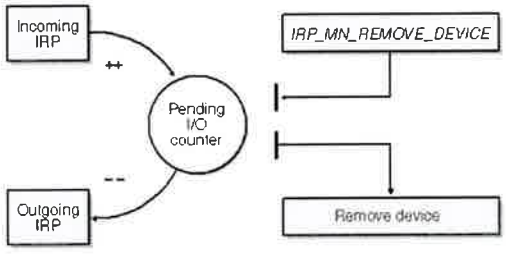


Figure 6-5. *Operation of an IO_REMOVE_LOCK.*

To handle the mechanics of this process, you define a variable in the device extension:

```
struct DEVICE EXTENSION {
  :
  IO REMOVE LOCK RemoveLock;
  :
  };
```

You initialize the lock object during AddDevice:

```
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
  :
  IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 0);
  :
  }
```

The last three parameters to *IoInitializeRemoveLock* are, respectively, a tag value, an expected maximum lifetime for a lock, and a maximum lock count, none of which is used in the free build of the operating system.

These preliminaries set the stage for what you do during the lifetime of the device object. Whenever you receive an I/O request that you plan to forward down the stack, you call *IoAcquireRemoveLock*. *IoAcquireRemoveLock* will return *STATUS_DELETE_PENDING* if a removal operation is under way. Otherwise, it will acquire the lock and return *STATUS_SUCCESS*. Whenever you finish such an I/O operation, you call *IoReleaseRemoveLock*, which will release the lock and might unleash a heretofore pending removal operation. In the context of some purely hypothetical dispatch function that synchronously forwards an IRP, the code might look like this:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
  status = ForwardAndWait(fdo, Irp);
  if (!NT SUCCESS(status))
    {
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return CompleteRequest(Irp, status, 0);
    }
  :
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return CompleteRequest(Irp, <some code>, <info value>);
  }
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 192

The second argument to *IoAcquireRemoveLock* and *IoReleaseRemoveLock* is just a tag value that a checked build of the operating system can use to match up acquisition and release calls, by the way.

The calls to acquire and release the remove lock dovetail with additional logic in the PnP dispatch function and the remove device subdispatch function. First *DispatchPnp* has to obey the rule about locking and unlocking the device, so it will contain the following code, which I didn't show you earlier in "*IRP_MJ_PNP* Dispatch Function":

```
NTSTATUS DispatchPnp(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
  .
  .
  .
  status = (*fcntab[fcn](fdo, Irp);
  if (fcn != IRP MN REMOVE DEVICE)
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
  }
```

In other words, *DispatchPnp* locks the device, calls the subdispatch routine, and then (usually) unlocks the device afterward. The subdispatch routine for *IRP_MN_REMOVE_DEVICE* has additional special logic that you also haven't seen yet:

```
NTSTATUS HandleRemoveDevice(PDEVICE OBJECT fdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
  AbortRequests(&pdx->dqReadWrite, STATUS DELETE PENDING);
  DeregisterAllInterfaces(pdx);
  StopDevice(fdo, pdx->state == WORKING);
  pdx->state = REMOVED;
2
  NTSTATUS status = DefaultPnpHandler(pdx->LowerDeviceObject, Irp);
3
  IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
  RemoveDevice(fdo);
  return status;
  }
```

1. Windows 98/Me doesn't send the *SURPRISE_REMOVAL* request, so this *REMOVE* IRP may be the first indication you have that the device has disappeared. Calling *StopDevice* allows you to release all your I/O resources in case you didn't get an earlier IRP that caused you to release them. Calling *AbortRequests* causes you to complete any queued IRPs and to start rejecting any new IRPs.

2. We pass this request to the lower layers now that we've done our work.

3. The PnP dispatch routine acquired the remove lock. We now call the special function *IoReleaseRemoveLockAndWait* to release that lock reference and wait until all references to the lock are released. Once you call *IoReleaseRemoveLockAndWait*, any subsequent call to *IoAcquireRemoveLock* will elicit a *STATUS_DELETE_PENDING* status to indicate that device removal is under way.

NOTE
You'll notice that the *IRP_MN_REMOVE_DEVICE* handler might block while an IRP finishes. This is certainly OK in Windows 98/Me and Windows XP, which were designed with this possibility in mind—the IRP gets sent in the context of a system thread that's allowed to block. Some WDM functionality (a Microsoft developer even called it "embryonic") is present in OEM releases of Microsoft Windows 95, but you can't block a remove device request there. Consequently, if your driver needs to run in Windows 95, you need to discover that fact and avoid blocking. That discovery process is left as an exercise for you.

It bears repeating that you need to use the remove lock only for an IRP that you pass down the PnP stack. If you have the stamina, you can read the next section to understand exactly why this conclusion is true—and note that it differs from the conventional wisdom that I and others have been espousing for several years. If someone sends you an IRP that you handle entirely inside your own driver, you can rely on whoever sent you the IRP to make sure your driver remains in memory until you both complete the IRP and return from your dispatch routine. If you send an IRP to someone outside your PnP stack, you'll use other means (such as a referenced file or device object) to keep the target driver in memory until it both completes the IRP and returns from its dispatch routine.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 193

6.3.6 Why Do I Need This @#$! Remove Lock, Anyway?

A natural question at this point is why, in the context of a robust and full-featured modern operating system, you even need to worry about somebody unloading a driver when it knows, or should know, that it's busy handling an IRP. This question is hard to answer, but here goes.



The remove lock isn't necessary to guard against having your device object removed out from under you while you're processing an IRP. Rather, it protects you from sending an IRP down your PnP stack to a lower device object that no longer exists or that might cease to exist before the IRP finishes. To make this clear, I need to explain rather fully how the PnP Manager and the Object Manager work together to keep drivers and device objects around while they're needed. I'm grossly oversimplifying here in order to emphasize the basic things you need to understand.

First of all, every object that the Object Manager manages carries a reference count. When someone creates such an object, the Object Manager initializes the reference count to 1. Thereafter, anyone can call *ObReferenceObject* to increment the reference count and *ObDereferenceObject* to decrement it. For each type of object, there is a routine that you can call to destroy the object. For example, *IoDeleteDevice* is the routine you call to delete a *DEVICE_OBJECT*. That routine never directly releases the memory occupied by the object. Instead, it directly or indirectly calls *ObDereferenceObject* to release the original reference. Only when the reference count drops to 0 will the Object Manager actually destroy the object.

NOTE
In Chapter 5, I advised you to take an extra reference to a file object or device object discovered via *IoGetDeviceObjectPointer* around the call to *IoCallDriver* for an asynchronous IRP. The reason for the advice may now be clear: you want to be sure the target driver for the IRP is pinned in memory until its dispatch routine returns regardless of whether your completion routine releases the reference taken by IoGetDeviceObjectPointer. Dang, but this is getting complicated!

IoDeleteDevice makes some checks before it releases the last reference to a device object. In both operating systems, it checks whether the *AttachedDevice* pointer is *NULL*. This field in the device object points upward to the device object for the next upward driver. This field is set by *IoAttachDeviceToDeviceStack* and reset by *IoDetachDevice*, which are functions that WDM drivers call in their *AddDevice* and *RemoveDevice* functions, respectively.

You want to think about the entire PnP stack of device objects as being the target of IRPs that the I/O Manager and drivers outside the stack send to "your" device. This is because the driver for the topmost device object in the stack is always first to process any IRP. Before anyone sends an IRP to your stack, however, they will have a referenced pointer to this topmost device object, and they won't release the reference until after the IRP completes. So if a driver stack contains just one device object, there will never be any danger of having a device object or driver code disappear while the driver is processing an IRP: the IRP sender's reference pins the device object in memory, even if someone calls *IoDeleteDevice* before the IRP completes, and the device object pins the driver code in memory.

WDM driver stacks usually contain two or more device objects, so you have to wonder about the second and lower objects in a stack. After all, whoever sends an IRP to the device has a reference only to the topmost device object, not to the objects lower down in the stack. Imagine the following scenario, then. Someone sends an *IRP_MJ_SOMETHING* (a made-up major function to keep us focused on the remove lock) to the topmost filter device object (FiDO), whose driver sends it down the stack to your function driver. You plan to send this IRP down to the filter driver underneath you. But, at about the same time on another CPU, the PnP Manager has sent your driver stack an *IRP_MN_REMOVE_DEVICE* request.

Before the PnP Manager sends *REMOVE_DEVICE* requests, it takes an extra reference to every device object in the stack. Then it sends the IRP. Each driver passes the IRP down the stack and then calls *IoDetachDevice* followed by *IoDeleteDevice*. At each level, *IoDeleteDevice* sees that *AttachedDevice* is not (yet) *NULL* and decides that the time isn't quite right to dereference the device object. When the driver at the next higher level calls *IoDetachDevice*, however, the time is right, and the I/O Manager dereferences the device object. Without the PnP Manager's extra reference, the object would then disappear, and that might trigger unloading the driver at that level of the stack. Once the *REMOVE_DEVICE* request is complete, the PnP Manager will release all the extra references. That will allow all but the topmost device object to disappear because only the topmost object is protected by the reference owned by the sender of the *IRP_MJ_SOMETHING*.

IMPORTANT
Every driver I've ever seen or written processes *REMOVE_DEVICE* synchronously. That is, no driver ever pends a *REMOVE_DEVICE* request. Consequently, the calls to *IoDetachDevice* and *IoDeleteDevice* at any level of the PnP stack always happen after the lower-level drivers have already performed those calls. This fact doesn't impact our analysis of the remove lock because the PnP Manager won't release its extra reference to the stack until after *REMOVE_DEVICE* actually completes, which requires *IoCompleteRequest* to run to conclusion.

Can you see why the Microsoft folks who understand the PnP Manager deeply are fond of saying, "Game Over" at this point? We're going to trust whoever is above us in the PnP stack to keep our device object and driver code in memory until we're done handling the *IRP_MJ_SOMETHING* that I hypothesized. But we haven't (yet) done anything to keep the next lower device object and driver in memory. While we were getting ready to send the IRP down, the *IRP_MN_REMOVE_DEVICE* ran to completion, and the lower driver is now gone!

And that's the problem that the remove lock solves: we simply don't want to pass an IRP down the stack if we've already

IPR2024-00292
KASPERSKY EX 1008 Page 194

returned from handling an *IRP_MN_REMOVE_DEVICE*. Conversely, we don't want to return from *IRP_MN_REMOVE_DEVICE* (and thereby allow the PnP Manager to release what might be the last reference to the lower device object) until we know the lower driver is done with all the IRPs that we've sent to it.

Armed with this understanding, let's look again at an IRP-handling scenario in which the remove lock is helpful. This is an example of my IRP-handling scenario 1 (pass down with completion routine) from Chapter 5:

```
NTSTATUS DispatchSomething(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
A NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp,
    (PIO COMPLETION ROUTINE) CompletionRoutine, pdx, TRUE, TRUE, TRUE);
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }

NTSTATUS CompletionRoutine(PDEVICE OBJECT fdo, PIRP Irp, PDEVICE EXTENSION pdx)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
  <desired completion processing>
B IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return STATUS SUCCESS;
  }
```

In summary, we acquire the remove lock for this IRP in the dispatch routine, and we release it in the completion routine. Suppose this IRP is racing an *IRP_MN_REMOVE_DEVICE* down the stack. If our *HandleRemoveDevice* function has gotten to the point of calling *IoReleaseRemoveLockAndWait* before we get to point A, perhaps all the device objects in the stack are teetering on the edge of extinction because the *REMOVE_DEVICE* may have finished long ago. If we're the topmost device object, somebody's reference is keeping us alive. If we're lower down the stack, the driver above us is keeping us alive. Either way, it's certainly OK for us to execute instructions. We'll find that our call to *IoAcquireRemoveLock* returns *STATUS_DELETE_PENDING*, so we'll just complete the IRP and return.

Suppose instead that we win the race by calling *IoAcquireRemoveLock* before our *HandleRemoveDevice* function calls *IoReleaseRemoveLockAndWait*. In this case, we'll pass the IRP down the stack. *IoReleaseRemoveLockAndWait* will block until our completion routine (at point B) releases the lock. At this exact instant, we fall back on the IRP sender's reference or the driver above us to keep us in memory long enough for our completion routine to return.

At this point in the analysis, I have to raise an alarming point that everyone who writes WDM drivers or writes or lectures about them, including me, has missed until now. Passing an IRP down without a completion routine is actually unsafe because it allows us to send an IRP down to a driver that isn't pinned in memory. Anytime you see a call to *IoSkipCurrentIrpStackLocation* (there are 204 of them in the Windows XP DDK), your antennae should twitch. We've all been getting away with this because some redundant protections are in place and because the coincidence of an *IRP_MN_REMOVE_DEVICE* with some kind of problem IRP is very rare. Refer to the sidebar for a discussion.

The Redundant Guards Against Early Removal

As the text says, Windows XP contains some redundant protections against early removal of device objects. In both Windows XP and Windows 2000, the PnP Manager won't send an *IRP_MN_REMOVE_DEVICE* if any file objects exist that point to any device object in the stack. Many IRPs are handle based in that they originate in callers that hold a referenced pointer to a file object. Consequently, there is never a concern with these handle-based IRPs that your lower device object might disappear. You can dispense with the remove lock altogether for these IRPs if you trust all the drivers who send them to you to either have a referenced file object or hold their own remove lock while they're outstanding.

There is a large class of IRP that device drivers never see because these IRPs involve file system operations on volumes. Thus, worrying about what might happen as a device driver handles an *IRP_MJ_QUERY_VOLUME_INFORMATION*, for example, isn't practical.

Only a few IRPs aren't handle based or aimed at file system drivers, and most of them carry their own built-in safeguards. To get an *IRP_MJ_SHUTDOWN*, you have to specifically register with the I/O Manager by calling *IoRegisterShutdownNotification*. *IoDeleteDevice* automatically deregisters you if you happen to forget, and you won't be getting *REMOVE_DEVICE* requests while shutdown notifications are in progress. (While we're on the subject, note these additional details about *IRP_MJ_SHUTDOWN*. Like every other IRP, this one will be sent first to the topmost FiDO in the PnP stack if *any* driver in the stack has called *IoRegisterShutdownNotification*. Furthermore, as many IRPs will be sent as there are drivers in the stack with active notification requests. Thus, drivers should take care to do their shutdown processing only once and should pass this IRP down the stack *after* doing their own shutdown processing.)

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 195

IRP_MJ_SYSTEM_CONTROL is another special case. The Windows Management Instrumentation (WMI) subsystem uses this request to perform WMI query and set operations. Part of your *StopDevice* processing ought to be deregistering with WMI, and the deregistration call doesn't return until all of these IRPs have drained through your device. After the deregistration call, you won't get any more WMI requests.

The PnP Manager itself is the source of most *IRP_MJ_PNP* requests, and you can be sure that it won't overlap a *REMOVE_DEVICE* request with another PnP IRP. You can't, however, be sure there's no overlap with PnP IRPs sent by other drivers, such as a *QUERY_DEVICE_RELATIONS* to get the physical device object (PDO) address or a *QUERY_INTERFACE* to locate a direct-call interface.

Finally, there's *IRP_MJ_POWER*, which is a potential problem because the Power Manager doesn't lock an entire device stack and doesn't hold a file object pointer.

The window of vulnerability is actually pretty small. Consider the following fragment of dispatch routines in two drivers:

```
NTSTATUS DriverA DispatchSomething(...)
  {
  :
  NTSTATUS status = IoAcquireRemoveLock(...);
  if (!NT SUCCESS(status))
    return CompleteRequest(...);
  IoSkipCurrentIrpStackLocation(...);
  status = IoCallDriver(...);
  IoReleaseRemoveLock(...);
  return status;
  }

NTSTATUS DriverB DispatchSomething(...)
  {
  :
  return ??;
  }
```

Driver A's use of the remove lock protects Driver B until Driver B's dispatch routine returns. Thus, if Driver B completes the IRP or itself passes the IRP down using *IoSkipCurrentIrpStackLocation*, Driver B's involvement with the IRP will certainly be finished by the time Driver A is able to release the remove lock. If Driver B were to pend the IRP, Driver A wouldn't be holding the remove lock by the time Driver B got around to completing the IRP. We can assume, however, that Driver B will have some mechanism in place for purging its queues of pending IRPs before returning from its own *HandleRemoveDevice* function. Driver A won't call *IoDetachDevice* or return from its own *HandleRemoveDevice* function until afterwards.

The only time there will be a problem is if Driver B passes the IRP down with a completion routine installed via the original *IoSetCompletionRoutine* macro. Even here, if the lowest driver that handles this IRP does so correctly, *itsHandleRemoveDevice* function won't return until the IRP is completed. We'll have just a slim chance that Driver B could be unloaded before its completion routine runs.

There is, unfortunately, no way for a driver to completely protect itself from being unloaded while processing an IRP. Any scheme you or I can devise will inevitably risk executing at least one instruction (a return) after the system removes the driver image from memory. You can, however, hope that the drivers above you minimize the risk by using the techniques I've outlined here.

6.3.7 How the *DEVQUEUE* Works with PnP

In contrast with other examples in this book, I'm going to show you the full implementation of the *DEVQUEUE* object, even though the source code is in the companion content. I'm making an exception in this case because I think an annotated listing of the functions will make it easier for you to understand how to use it. We've already discussed the major routines in the preceding chapter, so I can focus here on the routines that dovetail with *IRP_MJ_PNP*.

Stalling the Queue

Stalling the IRP queue involves two *DEVQUEUE* functions:

```
VOID NTAPI StallRequests(PDEVQUEUE pdq)
  {
1
  InterlockedIncrement(&pdq->stallcount);
  }
```


IPR2024-00292
KASPERSKY EX 1008 Page 196

```
BOOLEAN NTAPI CheckBusyAndStall(PDEVQUEUE pdq)
  {
  KIRQL oldirql;
2
  KeAcquireSpinLock(&pdq->lock, &oldirql);
3
  BOOLEAN busy = pdq->CurrentIrp != NULL;
  if (!busy)
4
    InterlockedIncrement(&pdq->stallcount);
  KeReleaseSpinLock(&pdq->lock, oldirql);
  return busy;
  }
```

1. To stall requests, we just need to set the stall counter to a nonzero value. It's unnecessary to protect the increment with a spin lock because any thread that might be racing with us to change the value will also be using an interlocked increment or decrement.
2. Since *CheckBusyAndStall* needs to operate as an atomic function, we first take the queue's spin lock.
3. *CurrentIrp* being non-*NULL* is the signal that the device is busy handling one of the requests from this queue.
4. If the device is currently idle, this statement starts stalling the queue, thereby preventing the device from becoming busy later on.

Recall that *StartPacket* and *StartNextPacket* don't send IRPs to the queue's *StartIo* routine while the stall counter is nonzero. In addition, *InitializeQueue* initializes the stall counter to 1, so the queue begins life in the stalled state.

Restarting the Queue

RestartRequests is the function that unstalls a queue. This function is quite similar to *StartNextPacket*, which I showed you in Chapter 5.

```
VOID RestartRequests(PDEVQUEUE pdq, PDEVICE_OBJECT fdo)
  {
  KIRQL oldirql;
1
  KeAcquireSpinLock(&pdq->lock, &oldirql);
2
  if (InterlockedDecrement(&pdq->stallcount) > 0)
    {
    KeReleaseSpinLock(&pdq->lock, oldirql);
    return;
    }
3
  while (!pdq->stallcount && !pdq->CurrentIrp && !pdq->abortstatus
    && !IsListEmpty(&pdq->head))
    {
    PLIST_ENTRY next = RemoveHeadList(&pdq->head);
    PIRP Irp = CONTAINING_RECORD(next, IRP, Tail.Overlay.ListEntry);
    if (!IoSetCancelRoutine(Irp, NULL))
      {
      InitializeListHead(&Irp->Tail.Overlay.ListEntry);
      continue;
      }
    pdq->CurrentIrp = Irp;
    KeReleaseSpinLockFromDpcLevel(&pdq->lock);
    (*pdq->StartIo)(fdo, Irp);
    KeLowerIrql(oldirql);
    return;
    }
  KeReleaseSpinLock(&pdq->lock, oldirql);
  }
```

1. We acquire the queue spin lock to prevent interference from a simultaneous invocation of *StartPacket*.
2. Here we decrement the stall counter. If it's still nonzero, the queue remains stalled, and we return.
3. This loop duplicates a similar loop inside *StartNextPacket*. We need to duplicate the code here to accomplish all of this function's actions within one invocation of the spin lock.

IPR2024-00292
KASPERSKY EX 1008 Page 197

NOTE
True confession: The first edition described a much simpler—and incorrect—implementation of *RestartRequests*. A reader pointed out a race between the earlier implementation and *StartPacket*, which was corrected on my Web site as shown here.

Awaiting the Current IRP

The handler for *IRP_MN_STOP_DEVICE* might need to wait for the current IRP, if any, to finish by calling *WaitForCurrentIrp*:

```
VOID NTAPI WaitForCurrentIrp(PDEVQUEUE pdq)
  {
1
  KeClearEvent(&pdq->evStop);
2
  ASSERT(pdq->stallcount != 0);
  KIRQL oldirql;
3
  KeAcquireSpinLock(&pdq->lock, &oldirql);
  BOOLEAN mustwait = pdq->CurrentIrp != NULL;
  KeReleaseSpinLock(&pdq->lock, oldirql);
  if (mustwait)
    KeWaitForSingleObject(&pdq->evStop, Executive, KernelMode, FALSE, NULL);
  }
```

1. *StartNextPacket* signals the *evStop* event each time it's called. We want to be sure that the wait we're about to perform doesn't complete because of a now-stale signal, so we clear the event before doing anything else.
2. It doesn't make sense to call this routine without first stalling the queue. Otherwise, *StartNextPacket* will just start the next IRP if there is one, and the device will become busy again.
3. If the device is currently busy, we'll wait on the *evStop* event until someone calls *StartNextPacket* to signal that event. We need to protect our inspection of *CurrentIrp* with the spin lock because, in general, testing a pointer for *NULL* isn't an atomic event. If the pointer is *NULL* now, it can't change later because we've assumed that the queue is stalled.

Aborting Requests

Surprise removal of the device demands that we immediately halt every outstanding IRP that might try to touch the hardware. In addition, we want to make sure that all further IRPs are rejected. The *AbortRequests* function helps with these tasks:

```
VOID NTAPI AbortRequests(PDEVQUEUE pdq, NTSTATUS status)
  {
  pdq->abortstatus = status;
  CleanupRequests(pdq, NULL, status);
  }
```

Setting *abortstatus* puts the queue in the *REJECTING* state so that all future IRPs will be rejected with the status value our caller supplied. Calling *CleanupRequests* at this point—with a *NULL* file object pointer so that *CleanupRequests* will process the entire queue—empties the queue.

We don't dare try to do anything with the IRP, if any, that's currently active on the hardware. Drivers that don't use the hardware abstraction layer (HAL) to access the hardware—USB drivers, for example, which rely on the hub and host-controller drivers—can count on another driver to cause the current IRP to fail. Drivers that use the HAL might, however, need to worry about hanging the system or, at the very least, leaving an IRP in limbo because the nonexistent hardware can't generate the interrupt that would let the IRP finish. To deal with situations such as this, you call *AreRequestsBeingAborted*:

```
NTSTATUS AreRequestsBeingAborted(PDEVQUEUE pdq)
  {
  return pdq->abortstatus;
  }
```

It would be silly, by the way, to use the queue spin lock in this routine. Suppose we capture the instantaneous value of abortstatus in a thread-safe and multiprocessor-safe way. The value we return can become obsolete as soon as we release the spin lock.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 198

NOTE
If your device might be removed in such a way that an outstanding request simply hangs, you should also have some sort of watchdog timer running that will let you kill the IRP after a specified period of time. See the "Watchdog Timers" section in Chapter 14.

Sometimes we need to undo the effect of a previous call to *AbortRequest*. *AllowRequests* lets us do that:

```
VOID NTAPI AllowRequests(PDEVQUEUE pdq)
  {
  pdq->abortstatus = (NTSTATUS) 0;
  }
```

6.4 Other Configuration Functionality

Up to this point, I've talked about the important concepts you need to know to write a hardware device driver. I'll discuss two less important minor function codes—*IRP_MN_FILTER_RESOURCE_REQUIREMENTS* and *IRP_MN_DEVICE_USAGE_NOTIFICATION*—that you might need to process in a practical driver. Finally I'll mention how you can register to receive notifications about PnP events that affect devices other than your own.

6.4.1 Filtering Resource Requirements

Sometimes the PnP Manager is misinformed about the resource requirements of your driver. This can occur because of hardware and firmware bugs, mistakes in the INF file for a legacy device, or other reasons. The system provides an escape valve in the form of the *IRP_MN_FILTER_RESOURCE_REQUIREMENTS* request, which affords you a chance to examine and possibly alter the list of resources before the PnP Manager embarks on the arbitration and assignment process that culminates in your receiving a start device IRP.

When you receive a filter request, the *FilterResourceRequirements* substructure of the Parameters union in your stack location points to an *IO_RESOURCE_REQUIREMENTS_LIST* data structure that lists the resource requirements for your device. In addition, if any of the drivers above you have processed the IRP and modified the resource requirements, the *IoStatus.Information* field of the IRP will point to a second *IO_RESOURCE_REQUIREMENTS_LIST*, which is the one from which you should work. Your overall strategy will be as follows: If you want to add a resource to the current list of requirements, you do so in your dispatch routine. Then you pass the IRP down the stack synchronously—that is, by using the *ForwardAndWait* method you use with a start device request. When you regain control, you can modify or delete any of the resource descriptions that appear in the list.

Here's a brief and not very useful example that illustrates the mechanics of the filtering process:

```
NTSTATUS HandleFilterResources(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
1
  PIO_RESOURCE_REQUIREMENTS_LIST original = stack->Parameters
    .FilterResourceRequirements.IoResourceRequirementList;
2
  PIO_RESOURCE_REQUIREMENTS_LIST filtered =
    (PIO_RESOURCE_REQUIREMENTS_LIST) Irp->IoStatus.Information;
3
  PIO_RESOURCE_REQUIREMENTS_LIST source = filtered ? filtered : original;
4
  if (source->AlternativeLists != 1)
    return DefaultPnpHandler(fdo, Irp);
5
  ULONG sizelist = source->ListSize;
  PIO_RESOURCE_REQUIREMENTS_LIST newlist =
    (PIO_RESOURCE_REQUIREMENTS_LIST) ExAllocatePool(PagedPool,
    sizelist + sizeof(IO_RESOURCE_DESCRIPTOR));
  if (!newlist)
    return DefaultPnpHandler(fdo, Irp);
  RtlCopyMemory(newlist, source, sizelist);
6
  newlist->ListSize += sizeof(IO_RESOURCE_DESCRIPTOR);
  PIO_RESOURCE_DESCRIPTOR resource =
    &newlist->List[0].Descriptors[newlist->List[0].Count++];
  RtlZeroMemory(resource, sizeof(IO_RESOURCE_DESCRIPTOR));
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 199

```
    resource->Type = CmResourceTypeDevicePrivate;
    resource->ShareDisposition = CmResourceShareDeviceExclusive;
    resource->u.DevicePrivate.Data[0] = 42;
7
    Irp->IoStatus.Information = (ULONG_PTR) newlist;
    if (filtered && filtered != original)
      ExFreePool(filtered);
8
    NTSTATUS status = ForwardAndWait(fdo, Irp);
    if (NT_SUCCESS(status))
      {
      // stuff
      }
9
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
    }
```

1. The parameters for this request include a list of I/O resource requirements. These are derived from the device's configuration space, the registry, or wherever the bus driver happens to find them.
2. Higher-level drivers might have already filtered the resources by adding requirements to the original list. If so, they set the *IoStatus.Information* field to point to the expanded requirements list structure.
3. If there's no filtered list, we'll extend the original list. If there's a filtered list, we'll extend that.
4. Theoretically, several alternative lists of requirements could exist, but dealing with that situation is beyond the scope of this simple example.
5. We need to add any resources before we pass the request down the stack. First we allocate a new requirements list and copy the old requirements into it.
6. Taking care to preserve the preexisting order of the descriptors, we add our own resource description. In this example, we're adding a resource that's private to the driver.
7. We store the address of the expanded list of requirements in the IRP's *IoStatus.Information* field, which is where lower-level drivers and the PnP system will be looking for it. If we just extended an already filtered list, we need to release the memory occupied by the old list.
8. We pass the request down using the same *ForwardAndWait* helper function that we used for *IRP_MN_START_DEVICE*. If we weren't going to modify any resource descriptors on the IRP's way back up the stack, we could just call *DefaultPnpHandler* here and propagate the returned status.
9. When we complete this IRP, whether we indicate success or failure, we must take care not to modify the Information field of the I/O status block: it might hold a pointer to a resource requirements list that some driver—maybe even ours!—installed on the way down. The PnP Manager will release the memory occupied by that structure when it's no longer needed.

6.4.2 Device Usage Notifications

Disk drivers (and the drivers for disk controllers) in particular sometimes need to know extrinsic facts about how they're being used by the operating system, and the *IRP_MN_DEVICE_USAGE_NOTIFICATION* request provides a means to gain that knowledge. The I/O stack location for the IRP contains two parameters in the *Parameters.UsageNotification* substructure. See Table 6-4. The *InPath* value (a *Boolean*) indicates whether the device is in the device path required to support that usage, and the Type value indicates one of several possible special usages.

Parameter	Description
InPath	TRUE if device is in the path of the Type usage; FALSE if not
Type	Type of usage to which the IRP applies

Table 6-4. *Fields in the Parameters.UsageNotification Substructure of an I/O Stack Location*

In the subdispatch routine for the notification, you should have a switch statement (or other logic) that differentiates among the notifications you know about. In most cases, you'll pass the IRP down the stack. Consequently, a skeleton for the subdispatch function is as follows:

```
NTSTATUS HandleUsageNotification(PDEVICE_OBJECT fdo, PIRP Irp)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```


IPR2024-00292
KASPERSKY EX 1008 Page 200

```
PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
DEVICE_USAGE_NOTIFICATION_TYPE type =
  stack->Parameters.UsageNotification.Type;
BOOLEAN inpath = stack->Parameters.UsageNotification.InPath;
switch (type)
  {
case DeviceUsageTypeHibernation:
  :
  Irp->IoStatus.Status = STATUS_SUCCESS;
  break;
case DeviceUsageTypeDumpFile:
  :
  Irp->IoStatus.Status = STATUS_SUCCESS;
  break;
case DeviceUsageTypePaging:
  :
  Irp->IoStatus.Status = STATUS_SUCCESS;
  break;
default:
  break;
  }
return DefaultPnpHandler(fdo, Irp);
}
```

Set the *Status* field of the IRP to *STATUS_SUCCESS* for only the notifications that you explicitly recognize, as a signal to the bus driver that you've processed them. The bus driver will assume that you didn't know about—and therefore didn't process—a notification for which you don't set *STATUS_SUCCESS*.

You might know that your device can't support a certain kind of usage. Suppose, for example, that some fact that only you know prevents your disk device from being used to store a hibernation file. In such a case, you should have the IRP fail if it specifies the *InPath* value:

```
  :
case DeviceUsageTypeHibernation:
  if (inpath)
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
```

In the remainder of this section, I'll briefly describe each of the current usage types.

DeviceUsageTypePaging

The *InPathTRUE* notification indicates that a paging file will be opened on the device. The *InPathFALSE* notification indicates that a paging file has been closed. Generally, you should maintain a counter of the number of paging files you've been notified about. While any paging file remains active, you'll cause queries for *STOP* and *REMOVE* functions to fail. In addition, when you receive the first paging notification, make sure that your dispatch routines for *READ*, *WRITE*, *DEVICE_CONTROL*, *PNP*, and *POWER* requests are locked into memory. (Refer to the information on driver paging in "User-Mode and Kernel-Mode Address Spaces" in Chapter 3 for more information.) You should also clear the *DO_POWER_PAGABLE* flag in your device object to force the Power Manager to send you power IRPs at *DISPATCH_LEVEL*. To be safe, I'd also suggest nullifying any idle-notification registration you might have made. (See Chapter 8 for a discussion of idle detection.)

NOTE
In Chapter 8, I'll discuss how to set the *DO_POWER_PAGABLE* flag in a device object. You need to be sure that you never clear this flag while a device object under yours has the flag set. You'll want to clear the flag only in a completion routine, after the lower-level drivers have cleared their own flags. You need a completion routine anyway because you must undo anything you did in your dispatch routine if the IRP fails in the lower layers.

DeviceUsageTypeDumpFile

The *InPathTRUE* notification indicates that the device has been chosen as the repository for a crash dump file should one be necessary. The *InPathFALSE* notification cancels that. Maintain a counter of *TRUE* minus *FALSE* notifications. While the counter is nonzero:

- Make sure that your power management code—see Chapter 8—will never take the device out of the D0, or fully on, state. You can optimize your power behavior by inspecting the *ShutdownType* specified in system power IRPs in light of other usages of which you've been notified. Explaining this advanced topic is beyond the scope of this book.
- Avoid registering the device for idle detection, and nullify any outstanding registration.
- Make sure that your driver causes stop and remove queries to fail.

IPR2024-00292
KASPERSKY EX 1008 Page 201

DeviceUsageTypeHibernation

The *InPathTRUE* notification indicates that the device has been chosen to hold the hibernation state file should one be written. The *InPathFALSE* notification cancels that. You should maintain a counter of *TRUE* minus *FALSE* notifications. Your response to system power IRPs that specify the *PowerSystemHibernate* state will be different than normal because your device will be used momentarily to record the hibernate file. Elaboration of this particular feature of disk drivers is beyond the scope of this book.

6.4.3 PnP Notifications

Windows XP and Windows 98/Me provide a way to notify both user-mode and kernel-mode components of particular PnP events. Windows 95 has a *WM_DEVICECHANGE* message that user-mode programs can use to monitor, and sometimes control, hardware and power changes in the system. The newer operating systems build on *WM_DEVICECHANGE* to allow user-mode programs to easily detect when a driver enables or disables a registered device interface. Kernel-mode drivers can also register for similar notifications.

NOTE
Refer to the documentation for *WM_DEVICECHANGE*, *RegisterDeviceNotification*, and *UnregisterDeviceNotification* in the Platform SDK. I'll give you examples of using this message and these APIs, but I won't explain all possible uses of them. Some of the illustrations that follow also assume you're comfortable programming with Microsoft Foundation Classes.

Using *WM_DEVICECHANGE*

An application with a window can subscribe for *WM_DEVICECHANGE* messages related to a specific interface GUID (globally unique identifier). For example:

```
DEV_BROADCAST_DEVICEINTERFACE filter;
filter.dbcc_size = sizeof(filter);
filter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
filter.dbcc_classguid = GUID_DEVINTERFACE_PNPEVENT;
m_hInterfaceNotification = RegisterDeviceNotification(m_hWnd, &filter, 0);
```

NOTE
The PNPEVENT sample shows how to monitor WM_DEVICECHANGE in order to monitor device insertion and removal events. The examples in this section are drawn from the TEST program accompanying that sample. The PNPEVENT driver itself is actually not very interesting.

The key statement here is the call to *RegisterDeviceNotification*, which asks the PnP Manager to send our window a *WM_DEVICECHANGE* message whenever anyone enables or disables a *GUID_DEVINTERFACE_PNPEVENT* interface. So suppose a device driver calls *IoRegisterDeviceInterface* with this interface GUID during its *AddDevice* function. We're asking to be notified when that driver calls *IoSetDeviceInterfaceState* to either enable or disable that registered interface.

When an application has a handle to a device, it can register for notifications concerning that specific handle:

```
DEV_BROADCAST_HANDLE filter = {0};
filter.dbch_size = sizeof(filter);
filter.dbch_devicetype = DBT_DEVTYP_HANDLE;
filter.dbch_handle = m_hDevice;
m_hHandleNotification = RegisterDeviceNotification(m_hWnd, &filter, 0);
```

For each of the notification handles you register, you should eventually call *UnregisterDeviceNotification*. In preparing the first edition of this book, I found that this function was destabilizing Windows 98. A reader figured out these undocumented rules about how to call this function safely:

- Call *UnregisterDeviceNotification* while the window whose handle you specified in the registration call still exists.
- In Windows 98 (and, presumably, Windows Me), don't call *UnregisterDeviceNotification* from within the message handler for a notification relating to the same notification handle. Doing so is perfectly safe in Windows 2000 and Windows XP, though.

After your application registers for notifications, the system will send you *WM_DEVICECHANGE* window messages to alert you to various events of possible interest. I'll discuss here the *DBT_DEVICEQUERYREMOVE* and *DBT_DEVICEREMOVECOMPLETE* notifications, which are of particular interest to applications. Unless an application processes these notifications correctly, the PnP Manager can't successfully handle the two most common device removal scenarios.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 202

The *QUERYREMOVE* and *REMOVECOMPLETE* notifications relate to a specific handle. The TEST program for *PNPEVENT* handles them this way:

```
BEGIN MESSAGE MAP(CTestDlg, CDialog)
  //{{AFX MSG MAP(CTestDlg)
  .
  .
  //}}AFX MSG MAP
  ON WM DEVICECHANGE()END MESSAGE MAP()

BOOL CTestDlg::OnDeviceChange(UINT nEventType, DWORD dwData)
  {
  if (!dwData)
    return TRUE;

   DEV BROADCAST HEADER* p = ( DEV BROADCAST HEADER*) dwData;

  if (p->dbcd devicetype == DBT DEVTYP DEVICEINTERFACE)
    return HandleDeviceChange(nEventType,
      (PDEV BROADCAST DEVICEINTERFACE) p);
  else if (p->dbcd devicetype == DBT DEVTYP HANDLE)
    return HandleDeviceChange(nEventType,
      (PDEV BROADCAST HANDLE) p);
  else
    return TRUE;
  }

BOOL CTestDlg::HandleDeviceChange(DWORD evtype,
  PDEV BROADCAST HANDLE dhp)
  {

  if (dhp->dbch handle != m hDevice)
    return TRUE;

  switch (evtype)
    {

  case DBT DEVICEQUERYREMOVE:
1
    if (!<okay to remove device>)
      return BROADCAST QUERY DENY;

  case DBT DEVICEREMOVECOMPLETE:
  case DBT DEVICEREMOVEPENDING:
2
    if (m hHandleNotification && !win98)
      {
      UnregisterDeviceNotification(m hHandleNotification);
      m hHandleNotification = NULL;
      }
3
    CloseHandle(m hDevice);
    break;
    }

  return TRUE;
  }
```

1. TEST actually displays a message box at this point to ask you whether it will be OK to remove the device. In a real application, you might have some reason to demur. If you decide it's OK to remove the device, the code falls through into the next case. I found that it was necessary in Windows 98/Me to close my handle now rather than wait for another notification.

2. As I mentioned earlier, you can close a notification handle while handling a notification in Windows 2000 or XP but not in Windows 98/Me.

3. This is the point of all the machinery: we want to close our handle to the device when it's about to disappear or has already disappeared.

I suggest that you now try the following experiment to exercise both of these code paths. Launch the test program for PNPEVENT and install the PNPEVENT device. (See PNPEVENT.HTM for details of how to do this.) If you're running

IPR2024-00292
KASPERSKY EX 1008 Page 203

DbgView (see http://www.sysinternals.com), you'll observe the debug trace shown in lines 0 through 12 of Figure 6-6. The window for TEST will show an arrival message for the device (the first line in Figure 6-7). Now follow the instructions for installing and launching the PNPDTEST applet from the DDK Tools directory, and locate the device entry for PNPEVENT. (See Figure 6-8.) You'll find it indented below the Root node in the device list. Click the Test Surprise Remove button in PNPDTEST.

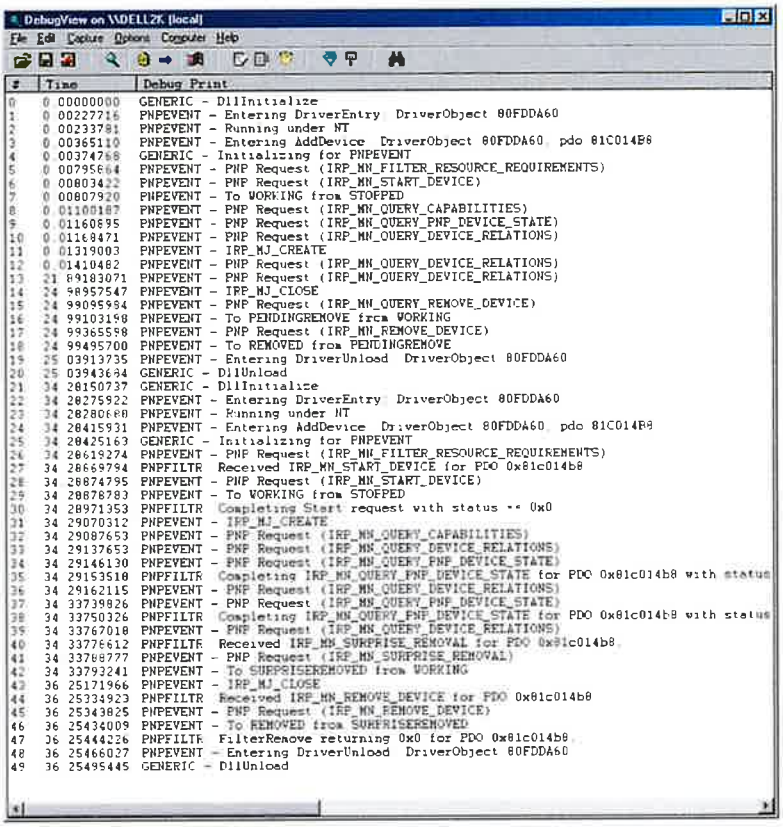


Figure 6-6. *Debug trace from the PNPEVENT experiment.*



Figure 6-7. *TEST's event trace.*

Because of the way PNPDTEST works internally, the first thing that happens is a *DBT_DEVICEQUERYREMOVE* notification so that PNPDTEST can disable the device in order to install a filter driver. TEST presents a dialog asking whether it's OK to remove the device (Figure 6-9). You should answer yes, whereupon TEST will close its handle (Figure 6-10). Thereafter, the PnP Manager will send an *IRP_MN_QUERY_REMOVE_DEVICE* followed by an *IRP_MN_REMOVE_DEVICE* to the PNPEVENT driver. This sequence corresponds to lines 13 through 20 of the debug trace and the second event message in the TEST window.

IPR2024-00292
KASPERSKY EX 1008 Page 204

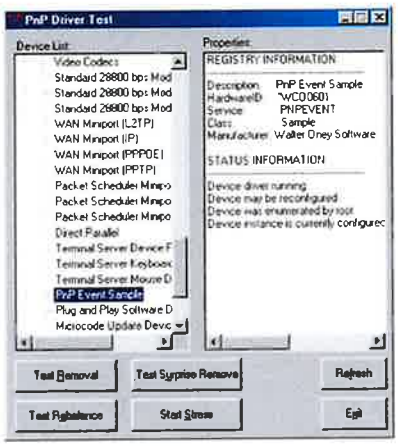


Figure 6-8. *PNPDTEST window.*

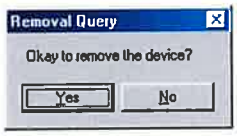


Figure 6-9. *TEST asks whether it's OK to remove the device.*



Figure 6-10. *TEST closes its handle.*

PNPDTEST will now restart the device, generating lines 21 through 32 of the debug trace and the third notification in the TEST window. Note that TEST opens a new handle in response to the arrival notification.

Finally PNPDTEST causes an *IRP_MN_SURPRISE_REMOVAL* request to be sent to the PNPEVENT device. (Actually, it causes its associated filter driver to call *IoInvalidateDeviceState*, which eventually triggers the surprise removal IRP in the way I discussed earlier. You can see the trace of this internal magic in lines 33 through 39 of the debug trace.) PNPEVENT processes this in the way we've discussed in this chapter. See lines 40 through 42 of the debug trace.

At this point, PNPEVENT will be in the *SURPRISEREMOVED* state. The driver can't be unloaded because a handle is still open. The test application will receive a *WM_DEVICECHANGE* with the *DBT_DEVICEREMOVECOMPLETE* code. The application closes its handle (debug trace line 43), whereupon the PnP Manager finishes up by sending an *IRP_MN_REMOVE_DEVICE* to PNPEVENT (debug trace lines 44 through 49).

You should notice that the application never receives a query in the surprise removal case.

I want to mention one fine point in connection with the *DBT_DEVICEQUERYREMOVE* notification. According to the Platform SDK documentation for this query, an application might return the special value *BROADCAST_QUERY_DENY* to decline permission to remove the device. This works as expected in Windows XP. That is, if you go to the Device Manager and attempt to remove the device, and if the application declines permission, the device will not be removed. In fact, the PnP Manager won't even send the *IRP_MN_QUERY_REMOVE_DEVICE* request to the driver in this situation.

In other versions of the operating system, however, *BROADCAST_QUERY_DENY* doesn't work as expected. The Device Manager will appear to ignore the return code and proceed to remove the device from its window and to mark the device for deletion. It will realize that the device can't yet be removed, however, so it will post a dialog to the effect that you must restart the system for the removal to take effect. The driver remains in memory.

Notifications to Windows XP Services

Windows XP service programs can also subscribe for PnP notifications. The service should call *RegisterServiceCtrlHandlerEx* to register an extended control handler function. Then it can register for service control notifications about device interface changes. For example, take a look at the following code (and see the AUTOLAUNCH sample in Chapter 15):

IPR2024-00292
KASPERSKY EX 1008 Page 205

```
DEV BROADCAST DEVICEINTERFACE filter = {0};
filter.dbcc size = sizeof(filter);
filter.dbcc devicetype = DBT DEVTYPE DEVICEINTERFACE;
filter.dbcc classguid = GUID AUTOLAUNCH NOTIFY;
m hNotification = RegisterDeviceNotification(m hService,
  (PVOID) &filter, DEVICE_NOTIFY_SERVICE_HANDLE);
```

Here m_*hService* is a service handle provided by the service manager when it starts your service, and *DEVICE_NOTIFY_SERVICE_HANDLE* indicates that you're registering for service control notifications instead of window messages. After receiving a *SERVICE_CONTROL_STOP* command, you want to deregister the notification handle:

```
UnregisterDeviceNotification(m_hNotification);
```

When a PnP event involving the interface GUID occurs, the system calls your extended service control handler function:

```
DWORD   stdcall HandlerEx(DWORD ctlcode, DWORD evtype,
  PVOID evdata, PVOID context)
  {
  }
```

where *ctlcode* will equal *SERVICE_CONTROL_DEVICEEVENT*, *evtype* will equal *DBT_DEVICEARRIVAL* or one of the other *DBT_Xxx* codes, *evdata* will be the address of a Unicode version of the *DEV_BROADCAST_DEVICEINTERFACE* structure, and *context* will be whatever context value you specified in your call to the *RegisterServiceCtrlHandlerEx* function.

Kernel-Mode Notifications

WDM drivers can use *IoRegisterPlugPlayNotification* to subscribe for interface and handle notifications. Here's an exemplary statement from the PNPMON sample driver that registers for notifications about the arrival and departure of an interface GUID designated by an application—PNPMON's TEST.EXE in this case—via an I/O control (IOCTL) operation:

```
status = IoRegisterPlugPlayNotification
  (EventCategoryDeviceInterfaceChange,
  PNPNOTIFY DEVICE INTERFACE INCLUDE EXISTING INTERFACES,
  &p->guid, pdx->DriverObject,
  (PDRIVER NOTIFICATION CALLBACK ROUTINE) OnPnpNotify,
  reg, &reg->InterfaceNotificationEntry);
```

The first argument indicates that we want to receive notifications whenever someone enables or disables a specific interface GUID. The second argument is a flag indicating that we want to receive callbacks right away for all instances of the interface GUID that are already enabled. This flag allows our driver to start after some or all of the drivers that export the interface in question and still receive notification callbacks about those interfaces. The third argument is the interface GUID in question. In this case, it comes to us via an IOCTL from an application. The fourth argument is the address of our driver object. The PnP Manager adds a reference to the object so that we can't be unloaded while we have any notification handles outstanding. The fifth argument is the address of a notification callback routine. The sixth argument is a context parameter for the callback routine. In this case, I specified the address of a structure (reg) that contains information relative to this registration call. The seventh and final argument gives the address of a variable where the PnP Manager should record a notification handle. We'll eventually call *IoUnregisterPlugPlayNotification* with the notification handle.

You need to call *IoUnregisterPlugPlayNotification* to close the registration handle. Because *IoRegisterPlugPlayNotification* adds a reference to your driver object, it won't do you any particular good to put this call in your *DriverUnload* routine. *DriverUnload* won't be called until the reference count drops to 0, which will never happen if *DriverUnload* itself has the deregistration calls. This problem isn't hard to solve—you just need to pick an appropriate time to deregister, such as when you notice the last interface of a particular type being removed or in response to an IOCTL request from an application.

Given a symbolic link name for an enabled interface, you can also request notifications about changes to the device named by the link. For example:

```
PUNICODE STRING SymbolicLinkName; // <== input to this process
PDEVICE OBJECT DeviceObject; // <== an output
PFILE OBJECT FileObject; // <== another output
IoGetDeviceObjectPointer(&SymbolicLinkName, 0, &FileObject, &DeviceObject);
IoRegisterPlugPlayNotification(EventCategoryTargetDeviceChange, 0,
  FileObject, pdx->DriverObject,
  (PDRIVER NOTIFICATION CALLBACK ROUTINE) OnPnpNotify,
  reg, &reg->HandleNotificationEntry);
```

You shouldn't put this code inside your PnP event handler, by the way. *IoGetDeviceObjectPointer* internally performs an open operation for the named device object. A deadlock might occur if the target device were to perform certain kinds of PnP

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 206

operations. You should instead schedule a work item by calling *IoQueueWorkItem*. Chapter 14 has more information about work items. The PNPMON sample driver illustrates how to use a work item in this particular situation.

The notifications that result from these registration calls take the form of a call to the callback routine you specified:

```
NTSTATUS OnPnpNotify(PPLUGPLAY NOTIFICATION HEADER hdr, PVOID Context)
  {
  .
  .
  .
  return STATUS SUCCESS;
  }
```

The *PLUGPLAY_NOTIFICATION_HEADER* structure is the common header for several different structures that the PnP Manager uses for notifications:

```
typedef struct  PLUGPLAY NOTIFICATION HEADER {
  USHORT Version;
  USHORT Size;
  GUID Event;
  } PLUGPLAY_NOTIFICATION_HEADER, *PPLUGPLAY_NOTIFICATION_HEADER;
```

The *Event* GUID indicates what sort of event is being reported to you. See Table 6 - 5. The DDK header file WDMGUID.H contains the definitions of these GUIDs.

GUID Name	Purpose of Notification
GUID_HWPROFILE_QUERY_CHANGE	OK to change to a new hardware profile?
GUID_HWPROFILE_CHANGE_CANCELLED	Change previously queried about has been cancelled.
GUID_HWPROFILE_CHANGE_COMPLETE	Change previously queried about has been accomplished.
GUID_DEVICE_INTERFACE_ARRIVAL	A device interface has just been enabled.
GUID_DEVICE_INTERFACE_REMOVAL	A device interface has just been disabled.
GUID_TARGET_DEVICE_QUERY_REMOVE	OK to remove a device object?
GUID_TARGET_DEVICE_REMOVE_CANCELLED	Removal previously queried about has been cancelled.
GUID_TARGET_DEVICE_REMOVE_COMPLETE	Removal previously queried about has been accomplished.

Table 6-5. *PnP Notification GUIDs*

If you receive either of the *DEVICE_INTERFACE* notifications, you can cast the *hdr* argument to the callback function as a pointer to the following structure:

```
typedef struct  DEVICE INTERFACE CHANGE NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  GUID InterfaceClassGuid;
  PUNICODE STRING SymbolicLinkName;
  } DEVICE_INTERFACE_CHANGE_NOTIFICATION, *PDEVICE_INTERFACE_CHANGE_NOTIFICATION;
```

In the interface change notification structure, *InterfaceClassGuid* is the interface GUID, and *SymbolicLinkName* is the name of an instance of the interface that's just been enabled or disabled.

If you receive any of the *TARGET_DEVICE* notifications, you can cast the *hdr* argument as a pointer to this structure instead:

```
typedef struct  TARGET DEVICE REMOVAL NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  PFILE OBJECT FileObject;
  } TARGET_DEVICE_REMOVAL_NOTIFICATION, *PTARGET_DEVICE_REMOVAL_NOTIFICATION;
```

where *FileObject* is the file object for which you requested notifications.

Finally, if you receive any of the *HWPROFILE_CHANGE* notifications, *hdr* will really be a pointer to this structure:

```
typedef struct  HWPROFILE CHANGE NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  } HWPROFILE_CHANGE_NOTIFICATION, *PHWPROFILE_CHANGE_NOTIFICATION;
```


IPR2024-00292
KASPERSKY EX 1008 Page 207

This doesn't have any more information than the header structure itself—just a different typedef name.

One way to use these notifications is to implement a filter driver for an entire class of device interfaces. (There is a standard way to implement filter drivers, either for a single driver or for a class of devices, based on setting entries in the registry. I'll discuss that subject in Chapter 16. Here I'm talking about filtering all devices that register a particular interface, for which there's no other mechanism.) In your driver's *DriverEntry* routine, you register for PnP notifications about one or more interface GUIDs. When you receive the arrival notification, you use *IoGetDeviceObjectPointer* to open a file object and then register for target device notifications about the associated device. You also get a device object pointer from *IoGetDeviceObjectPointer*, and you can send IRPs to that device by calling *IoCallDriver*. Be on the lookout for the *GUID_TARGET_DEVICE_QUERY_REMOVE* notification because you have to dereference the file object before the removal can continue.

The PNPMON Sample
The PNPMON sample illustrates how to register for and process PnP notifications in kernel mode. To give you something you can run on your computer and actually see working, I designed PNPMON to simply pass notifications back to a user-mode application (named TEST—what else?). This is pretty silly in that a user-mode application can get these notifications on its own by calling *RegisterDeviceNotification*.

PNPMON is different from the other driver samples in this book. It's intended to be dynamically loaded as a helper for a user-mode application. The other drivers we look at are intended to manage hardware, real or imagined. The user-mode application uses service manager API calls to load PNPMON, which creates exactly one device object in its *DriverEntry* routine so that the application can use *DeviceIoControl* to get things done in kernel mode. When the application exits, it closes its handle and calls the service manager to terminate the driver.

PNPMON also includes a Windows 98/Me virtual device driver (VxD) that the test application can dynamically load. It's possible to dynamically load a WDM driver in Windows 98/Me by using an undocumented function (*_NtKernLoadDriver*, if you care), but there's no way to unload a driver that you've loaded in this way. You don't need to resort to undocumented functions, though, because VxDs can call most of the WDM support routines directly by means of the WDMVXD import library in the Windows 98 DDK. (This library is missing from the Windows Me portion of the Windows XP DDK.) Just about the only extra things you need to do in your VxD project are include WDM.H ahead of the VxD header files and add WDMVXD.CLB to the list of inputs to the linker. So PNPMON.VXD simply registers for PnP notifications as if it were a WDM driver and supports the same IOCTL interface that PNPMON.SYS supports.

Custom Notifications

I'll close this section by explaining how a WDM driver can generate custom PnP notifications. To signal a custom PnP event, create an instance of the custom notification structure and call one of *IoReportTargetDeviceChange* or *IoReportTargetDeviceChangeAsynchronous*. The asynchronous flavor returns immediately. The synchronous flavor waits—a long time, in my experience—until the notification has been sent. The notification structure has this declaration:

```
typedef struct  TARGET DEVICE CUSTOM NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  PFILE OBJECT FileObject;
  LONG NameBufferOffset;
  UCHAR CustomDataBuffer[1];
  } TARGET_DEVICE_CUSTOM_NOTIFICATION, *PTARGET_DEVICE_CUSTOM_NOTIFICATION;
```

Event is the custom GUID you've defined for the notification. *FileObject* is NULL—the PnP Manager will be sending notifications to drivers who opened file objects for the same PDO as you specify in the *IoReportXxx* call. *CustomDataBuffer* contains whatever binary data you elect followed by Unicode string data. *NameBufferOffset* is -1 if you don't have any string data; otherwise, it's the length of the binary data that precedes the strings. You can tell how big the total data payload is by subtracting the field offset of *CustomDataBuffer* from the Size value.

Here's how PNPEVENT generates a custom notification when you press the Send Event button in the associated test dialog:

```
struct  RANDOM NOTIFICATION  : public  TARGET DEVICE CUSTOM NOTIFICATION {
  WCHAR text[14];
  };
:
RANDOM NOTIFICATION notify;
notify.Version = 1;
notify.Size = sizeof(notify);
notify.Event = GUID PNPEVENT EVENT;
notify.FileObject = NULL;
notify.NameBufferOffset = FIELD_OFFSET(RANDOM_NOTIFICATION, text)
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 208

```
    - FIELD_OFFSET(RANDOM_NOTIFICATION, CustomDataBuffer);
  *(PULONG)(notify.CustomDataBuffer) = 42;
  wcscpy(notify.text, L"Hello, world!");
  IoReportTargetDeviceChangeAsynchronous(pdx->Pdo, &notify, NULL, NULL);
```

That is, PNPEVENT generates a custom notification whose data payload contains the number 42 followed by the string *Hello, world!*.

The notification shows up in any driver that registered for target device notifications pertaining to a file object for the same PDO. If your notification callback routine gets a notification structure with a nonstandard GUID in the Event field, you can expect that it's somebody's custom notification GUID. You need to understand what the GUID means before you go mucking about in the *CustomDataBuffer*!

User-mode applications are supposed to be able to receive custom event notifications too, but I've not been able to get that to work.

6.5 Windows 98/Me Compatibility Notes

There are a few important differences between Windows 98/Me on the one hand and Windows 2000/XP on the other with respect to Plug and Play.

6.5.1 Surprise Removal

Windows 98/Me never sends an *IRP_MN_SURPRISE_REMOVAL* request. Consequently, a WDM driver needs to treat an unexpected *IRP_MN_REMOVE_DEVICE* as indicating surprise removal. The code samples I showed you in this chapter accomplish that by calling *AbortRequests* and *StopDevice* when they get this IRP out of the blue.

6.5.2 PnP Notifications

Windows 98/Me has calls to the *IoReportTargetDeviceChange* function fail with *STATUS_NOT_IMPLEMENTED*. It doesn't export the symbol *IoReportTargetDeviceChangeAsynchronous* at all; a driver that calls that function will simply fail to load in Windows 98/Me. Refer to Appendix A for information about how you can stub this and other missing support functions so as to be able to ship a single driver binary.

6.5.3 The Remove Lock

The original edition of Windows 98 didn't include any of the remove lock functions. Windows 98, Second Edition, and Windows Me include all but *IoReleaseRemoveLockAndWait*, which is the same as not supporting any of the functions, in my view. I mean, the whole point of the remove lock mechanism is to gate *IRP_MN_REMOVE_DEVICE*, and that hinges on the omitted function.

To make matters worse, a driver that references a function that the Windows 98/Me kernel doesn't export simply won't load.

DDK sample programs cope with this incompatibility in one of two ways. Some samples use a custom-built mechanism instead of an *IO_REMOVE_LOCK*. Others provide functions with names like *XxxAcquireRemoveLock*, and so on, that mimic the names of the standard remove lock functions.

My sample drivers use a variation of the second of these approaches. By means of #define statements, I substitute my own declarations of the *IO_REMOVE_LOCK* object and support functions for the official ones. Thus, my sample code calls *IoAcquireRemoveLock*, and so on. In the samples that use GENERIC.SYS, preprocessor trickery actually routes these calls to functions with names such as *GenericAcquireRemoveLock* that reside in GENERIC.SYS. In the samples that don't use GENERIC.SYS, the preprocessor trickery routes the calls to functions with names such as *AcquireRemoveLock* that are located in a file named REMOVELOCK.CPP.

I could have written my samples in such a way that they would call the standard remove lock functions instead of my own in Windows XP. To make any of the samples work in Windows 98/Me, I'd have needed to have you install WDMSTUB.SYS before you could run any of the samples. (See Appendix A.) I didn't think this was a good way to explain WDM programming.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 209

Chapter 7

Reading and Writing Data

All the infrastructure I've described so far in this book leads up to this chapter, where I finally cover how to read and write data from a device. I'll discuss the service functions you call to perform these important operations on a device plugged in to one of the traditional buses, such as Peripheral Component Interconnect (PCI). Since many devices use a hardware interrupt to notify system software about I/O completion or exceptional events, I'll also discuss how to handle an interrupt. Interrupt processing normally requires you to schedule a deferred procedure call (DPC), so I'll describe the DPC mechanism too. Finally I'll tell you how to arrange direct memory access (DMA) transfers between your device and main memory.

7.1 Configuring Your Device

In the preceding chapter, I discussed the various *IRP_MJ_PNP* requests that the Plug and Play (PnP) Manager sends you. *IRP_MN_START_DEVICE* is the vehicle for giving you information about the I/O resources that have been assigned by the PnP Manager for your use. I showed you how to obtain parallel lists of raw and translated resource descriptions and how to call a StartDevice helper function that would have the following prototype:

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
  PCM_PARTIAL_RESOURCE_LIST raw,
  PCM_PARTIAL_RESOURCE_LIST translated)
  {
  :
  }
```

The time has now come to explain what to do with these resource lists. In summary, you'll extract descriptions of your assigned resources from the translated list and use those descriptions to create additional kernel objects that give you access to your hardware.

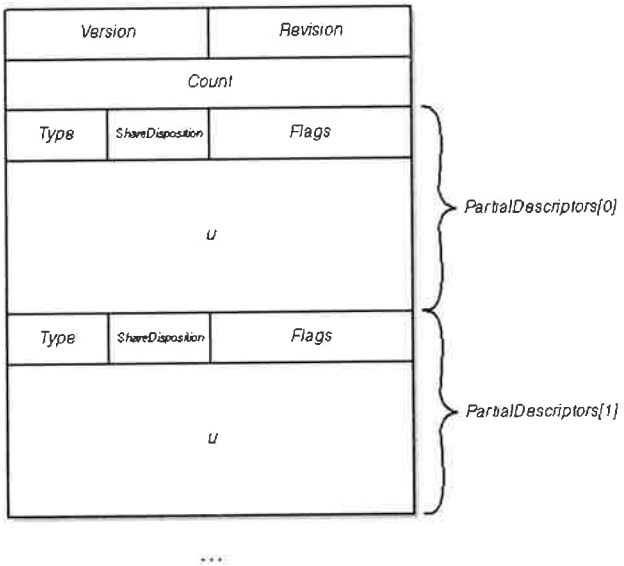


Figure 7-1. *Structure of a partial resource list.*

The *CM_PARTIAL_RESOURCE_LIST* structures contain a count and an array of *CM_PARTIAL_RESOURCE_DESCRIPTOR* structures, as illustrated in Figure 7-1. Each resource descriptor in the array has a Type member that indicates the type of resource it describes and some additional members that supply the particulars about some allocated resource. You're not going to be surprised by what you find in this array, by the way: if your device uses an IRQ and a range of I/O ports, you'll get two resource descriptors in the array. One of the descriptors will be for your IRQ, and the other will be for your I/O port range. Unfortunately, you can't predict in advance the order in which these descriptors will happen to appear in the array.

IPR2024-00292
KASPERSKY EX 1008 Page 211

Consequently, your *StartDevice* helper function has to begin with a loop that "flattens" the array by extracting resource values into a collection of local variables. You can later use the local variables to deal with the assigned resources in whatever order you please (which, it goes without saying, can be different from the order in which the PnP Manager chose to present them to you).

In sketch, then, your StartDevice function looks like this:

```
NTSTATUS StartDevice(PDEVICE OBJECT fdo, PCM PARTIAL RESOURCE LIST raw,
  PCM PARTIAL RESOURCE LIST translated)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
  PCM PARTIAL RESOURCE DESCRIPTOR resource = translated->PartialDescriptors;
2
  ULONG nres = translated->Count;
3
  <local variable declarations>
  for (ULONG i = 0; i < nres; ++i, ++resource)
    {
4
    switch (resource->Type)
      {
    case CmResourceTypePort:
      <save port info in local variables>
      break;
    case CmResourceTypeInterrupt:
      <save interrupt info in local variables>
      break;
    case CmResourceTypeMemory:
      <save memory info in local variables>
      break;
    case CmResourceTypeDma:
      <save DMA info in local variables>
      break;
      }
    }
5
  <use local variables to configure driver & hardware>  return STATUS SUCCESS;
  }
```

1. I'll use the resource pointer to point to the current resource descriptor in the variable-length array. By the end of the upcoming loop, it will point past the last valid descriptor.
2. The Count member of a resource list indicates how many resource descriptors are in the *PartialDescriptors* array.
3. You should declare appropriate local variables for each of the I/O resources you expect to receive. I'll detail what these should be later on when I discuss how to deal with each of the standard I/O resources.
4. Within the loop over resource descriptors, you use a switch statement to save resource description information into the appropriate local variables. In the text, I posited a device that needed just an I/O port range and an interrupt, and such a device would expect to find resource types *CmResourceTypePort* and *CmResourceTypeInterrupt*. I'm showing the other two standard resource types—*CmResourceTypeMemory* and *CmResourceTypeDma*—for thoroughness.
5. Once outside the loop, the local variables you initialized in the various case labels will hold the resource information you need.

If you have more than one resource of a particular type, you need to invent a way to tell the resource descriptors apart. To give a concrete (but entirely fictitious) example, suppose your device uses one 4-KB range of memory for control purposes and a different, 16-KB, range of memory as a data capture buffer. You expect to receive two *CmResourceTypeMemory* resources from the PnP Manager. The control memory is the block that's 4 KB long, whereas the data memory is the block that's 16 KB long. If your device's resources have a distinguishing characteristic such as the size difference in the example, you'll be able to tell which resource is which.

When dealing with multiple resources of the same type, don't assume that the resource descriptors will be in the same order that your configuration space lists them in, and don't assume that the same bus driver will always construct resource descriptors in the same order on every platform or every release of the operating system. The first assumption is tantamount to assuming that the bus driver programmer adopted a particular algorithm, while the second is tantamount to assuming that all bus driver programmers think alike and will never change their minds.

I'll explain how to deal with each of the four standard I/O resource types at appropriate places in the remainder of this chapter. Table 7-1 presents an overview of the critical step (or steps) for each type of resource.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 212

Resource Type	Overview
Port	Possibly map port range; save base port address in device extension
Memory	Map memory range; save base address in device extension
Dma	Call *IoGetDmaAdapter* to create an adapter object
Interrupt	Call *IoConnectInterrupt* to create an interrupt object that points to your interrupt service routine (ISR)

Table 7-1. *Overview of Processing Steps for I/O Resources*

7.2 Addressing a Data Buffer

When an application initiates a read or write operation, it provides a data buffer by giving the I/O Manager a user-mode virtual address and length. As I said back in Chapter 3, a kernel driver hardly ever accesses memory using a user-mode virtual address because, in general, you can't pin down the thread context with certainty. MicrosoftWindows XP gives you three ways to access a user-mode data buffer:

1. In the *buffered* method, the I/O Manager creates a system buffer equal in size to the user-mode data buffer. You work with this system buffer. The I/O Manager takes care of copying data between the user-mode buffer and the system buffer.
2. In the *direct* method, the I/O Manager locks the physical pages containing the user-mode buffer and creates an auxiliary data structure called a memory descriptor list (MDL) to describe the locked pages. You work with the MDL.
3. In the *neither* method, the I/O Manager simply passes the user-mode virtual address to you. You work—very carefully!—with the user-mode address.

Figure 7-2 illustrates the first two methods. The last method, of course, is kind of a nonmethod in that the system doesn't do anything to help you reach your data.

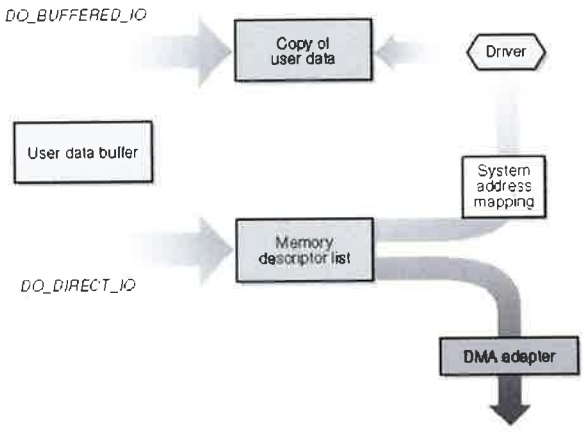


Figure 7-2. *Accessing user-mode data buffers.*

7.2.1 Specifying a Buffering Method

You specify your device's buffering method for reads and writes by setting certain flag bits in your device object shortly after you create it in your *AddDevice* function:

```
NTSTATUS AddDevice(...)
  {
  PDEVICE OBJECT fdo;
  IoCreateDevice(..., &fdo);
  fdo->Flags |= DO BUFFERED IO;
          <or>
  fdo->Flags |= DO DIRECT IO;
          <or>
  fdo->Flags |= 0; // i.e., neither direct nor buffered
  }
```

You can't change your mind about the buffering method afterward. Filter drivers might copy this flag setting and will have no way of knowing if you *do* change your mind and specify a different buffering method.

IPR2024-00292
KASPERSKY EX 1008 Page 213

The Buffered Method

When the I/O Manager creates an *IRP_MJ_READ* or *IRP_MJ_WRITE* request, it inspects the direct and buffered flags to decide how to describe the data buffer in the new I/O request packet (IRP). If *DO_BUFFERED_IO* is set, the I/O Manager allocates nonpaged memory equal in size to the user buffer. It saves the address and length of the buffer in two wildly different places, as shown in boldface in the following code fragment. You can imagine the I/O Manager code being something like this—this is not the actual Microsoft Windows NT source code.

```
PVOID uva;              // <== user-mode virtual buffer address
ULONG length;           // <== length of user-mode buffer

PVOID sva = ExAllocatePoolWithQuota(NonPagedPoolCacheAligned, length);
if (writing)
  RtlCopyMemory(sva, uva, length);

Irp->AssociatedIrp.SystemBuffer = sva;
PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
if (reading)
  stack->Parameters.Read.Length = length;else
  stack->Parameters.Write.Length = length;
<code to send and await IRP>
if (reading)
  RtlCopyMemory(uva, sva, length);

ExFreePool(sva);
```

In other words, the system (copy) buffer address is in the IRP's *AssociatedIrp.SystemBuffer* field, and the request length is in the *stack->Parameters* union. This process includes additional details that you and I don't need to know to write drivers. For example, the copy that occurs after a successful read operation actually happens during an asynchronous procedure call (APC) in the original thread context and in a different subroutine from the one that constructs the IRP. The I/O Manager saves the user-mode virtual address (my *uva* variable in the preceding fragment) in the IRP's *UserBuffer* field so that the copy step can find it. Don't count on either of these facts, though—they're subject to change at any time.

The I/O Manager also takes care of releasing the free storage obtained for the system copy buffer when somebody eventually completes the IRP.

The Direct Method

If you specified *DO_DIRECT_IO* in the device object, the I/O Manager creates an MDL to describe locked pages containing the user-mode data buffer. The MDL structure has the following declaration:

```
typedef struct  MDL {
  struct  MDL *Next;
  CSHORT Size;
  CSHORT MdlFlags;
  struct  EPROCESS *Process;
  PVOID MappedSystemVa;
  PVOID StartVa;
  ULONG ByteCount;
  ULONG ByteOffset;
  } MDL, *PMDL;
```

Figure 7-3 illustrates the role of the MDL. The *StartVa* member gives the virtual address—valid only in the context of the user-mode process that owns the data—of the buffer. *ByteOffset* is the offset of the beginning of the buffer within a page frame, and *ByteCount* is the size of the buffer in bytes. The *Pages* array, which isn't formally declared as part of the MDL structure, follows the MDL in memory and contains the numbers of the physical page frames to which the user-mode virtual addresses map.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 214

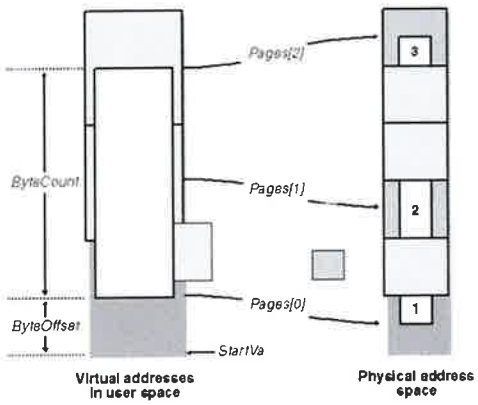


Figure 7-3. *The memory descriptor list structure.*

e never, by the way, access members of an MDL structure directly. We use macros and support functions instead—see Table 7-2.

Macro or Function	Description
IoAllocateMdl	Creates an MDL.
IoBuildPartialMdl	Builds an MDL for a subset of an existing MDL.
IoFreeMdl	Destroys an MDL.
MmBuildMdlForNonPagedPool	Modifies an MDL to describe a region of kernel-mode nonpaged memory.
MmGetMdlByteCount	Determines byte size of buffer.
MmGetMdlByteOffset	Gets buffer offset within first page.
MmGetMdlPfnArray	Locates array of physical page pointers.
MmGetMdlVirtualAddress	Gets virtual address.
MmGetSystemAddressForMdl	Creates a kernel-mode virtual address that maps to the same locations in memory.
MmGetSystemAddressForMdlSafe	Same as *MmGetSystemAddressForMdl* but preferred in Windows 2000 and later systems.
MmInitializeMdl	(Re)initializes an MDL to describe a given virtual buffer.
MmMapLockedPages	Creates a kernel-mode virtual address that maps to the same locations in memory.
MmMapLockedPagesSpecifyCache	Similar to *MmMapLockedPages* but preferred in Windows 2000 and later systems.
MmPrepareMdlForReuse	Reinitializes an MDL.
MmProbeAndLockPages	Locks pages after verifying address validity.
MmSizeOfMdl	Determines how much memory would be needed to create an MDL to describe a given virtual buffer. You don't need to call this routine if you use *IoAllocateMdl* to create the MDL in the first place.
MmUnlockPages	Unlocks the pages for this MDL.
MmUnmapLockedPages	Undoes a previous *MmMapLockedPages*.

Table 7-2. *Macros and Support Functions for Accessing an MDL*

You can imagine the I/O Manager executing code like the following to perform a direct-method read or write:

```
KPROCESSOR MODE mode;    // <== either KernelMode or UserMode
PMDL mdl = IoAllocateMdl(uva, length, FALSE, TRUE, Irp);
MmProbeAndLockPages(mdl, mode, reading ? IoWriteAccess : IoReadAccess);
<code to send and await IRP>
MmUnlockPages(mdl);
IoFreeMdl(mdl);
```

The I/O Manager first creates an MDL to describe the user buffer. The third argument to *IoAllocateMdl* (*FALSE*) indicates that this is the primary data buffer. The fourth argument (*TRUE*) indicates that the Memory Manager should charge the process quota. The last argument (*Irp*) specifies the IRP to which this MDL should be attached. Internally, *IoAllocateMdl* sets *Irp->MdlAddress* to the address of the newly created MDL, which is how you find it and how the I/O Manager eventually

IPR2024-00292
KASPERSKY EX 1008 Page 215

finds it so as to clean up.

The key event in this code sequence is the call to *MmProbeAndLockPages*, shown in boldface. This function verifies that the data buffer is valid and can be accessed in the appropriate mode. If we're writing to the device, we must be able to read the buffer. If we're reading from the device, we must be able to write to the buffer. In addition, the function locks the physical pages containing the data buffer and fills in the array of page numbers that follows the MDL proper in memory. In effect, a locked page becomes part of the nonpaged pool until as many callers unlock it as locked it in the first place.

The thing you'll most likely do with an MDL in a direct-method read or write is to pass it as an argument to somebody else. DMA transfers, for example, require an MDL for the *MapTransfer* step you'll read about later in this chapter in "Performing DMA Transfers." Universal serial bus (USB) reads and writes, to give another example, always work internally with an MDL, so you might as well specify *DO_DIRECT_IO* and pass the resulting MDLs along to the USB bus driver.

Incidentally, the I/O Manager does save the read or write request length in the *stack->Parameters* union. It's nonetheless customary for drivers to learn the request length directly from the MDL:

```
ULONG length = MmGetMdlByteCount(mdl);
```

The Neither Method

If you omit both the *DO_DIRECT_IO* and *DO_BUFFERED_IO* flags in the device object, you get the neither method by default. The I/O Manager simply gives you a user-mode virtual address and a byte count (as shown in boldface) and leaves the rest to you:

```
Irp->UserBuffer = uva;PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
if (reading)
  stack->Parameters.Read.Length = length;
else
  stack->Parameters.Write.Length = length;
<code to send and await IRP>
```



Never simply access memory using a pointer you get from user mode. You can use *MmProbeAndLockPages* to create an MDL for user-mode pages. Alternatively, you can call *ProbeForRead* or *ProbeForWrite* to verify that a range of addresses truly belongs to user mode and then access the memory directly. You should use a structured exception frame in either case to avoid a bug check in case there's some problem with the pointer or the length. For example:

```
PVOID buffer = Irp->UserBuffer;
ULONG length = stack->Parameters.Read.Length;
if (Irp->RequestorMode != KernelMode)
  {
    try
    {
    PMDL mdl = IoAllocateMdl(...);
    MmProbeAndLockPages(...);

          -or-

    ProbeForRead(...);
    <access memory at buffer>
    }
    except(EXCEPTION EXECUTE HANDLER)
    {
    return CompleteRequest(Irp, GetExceptionCode(), 0);
    }
```

7.3 Ports and Registers

Windows XP uses the abstract computer model depicted in Figure 7-4 to provide a unified driver interface in all CPU architectures. In this mode, a CPU can have separate memory and I/O address spaces. To access a *memory-mapped* device, the CPU employs a memory-type reference such as a load or a store directed to a virtual address. The CPU translates the virtual address to a physical address by using a set of page tables. To access an *I/O-mapped* device, on the other hand, the CPU invokes a special mechanism such as the x86 IN and OUT instructions.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 216

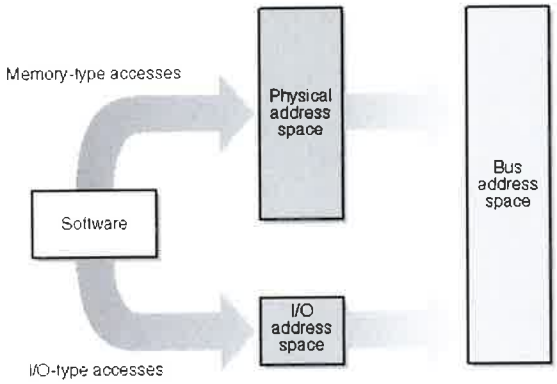


Figure 7-4. *Accessing ports and registers.*

Devices have bus-specific ways of decoding memory and I/O addresses. In the case of the PCI bus, a host bridge maps CPU physical memory addresses and I/O addresses to a bus address space that's directly accessible to devices. Flag bits in the device's configuration space determine whether the bridge maps the device's registers to a memory or an I/O address on CPUs that have both address spaces.

As I've said, some CPUs have separate memory and I/O address spaces. Intel architecture CPUs have both, for example. Other CPUs, such as the Alpha, have just a memory address space. If your device is I/O-mapped, the PnP Manager will give you port resources. If your device is memory-mapped, it will give you memory resources instead.

Rather than have you place reams of conditionally compiled code in your driver for all possible platforms, the Windows NT designers invented the hardware abstraction layer (HAL), to which I've alluded a few times in this book. The HAL provides functions that you use to access port and memory resources. See Table 7-3. As the table indicates, you can READ/WRITE either a single *UCHAR/USHORT/ULONG* or an array of them from or to a PORT/REGISTER. That makes 24 HAL functions in all that are used for device access. Since a WDM driver doesn't directly rely on the HAL for anything else, you might as well think of these 24 functions as being the entire public interface to the HAL.

Access Width	Functions for Port Access	Functions for Memory Access
8 bits	READ_PORT_UCHAR WRITE_PORT_UCHAR	READ_REGISTER_UCHAR WRITE_REGISTER_UCHAR
16 bits	READ_PORT_USHORT WRITE_PORT_USHORT	READ_REGISTER_USHORT WRITE_REGISTER_USHORT
32 bits	READ_PORT_ULONG WRITE_PORT_ULONG	READ_REGISTER_ULONG WRITE_REGISTER_ULONG
String of 8-bit bytes	READ_PORT_BUFFER_UCHAR WRITE_PORT_BUFFER_UCHAR	READ_REGISTER_BUFFER_UCHAR WRITE_REGISTER_BUFFER_UCHAR
String of 16-bit words	READ_PORT_BUFFER_USHORT WRITE_PORT_BUFFER_USHORT	READ_REGISTER_BUFFER_USHORT WRITE_REGISTER_BUFFER_USHORT
String of 32-bit doublewords	READ_PORT_BUFFER_ULONG WRITE_PORT_BUFFER_ULONG	READ_REGISTER_BUFFER_ULONG WRITE_REGISTER_BUFFER_ULONG

Table 7-3. *HAL Functions for Accessing Ports and Memory Registers*

What goes on inside these access functions is (obviously!) highly dependent on the platform. The Intel x86 version of *READ_PORT_CHAR*, for example, performs an IN instruction to read 1 byte from the designated I/O port. The Microsoft Windows 98/Me implementation goes so far as to overstore the driver's call instruction with an actual IN instruction in some situations. The Alpha version of this routine performs a memory fetch. The Intel x86 version of *READ_REGISTER_UCHAR* performs a memory fetch also; this function is macro'ed as a direct memory reference on the Alpha. The buffered version of this function (*READ_REGISTER_BUFFER_UCHAR*), on the other hand, does some extra work in the Intel x86 environment to ensure that all CPU caches are properly flushed when the operation finishes.

The whole point of having the HAL in the first place is so that you don't have to worry about platform differences or about the sometimes arcane requirements for accessing devices in the multitasking, multiprocessor environment of Windows XP. Your job is quite simple: use a PORT call to access what you think is a port resource, and use a REGISTER call to access what you think is a memory resource.

IPR2024-00292
KASPERSKY EX 1008 Page 217

7.3.1 Port Resources

I/O-mapped devices expose hardware registers that, on some CPU architectures (including Intel x86), are addressed by software using a special I/O address space. On other CPU architectures, no separate I/O address space exists, and these registers are addressed using regular memory references. Luckily, you don't need to understand these addressing complexities. If your device requests a port resource, one iteration of your loop over the translated resource descriptors will find a *CmResourceTypePort* descriptor, and you'll save three pieces of information.

```
typedef struct  DEVICE EXTENSION {
:
  PUCHAR portbase;
  ULONG nports;
  BOOLEAN mappedport;
:
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

PHYSICAL ADDRESS portbase;     // base address of range
:
for (ULONG i = 0; i < nres; ++i, ++resource)
  {
  switch (resource->Type)
    {
  case CmResourceTypePort:
1   portbase = resource->u.Port.Start;
    pdx->nports = resource->u.Port.Length;
2   pdx->mappedport =
     (resource->Flags & CM RESOURCE PORT IO) == 0;
    break;
:
    }
:
if (pdx->mappedport)
  {
3 pdx->portbase = (PUCHAR) MmMapIoSpace(portbase,
    pdx->nports, MmNonCached);
  if (!pdx->portbase)
    return STATUS NO MEMORY;
  }
else
4 pdx->portbase = (PUCHAR) portbase.QuadPart;
```

1. The resource descriptor contains a union named *u* that has substructures for each of the standard resource types. *u.Port* has information about a port resource. *u.Port.Start* is the beginning address of a contiguous range of I/O ports, and *u.Port.Length* is the number of ports in the range. The start address is a 64-bit *PHYSICAL_ADDRESS* value.

2. The *Flags* member of the resource descriptor for a port resource has the *CM_RESOURCE_PORT_IO* flag set if the CPU architecture has a separate I/O address space to which the given port address belongs.

3. If the *CM_RESOURCE_PORT_IO* flag was clear, as it will be on an Alpha and perhaps other RISC platforms, you must call *MmMapIoSpace* to obtain a kernel-mode virtual address by which the port can be accessed. The access will really employ a memory reference, but you'll still call the PORT flavor of HAL routines (*READ_PORT_UCHAR* and so on) from your driver.

4. If the *CM_RESOURCE_PORT_IO* flag was set, as it will be on an x86 platform, you do not need to map the port address. You'll call the PORT flavor of HAL routines from your driver when you want to access one of your ports. The HAL routines demand a *PUCHAR* port address argument, which is why we cast the base address to that type. The *QuadPart* reference, by the way, results in your getting a 32-bit or 64-bit pointer, as appropriate to the platform for which you're compiling.

Whether or not the port address needs to be mapped via *MmMapIoSpace*, you'll always call the HAL routines that deal with I/O port resources: *READ_PORT_UCHAR*, *WRITE_PORT_UCHAR*, and so on. On a CPU that requires you to map a port address, the HAL will be making memory references. On a CPU that doesn't require the mapping, the HAL will be making I/O references; on an x86, this means using one of the IN and OUT instruction family.

Your *StopDevice* helper routine has a small cleanup task to perform if you happen to have mapped your port resource:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 218

```
VOID StopDevice(...)
  {
  :
  if (pdx->portbase && pdx->mappedport)
    MmUnmapIoSpace(pdx->portbase, pdx->nports);
  pdx->portbase = NULL;
  :
  }
```

7.3.2 Memory Resources

Memory-mapped devices expose registers that software accesses using load and store instructions. The translated resource value you get from the PnP Manager is a physical address, and you need to reserve virtual addresses to cover the physical memory. Later on, you'll be calling HAL routines that deal with memory registers, such as *READ_REGISTER_UCHAR*, *WRITE_REGISTER_UCHAR*, and so on. Your extraction and configuration code will look like the fragment below.

```
typedef struct  DEVICE EXTENSION {
  :
  PUCHAR membase;
  ULONG nbytes;
  :
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

PHYSICAL ADDRESS membase;     // base address of range
  :
for (ULONG i = 0; i < nres; ++i, ++resource)
  {
  switch (resource->Type)
    {
  case CmResourceTypeMemory:
1
    membase = resource->u.Memory.Start;
    pdx->nbytes = resource->u.Memory.Length;
    break;
  :
    }
  :
2
pdx->membase = (PUCHAR) MmMapIoSpace(membase, pdx->nbytes,
  MmNonCached);
if (!pdx->membase)
  return STATUS_NO_MEMORY;
```

1. Within the resource descriptor, *u.Memory* has information about a memory resource. *u.Memory.Start* is the beginning address of a contiguous range of memory locations, and *u.Memory.Length* is the number of bytes in the range. The start address is a 64-bit *PHYSICAL_ADDRESS* value. It's not an accident that the *u.Port* and *u.Memory* substructures are identical—it's on purpose, and you can rely on it being true if you want to.

2. You must call *MmMapIoSpace* to obtain a kernel-mode virtual address by which the memory range can be accessed.

Your *StopDevice* function unconditionally unmaps your memory resources:

```
VOID StopDevice(...)
  {
  :
  if (pdx->membase)
    MmUnmapIoSpace(pdx->membase, pdx->nbytes);
  pdx->membase = NULL;
  :
  }
```

7.4 Servicing an Interrupt

Many devices signal completion of I/O operations by asynchronously interrupting the processor. In this section, I'll discuss how you configure your driver for interrupt handling and how you service interrupts when they occur.

IPR2024-00292
KASPERSKY EX 1008 Page 219

7.4.1 Configuring an Interrupt

You configure an interrupt resource in your *StartDevice* function by calling *IoConnectInterrupt* using parameters that you can simply extract from a *CmResourceTypeInterrupt* descriptor. Your driver and device need to be entirely ready to work correctly when you call *IoConnectInterrupt*—you might even have to service an interrupt before the function returns—so you normally make the call near the end of the configuration process. Some devices have a hardware feature that allows you to prevent them from interrupting. If your device has such a feature, disable interrupts before calling *IoConnectInterrupt* and enable the interrupts afterward. The extraction and configuration code for an interrupt would look like this:

```
typedef struct  DEVICE EXTENSION {
:
  PKINTERRUPT InterruptObject;
:
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

ULONG vector;              // interrupt vector
KIRQL irql;                // interrupt level
KINTERRUPT MODE mode;      // latching mode
KAFFINITY affinity;        // processor affinity
BOOLEAN irqshare;          // shared interrupt?
:
for (ULONG i = 0; i < nres; ++i, ++resource)
  {
  switch (resource->Type)
    {
  case CmResourceTypeInterrupt:
1
    irql = (KIRQL) resource->u.Interrupt.Level;
2
    vector = resource->u.Interrupt.Vector;
3
    affinity = resource->u.Interrupt.Affinity;
4
    mode = (resource->Flags == CM RESOURCE INTERRUPT LATCHED)
      ? Latched : LevelSensitive;
5
    irqshare = resource->ShareDisposition == CmResourceShareShared;
    break;
:
  }
:
status = IoConnectInterrupt(&pdx->InterruptObject,
  (PKSERVICE ROUTINE) OnInterrupt, (PVOID) pdx, NULL,  vector, irql, irql, mode,
irqshare, affinity, FALSE);
```

1. The *Level* parameter specifies the interrupt request level (IRQL) for this interrupt.
2. The *Vector* parameter specifies the hardware interrupt vector for this interrupt. We don't care what this number is because we're just going to act as a conduit between the PnP Manager and *IoConnectInterrupt*. All that matters is that the HAL understand what the number means.
3. *Affinity* is a bit mask that indicates which CPUs will be allowed to handle this interrupt.
4. We need to tell *IoConnectInterrupt* whether our interrupt is edge-triggered or level-triggered. If the resource *Flags* field is *CM_RESOURCE_INTERRUPT_LATCHED*, we have an edge-triggered interrupt. Otherwise, we have a level-triggered interrupt.
5. Use this statement to discover whether your interrupt is shared.

In the call to *IoConnectInterrupt* at the end of this sequence, we will simply regurgitate the values we pulled out of the interrupt resource descriptor. The first argument (*&pdx->InterruptObject*) indicates where to store the result of the connection operation—namely, a pointer to a kernel interrupt object that describes your interrupt. The second argument (*OnInterrupt*) is the name of your interrupt service routine; I'll discuss ISRs a bit further on in this chapter. The third argument (*pdx*) is a context value that will be passed as an argument to the ISR each time your device interrupts. I'll have more to say about this context parameter later as well in "Selecting an Appropriate Context Argument."

The fifth and sixth arguments (*vector* and *irql*) specify the interrupt vector number and interrupt request level, respectively, for the interrupt you're connecting. The eighth argument (*mode*) is either *Latched* or *LevelSensitive* to indicate whether the interrupt is edge-triggered or level-triggered. The ninth argument is *TRUE* if your interrupt is shared with other devices and *FALSE* otherwise. The tenth argument (*affinity*) is the processor affinity mask for this interrupt. The eleventh and final

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 220

argument indicates whether the operating system needs to save the floating-point context when the device interrupts. Because you're not allowed to do floating-point calculations in an ISR on an x86 platform, a portable driver will always set this flag to *FALSE*.

I haven't yet described two other arguments to *IoConnectInterrupt*. These become important when your device uses more than one interrupt. In such a case, you create a spin lock for your interrupts and initialize it by calling *KeInitializeSpinLock*. You also calculate the largest IRQL needed by any of your interrupts before connecting any of them. In each call to *IoConnectInterrupt*, you specify the address of the spin lock for the fourth argument (which is *NULL* in my example), and you specify the maximum IRQL for the seventh argument (which is *irql* in my example). This seventh argument indicates the IRQL used for synchronizing the interrupts, which you should make the maximum of all your interrupt IRQLs so that you're troubled by only one of your interrupts at a time.

If, however, your device uses only a single interrupt, you won't need a special spin lock (because the I/O Manager automatically allocates one for you), and the synchronization level for your interrupt will be the same as the interrupt IRQL.

7.4.2 Handling Interrupts

Your device can interrupt on any of the CPUs specified in the affinity mask you specify in your call to *IoConnectInterrupt*. When an interrupt occurs, the system raises the CPU's IRQL to the appropriate synchronization level and claims the spin lock associated with your interrupt object. Then it calls your ISR, which will have the following skeletal form:

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PVOID Context)
  {
  if (<device not interrupting>)
    return FALSE;
  <handle interrupt>
  return TRUE;
  }
```

Windows NT's interrupt-handling mechanism assumes that hardware interrupts can be shared by many devices. Thus your first job in the ISR is to determine whether your device is interrupting at the present moment. If not, you return *FALSE* right away so that the kernel can send the interrupt to another device driver. If so, you clear the interrupt at the device level and return *TRUE*. Whether the kernel then calls other drivers' ISRs depends on whether the device interrupt is edge-triggered or level-triggered and on other platform details.

Your main job in the ISR is to service your hardware to clear the interrupt. I'll have some general things to say about this job, but the details pretty much depend on how your hardware works. Once you've performed this major task, you return *TRUE* to indicate to the HAL that you've serviced a device interrupt.

Programming Restrictions in the ISR

ISRs execute at an IRQL higher than *DISPATCH_LEVEL*. All code and data used in an ISR must therefore be in nonpaged memory. Furthermore, the set of kernel-mode functions that an ISR can call is very limited.

Since an ISR executes at an elevated IRQL, it freezes out other activities on its CPU that require the same or a lower IRQL. For best system performance, therefore, your ISR should execute as quickly as is reasonably possible. Basically, do the minimum amount of work required to service your hardware and return. If there is additional work to do (such as completing an IRP), schedule a DPC to handle that work.

Despite the admonition you usually receive to do the smallest amount of work possible in your ISR, you don't want to carry that idea to the extreme. For example, if you're dealing with a device that interrupts to signal its readiness for the next output byte, go ahead and send the next byte directly from your ISR. It's fundamentally silly to schedule a DPC just to transfer a single byte. Remember that the end user wants you to service your hardware (or else he or she wouldn't have the hardware installed on the computer), and you're entitled to your fair share of system resources to provide that service.

But don't go crazy calculating pi to a thousand decimal places in your ISR, either (unless your device requires you to do something that ridiculous, and it probably doesn't). Good sense should tell you what the right balance of work between an ISR and a DPC routine should be.

Selecting an Appropriate Context Argument

In the call to *IoConnectInterrupt*, the third argument is an arbitrary context value that eventually shows up as the second argument to your ISR. You want to choose this argument so as to allow your ISR to execute as rapidly as possible; the address of your device object or of your device extension would be a good choice. The device extension is where you'll be storing items—such as your device's base port address—that you'll use in testing whether your device is currently asserting an interrupt. To illustrate, suppose that your device, which is I/O-mapped, has a status port at its base address and that the low-order bit of the status value indicates whether the device is currently trying to interrupt. If you adopt my suggestion, the first few lines of your ISR will read like this:

IPR2024-00292
KASPERSKY EX 1008 Page 221

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject,
  PDEVICE EXTENSION pdx)
  {
  UCHAR devstatus = READ PORT UCHAR(pdx->portbase);
  if ((devstatus & 1))
    return FALSE;
  <etc.>
  }
```

The fully optimized code for this function will require only a few instructions to read the status port and test the low-order bit.

TIP
If you have any say in the hardware design, arrange matters so that your status port returns a 0 bit to indicate the pendency of an interrupt. Reads from ports to which nothing is connected usually return a 1 bit, and this simple design choice can prevent an infinite loop of calls to your ISR.

If you elect to use the device extension as your context argument, be sure to supply a cast when you call *IoConnectInterrupt*:

```
IoConnectInterrupt(..., (PKSERVICE_ROUTINE) OnInterrupt, ...);
```

If you omit the cast, the compiler will generate an exceptionally obscure error message because the second argument to your *OnInterrupt* routine (a *PDEVICE_EXTENSION*) won't match the prototype of the function pointer argument to *IoConnectInterrupt*, which demands a *PVOID*.

Synchronizing Operations with the ISR

As a general rule, the ISR shares data and hardware resources with other parts of the driver. Anytime you hear the word *share*, you should immediately start thinking about synchronization problems. For example, a standard universal asynchronous receiver-transmitter (UART) device has a data port the driver uses for reading and writing data. You'd expect a serial port driver's ISR to access this port from time to time. Changing the baud rate also entails setting a control flag called the *divisor latch*, performing two single-byte write operations—one of which involves this same data port—and then clearing the divisor latch. If the UART were to interrupt in the middle of changing the baud rate, you can see that a data byte intended to be transmitted could easily end up in the baud-rate divisor register or that a byte intended for the divisor register could end up being transmitted as data.

The system guards the ISR with a spin lock and with a relatively high IRQL—the device IRQL (DIRQL). To simplify the mechanics of obtaining the same spin lock and raising IRQL to the same level as an interrupt, the system provides this service function:

```
BOOLEAN result = KeSynchronizeExecution(InterruptObject,  SynchRoutine, Context);
```

where *InterruptObject* (PKINTERRUPT) is a pointer to the interrupt object describing the interrupt we're trying to synchronize with, *SynchRoutine* (*PKSYNCHRONIZE_ROUTINE*) is the address of a callback function in our driver, and *Context* (*PVOID*) is an arbitrary context parameter to be sent to the *SynchRoutine* as an argument. We use the generic term *synch critical section routine* to describe a subroutine that we call by means of *KeSynchronizeExecution*. The synch critical section routine has the following prototype:

```
BOOLEAN SynchRoutine(PVOID Context);
```

That is, it receives a single argument and returns a *BOOLEAN* result. When it gets control, the current CPU is running at the synchronization IRQL that the original call to *IoConnectInterrupt* specified, and it owns the spin lock associated with the interrupt. Consequently, interrupts from the device are temporarily blocked out, and the *SynchRoutine* can freely access data and hardware resources that it shares with the ISR.

KeSynchronizeExecution returns whatever value *SynchRoutine* returns, by the way. This gives you a way of providing a little bit of feedback from *SynchRoutine* to whomever calls *KeSynchronizeExecution*.

If you're designing a driver to run only in Windows XP and later systems, you can use *KeAcquireInterruptSpinLock* and *KeReleaseInterruptSpinLock* to avoid the sometimes cumbersome coding that *KeSynchronizeExecution* requires.

7.4.3 Deferred Procedure Calls

Completely servicing a device interrupt often requires you to perform operations that aren't legal inside an ISR or that are too expensive to carry out at the elevated IRQL of an ISR. To avoid these problems, the designers of Windows NT provided the deferred procedure call mechanism. The DPC is a general-purpose mechanism, but you use it most often in connection with interrupt handling. In the most common scenario, your ISR decides that the current request is complete and requests a DPC. Later on, the kernel calls your DPC routine at *DISPATCH_LEVEL*. Although restrictions on the service routines you can call

IPR2024-00292
KASPERSKY EX 1008 Page 222

and on paging still apply, there are fewer such restrictions because you're now running at a lower IRQL than inside the ISR. In particular, it's legal to call routines such as *IoCompleteRequest* and *StartNextPacket* that are logically necessary at the end of an I/O operation.

Every device object gets a DPC object "for free." That is, the *DEVICE_OBJECT* has a DPC object—named, prosaically enough, *Dpc*—built in. You need to initialize this built-in DPC object shortly after you create your device object:

```
NTSTATUS AddDevice(...)
  {
  PDEVICE_OBJECT fdo;
  IoCreateDevice(..., &fdo);
  IoInitializeDpcRequest(fdo, DpcForIsr);
  :
  }
```

IoInitializeDpcRequest is a macro in WDM.H that initializes the device object's built-in DPC object. The second argument is the address of the DPC procedure that I'll show you presently.

With your initialized DPC object in place, your ISR can request a DPC by using the following macro:

```
BOOLEAN OnInterrupt(...)
  {
  :
  IoRequestDpc(pdx->DeviceObject, NULL, (PVOID) pdx);
  :
  }
```

This call to *IoRequestDpc* places your device object's DPC object in a systemwide queue, as illustrated in Figure 7-5.

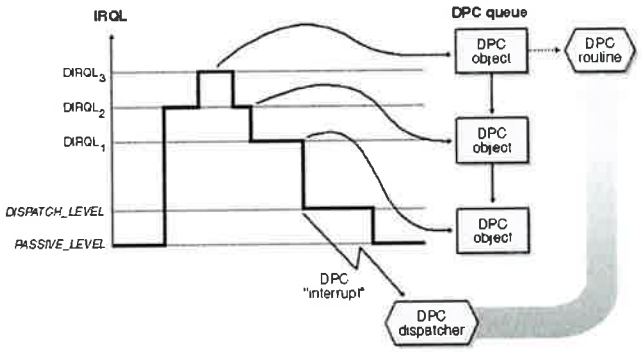


Figure 7-5. *Processing DPC requests.*

The *NULL* and *pdx* parameters are context values. Later on, when no other activity is occurring at *DISPATCH_LEVEL*, the kernel removes your DPC object from the queue and calls your DPC routine, which has the following prototype:

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
  {
  }
```

What you do inside the DPC routine depends in great measure on how your device works. A likely task would be to complete the current IRP and release the next IRP from the queue. If you use one of my *DEVQUEUE* objects for IRP queuing, the code will be as follows:

```
VOID DpcForIsr(...)
  {
  PIRP Irp = GetCurrentIrp(&pdx->dqRead);
  StartNextPacket(&pdx->dqRead, fdo);
  IoCompleteRequest(Irp, <boost value>);
  }
```

In this code fragment, we rely on the fact that the *DEVQUEUE* package remembers the IRP it sends to our *StartIo* routine. The IRP we want to complete is the one that's current when we commence the DPC routine. It's customary to call *StartNextPacket* before *IoCompleteRequest* so that we can get our device busy with a new request before we start the potentially long process of

IPR2024-00292
KASPERSKY EX 1008 Page 223

completing the current IRP.

DPC Scheduling

I've glossed over two fairly important details and a minor one about DPCs until now. The first important detail is implicit in the fact that you have a DPC *object* that gets put in a queue by *IoRequestDpc*. If your device generates an additional interrupt before the DPC routine actually runs, and if your ISR requests another DPC, the kernel will simply ignore the second request. In other words, your DPC object will be in the queue one time no matter how many DPCs are requested by successive invocations of your ISR, and the kernel will call your callback routine only once. During that one invocation, your DPC routine needs to accomplish all the work related to all the interrupts that have occurred since the last DPC.

As soon as the DPC dispatcher dequeues your DPC object, it's possible for somebody to queue it again, even while your DPC routine executes. This won't cause you any grief if the object happens to be queued on the same CPU both times. The second important detail about DPC processing, therefore, has to do with CPU affinity. Normally, the kernel queues a DPC object for handling on the same processor that requests the DPC—for example, the processor that just handled an interrupt and called *IoRequestDpc*. As soon as the DPC dispatcher dequeues the DPC object and calls your callback routine on one CPU, it's theoretically possible for your device to interrupt on a *different* CPU, which might end up requesting a DPC that could execute simultaneously on that different CPU. Whether simultaneous execution of your DPC routine poses a problem or not depends, obviously, on the details of your coding.

You can avoid the potential problems that might come from having your DPC routine simultaneously active on multiple CPUs in several ways. One way, which isn't the best, is to designate a particular CPU for running your DPC by calling *KeSetTargetProcessorDpc*. Also, you can theoretically restrict the CPU affinity of your interrupt when you first connect it; if you never queue the DPC except from your ISR, you'll never be executing the DPC on any different CPU. The real reason you'd specify the CPU affinity of a DPC or an interrupt, however, is to improve performance by allowing the code and data accessed during your DPC or ISR routines to remain in a cache.

You can also use a spin lock or other synchronization primitive to prevent interference between two instances of your DPC routine. Be careful of using a spin lock here: you often need to coordinate the hypothetical multiple instances of your DPC routine with your ISR, and an ISR runs at too high an IRQL to use an ordinary spin lock. An interlocked list—that is, one you manipulate by using support functions in the same family as *ExInterlockedInsertHeadList*—might help you because (so long as you never explicitly acquire the same spin lock that you use to guard the list) you can use the list at any IRQL. The *InterlockedOr*, *InterlockedAnd*, and *InterlockedXor* functions may also help by allowing you to manage a bit mask (such as a mask indicating recent interrupt conditions) that controls what your DPC routine is supposed to accomplish.

Most simply, you can just make sure that your device won't interrupt in between the time you request a DPC and the time your DPC routine finishes its work. ("Yo, hardware guys, stop flooding me with interrupts!")

The third DPC detail, which I consider less crucial than the two I've just explained, concerns the *importance* of the DPC. By calling *KeSetImportanceDpc*, you can designate one of three importance levels for your DPC:

- *MediumImportance* is the default and indicates that the DPC should be queued after all currently queued DPCs. If the DPC is queued to another processor, that other processor won't necessarily be interrupted right away to service the DPC. If it's queued to the current processor, the kernel will request a DPC interrupt as soon as possible to begin servicing DPCs.
- *HighImportance* causes the DPC to be queued first. If two or more high-importance DPCs get requested at about the same time, the last one queued gets serviced first.
- *LowImportance* causes the DPC to be queued last. In addition, the kernel won't necessarily request a DPC interrupt for whichever processor is destined to service the DPC.

The net effect of a DPC's importance level is to influence, but not necessarily control, how soon the DPC occurs. Even a DPC that has low importance might trigger a DPC interrupt on another CPU if that other CPU reaches some threshold for queued DPCs or if DPCs haven't been getting processed fast enough on it. If your device is capable of interrupting again before your DPC routine runs, changing your DPC to low importance will increase the likelihood that you'll have multiple work items to perform. If your DPC has an affinity for some CPU other than the one that requests the DPC, choosing high importance for your DPC will increase the likelihood that your ISR will still be active when your DPC routine begins to run. But neither of these possibilities is a certainty; conversely, altering or not altering your importance can't prevent either of them from happening.

Custom DPC Objects

You can create other DPC objects besides the one named *Dpc* in a device object. Simply reserve storage—in your device extension or some other persistent place that isn't paged—for a KDPC object, and initialize it:

```
typedef struct  DEVICE_EXTENSION {
:
  KDPC CustomDpc;
:
  };
```


IPR2024-00292
KASPERSKY EX 1008 Page 224

```
KeInitializeDpc(&pdx->CustomDpc,(PKDEFERRED_ROUTINE) DpcRoutine, fdo);
```

In the call to *KeInitializeDpc*, the second argument is the address of a DPC routine in nonpaged memory, and the third argument is an arbitrary context parameter that will be sent to the DPC routine as its second argument.

To request a deferred call to a custom DPC routine, call *KeInsertQueueDpc*:

```
BOOLEAN inserted = KeInsertQueueDpc(&pdx->CustomDpc, arg1, arg2);
```

Here *arg1* and *arg2* are arbitrary context pointers that will be passed to the custom DPC routine. The return value is *FALSE* if the DPC object was already in a processor queue and *TRUE* otherwise.

Also, you can remove a DPC object from a processor queue by calling *KeRemoveQueueDpc*.

7.4.4 A Simple Interrupt-Driven Device

I wrote the PCI42 sample driver (available in the companion content) to illustrate how to write the various driver routines that a typical interrupt-driven non-DMA device might use. The method used to handle such a device is often called programmed I/O (PIO) because program intervention is required to transfer each unit of data.

PCI42 is a dumbed-down driver for the S5933 PCI chip set from Applied Micro Circuits Corporation (AMCC). The S5933 acts as a matchmaker between the PCI bus and an add-on device that implements the actual function of a device. The S5933 is very flexible. In particular, you can program nonvolatile RAM so as to initialize the PCI configuration space for your device in any desired way. PCI42 uses the S5933 in its factory default state, however.

To grossly oversimplify matters, a WDM driver communicates with the add-on device connected to an S5933 either by doing DMA (which I'll discuss in the next major section of this chapter) or by sending and receiving data through a set of mailbox registers. PCI42 will be using 1 byte in one of the mailbox registers to transfer data 1 byte at a time.

The AMCC development kit for the S5933 (part number S5933DK1) includes two breadboard cards and an ISA (Industry Standard Architecture) interface card that connects to the S5933 development board via a ribbon cable. The ISA card allows you to access the S5933 from the add-on device side in order to provide software simulation of the add-on device. One component of the PCI42 sample is a driver (S5933DK1.SYS) for the ISA card that exports an interface for use by test programs.

Hardware people will snicker at the simplicity of the way PCI42 manages the device. The advantage of using such a trivial example is that you'll be able to see each step in the process of handling an I/O operation unfold at human speed. So chortle right back if your social dynamics allow it.

Initializing PCI42

The *StartDevice* function in PCI42 handles a port resource and an interrupt resource. The port resource describes a collection of sixteen 32-bit operation registers in I/O space, and the interrupt resource describes the host manifestation of the device's INTA# interrupt capability. At the end of *StartDevice*, we have the following device-specific code:

```
NTSTATUS StartDevice(...)
  {
  ResetDevice(pdx);
  status = IoConnectInterrupt(...);
  KeSynchronizeExecution(pdx->InterruptObject,
    (PKSYNCHRONIZE_ROUTINE) SetupDevice, pdx);
  return STATUS_SUCCESS;
  }
```

That is, we invoke a helper routine (*ResetDevice*) to reset the hardware. One of the tasks for *ResetDevice* is to prevent the device from generating any interrupts, insofar as that's possible. Then we call *IoConnectInterrupt* to connect the device interrupt to our ISR. Even before *IoConnectInterrupt* returns, it's possible for our device to generate an interrupt, so everything about our driver and the hardware has to be ready to go beforehand. After connecting the interrupt, we invoke another helper routine named *SetupDevice* to program the device to act the way we want it to. We must synchronize this step with our ISR because it uses the same hardware registers that our ISR would use, and we don't want any possibility of sending the device inconsistent instructions. The *SetupDevice* call is the last step in PCI42's *StartDevice* because—contrary to what I told you in Chapter 2—PCI42 hasn't registered any device interfaces and therefore has none to enable at this point.

ResetDevice is highly device-specific and reads as follows:

```
VOID ResetDevice(PDEVICE_EXTENSION pdx)
  {
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 225

```
    PAGED_CODE();

1
    WRITE_PORT_ULONG((PULONG) (pdx->portbase + MCSR), MCSR_RESET);

    LARGE_INTEGER timeout;
    timeout.QuadPart = -10 * 10000; // i.e., 10 milliseconds

2
    KeDelayExecutionThread(KernelMode, FALSE, &timeout);
    WRITE_PORT_ULONG((PULONG) (pdx->portbase + MCSR), 0);

3
    WRITE_PORT_ULONG((PULONG) (pdx->portbase + INTCSR),
      INTCSR_INTERRUPT_MASK);
    }
```

1. The S5933 has a master control/status register (MCSR) that controls bus-mastering DMA transfers and other actions. Asserting 4 of these bits resets different features of the device. I defined the constant *MCSR_RESET* to be a mask containing all four of these reset flags. This and other manifest constants for S5933 features are in the S5933.H file that's part of the PCI42 project.

2. Three of the reset flags pertain to features internal to the S5933 and take effect immediately. Setting the fourth flag to 1 asserts a reset signal for the add-on device. To deassert the add-on reset, you have to explicitly reset this flag to 0. In general, you want to give the hardware a little bit of time to recognize a reset pulse. *KeDelayExecutionThread*, which I discussed in Chapter 4, puts this thread to sleep for about 10 milliseconds. You can raise or lower this constant if your hardware has different requirements, but don't forget that the timeout will never be less than the granularity of the system clock. Since we're blocking our thread, we need to be running at *PASSIVE_LEVEL* in a nonarbitrary thread context. Those conditions are met because our ultimate caller is the PnP Manager, which has sent us an *IRP_MN_START_DEVICE* in the full expectation that we'd be blocking the system thread we happen to be in.

3. The last step in resetting the device is to clear any pending interrupts. The S5933 has six interrupt flags in an interrupt control/status register (INTCSR). Writing 1 bits in these six positions clears all pending interrupts. (If we write back a mask value that has a 0 bit in one of the interrupt flag positions, the state of that interrupt isn't affected. This kind of flag bit is called *read/write-clear* or just R/WC.) Other bits in the INTCSR enable interrupts of various kinds. By writing 0 bits in those locations, we're disabling the device to the maximum extent possible.

Our *SetupDevice* function is quite simple:

```
VOID SetupDevice(PDEVICE_EXTENSION pdx)
  {
  WRITE_PORT_ULONG((PULONG) (pdx->portbase + INTCSR),
    INTCSR_IMBI_ENABLE
    | (INTCSR_MB1 << INTCSR_IMBI_REG_SELECT_SHIFT)
    | (INTCSR_BYTE0 << INTCSR_IMBI_BYTE_SELECT_SHIFT)
    );
  }
```

This function reprograms the INTCSR to specify that we want an interrupt to occur when there's a change to byte 0 of inbound mailbox register 1. We could have specified other interrupt conditions for this chip, including the emptying of a particular byte of a specified outbound mailbox register, the completion of a read DMA transfer, and the completion of a write DMA transfer.

Starting a Read Operation

PCI42's *StartIo* routine follows the pattern we've already studied:

```
VOID StartIo(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
  {
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

  if (!stack->Parameters.Read.Length)
    {
    StartNextPacket(&pdx->dqReadWrite, fdo);
    CompleteRequest(Irp, STATUS_SUCCESS, 0);
    return;
    }

1
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 226

```
    pdx->buffer = (PUCHAR) Irp->AssociatedIrp.SystemBuffer;
    pdx->nbytes = stack->Parameters.Read.Length;
    pdx->numxfer = 0;

2   KeSynchronizeExecution(pdx->InterruptObject,
      (PKSYNCHRONIZE_ROUTINE) TransferFirst, pdx);
    }
```

1. Here we save parameters in the device extension to describe the ongoing progress of the input operation we're about to undertake. PCI42 uses the *DO_BUFFERED_IO* method, which isn't typical but helps make this driver simple enough to be used as an example.

2. Because our interrupt is connected, our device can interrupt at any time. The ISR will want to transfer data bytes when interrupts happen, but we want to be sure that the ISR is never confused about which data buffer to use or about the number of bytes we're trying to read. To restrain our ISR's eagerness, we put a flag in the device extension named *busy* that's ordinarily *FALSE*. Now is the time to set that flag to *TRUE*. As usual when dealing with a shared resource, we need to synchronize the setting of the flag with the code in the ISR that tests it, and we therefore need to invoke a *SynchCritSection* routine as I previously discussed. It might also happen that a data byte is already available, in which case the first interrupt will never happen. *TransferFirst* is a helper routine that checks for this eventuality and reads the first byte. The add-on function has ways of knowing that we emptied the mailbox, so it will presumably send the next byte in due course. Here's *TransferFirst*:

```
VOID TransferFirst(PDEVICE_EXTENSION pdx)
  {
  pdx->busy = TRUE;
  ULONG mbef = READ_PORT_ULONG((PULONG) (pdx->portbase + MBEF));
  if (!(mbef & MBEF_IN1_0))
    return;

  *pdx->buffer = READ_PORT_UCHAR(pdx->portbase + IMB1);
  ++pdx->buffer;
  ++pdx->numxfer;
  if (--pdx->nbytes == 0)
    {
    pdx->busy = FALSE;
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = pdx->numxfer;
    IoRequestDpc(pdx->DeviceObject, NULL, pdx);
    }
  }
```

The S5933 has a mailbox empty/full register (MBEF) whose bits indicate the current status of each byte of each mailbox register. Here we check whether the register byte we're using for input (inbound mailbox register 1, byte 0) is presently unread. If so, we read it. That might exhaust the transfer count. We already have a subroutine (*DpcForIsr*) that knows what to do with a complete request, so we request a DPC if this first byte turns out to satisfy the request. (Recall that we're executing at DIRQL under protection of an interrupt spin lock because we've been invoked as a *SynchCritSection* routine, so we can't just complete the IRP right now.)

Handling the Interrupt

In normal operation with PCI42, the S5933 interrupts when a new data byte arrives in mailbox 1. The following ISR then gains control:

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject,
  PDEVICE_EXTENSION pdx)
  {
1 ULONG intcsr =
    READ_PORT_ULONG((PULONG) (pdx->portbase + INTCSR));
  if (!(intcsr & INTCSR_INTERRUPT_PENDING))
    return FALSE;

  BOOLEAN dpc = FALSE;

2 PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
```


IPR2024-00292
KASPERSKY EX 1008 Page 227

```
3
  if (pdx->busy)
    {
    if (Irp->Cancel)
      status = STATUS_CANCELLED;
    else
      status = AreRequestsBeingAborted(&pdx->dqReadWrite);
    if (!NT_SUCCESS(status))
      dpc = TRUE, pdx->nbytes = 0;
    }

4
  while (intcsr & INTCSR_INTERRUPT_PENDING)
    {
5
    if (intcsr & INTCSR_IMBI)
      {
      if (pdx->nbytes && pdx->busy)
        {
        *pdx->buffer = READ_PORT_UCHAR(pdx->portbase + IMB1);
        ++pdx->buffer;
        ++pdx->numxfer;
        if (!--pdx->nbytes)
          {
          Irp->IoStatus.Information = pdx->numxfer;
          dpc = TRUE;
          status = STATUS_SUCCESS;
          }
        }
      }

6
    WRITE_PORT_ULONG((PULONG) (pdx->portbase + INTCSR), intcsr);

7
    intcsr = READ_PORT_ULONG((PULONG) (pdx->portbase + INTCSR));
    }

8
  if (dpc)
    {
    pdx->busy = FALSE;
    Irp->IoStatus.Status = status;
    IoRequestDpc(pdx->DeviceObject, NULL, NULL);
    }

  return TRUE;
  }
```

1. Our first task is to discover whether our own device is trying to interrupt now. We read the S5933's INTCSR and test a bit (*INTCSR_INTERRUPT_PENDING*) that summarizes all pending causes of interrupts. If this bit is clear, we return immediately. The reason I chose to use the device extension pointer as the context argument to this routine—back when I called *IoConnectInterrupt*—should now be clear: we need immediate access to this structure to get the base port address.

2. When we use a *DEVQUEUE*, we rely on the queue object to keep track of the current IRP. This interrupt might be one that we don't expect because we're not currently servicing any IRP. In that case, we still have to clear the interrupt but shouldn't do anything else.

3. It's also possible that a Plug and Play or power event has occurred that will cause any new IRPs to be rejected by the dispatch routine. The *DEVQUEUE*'s *AreRequestsBeingAborted* function tells us that fact so that we can abort the current request right now. Aborting an active request is a reasonable thing to do with a device such as this that proceeds byte by byte. Similarly, it's a good idea to check whether the IRP has been cancelled if it will take a long time to finish the IRP. If your device interrupts only when it's done with a long transfer, you can leave this test out of your ISR.

4. We're now embarking on a loop that will terminate when all of our device's current interrupts have been cleared. At the end of the loop, we'll reread the INTCSR to determine whether any more interrupt conditions have arisen. If so, we'll repeat the loop. We're not being greedy with CPU time here—we want to avoid letting interrupts cascade into the system because servicing an interrupt is by itself relatively expensive.

5. If the S5933 has interrupted because of a mailbox event, we'll read a new data byte from the mailbox into the I/O buffer

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 228

for the current IRP. If you were to look in the MBEF register immediately after the read, you'd see that the read clears the bit corresponding to inbound mailbox register 1, byte 0. Note that we needn't test the MBEF to determine whether our byte has actually changed because we programmed the device to interrupt only upon a change to that single byte.

6. Writing the INTCSR with its previous contents has the effect of clearing the six R/WC interrupt bits, not changing a few read-only bits, and preserving the original setting of all read/write control bits.

7. Here we read the INTCSR to determine whether additional interrupt conditions have arisen. If so, we'll repeat this loop to service them.

8. As we progressed through the preceding code, we set the *BOOLEANdpc* variable to *TRUE* if a DPC is now appropriate to complete the current IRP.

The DPC routine for PCI42 is as follows:

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE OBJECT fdo, PIRP junk,
  PDEVICE EXTENSION pdx)
  {
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  StartNextPacket(&pdx->dqReadWrite, fdo);
  IoCompleteRequest(Irp, IO NO INCREMENT);
  }
```

Testing PCI42

If you want to examine PCI42 in operation, you need to do several things. First obtain and install an S5933DK1 development board, including the ISA add-in interface card. Use the Add Hardware wizard to install the S5933DK1.SYS driver and the PCI42.SYS driver. (I found that Windows 98 initially identified the development board as a nonworking sound card and that I had to remove it in the Device Manager before I could install PCI42 as its driver. Windows XP handled the board normally.)

Then run *both* the ADDONSIM and TEST programs, which are in the PCI42 directory tree in the companion content. ADDONSIM writes a data value to the mailbox via the ISA interface. TEST reads a data byte from PCI42. Determining the value of the data byte is left as an exercise for you.

7.5 Direct Memory Access

Windows XP supports direct memory access transfers based on the abstract model of a computer depicted in Figure 7-6. In this model, the computer is considered to have a collection of *map registers* that translate between physical CPU addresses and bus addresses. Each map register holds the address of one physical page frame. Hardware accesses memory for reading or writing by means of a *logical*, or bus-specific, address. The map registers play the same role as page table entries for software by allowing hardware to use different numeric values for their addresses than the CPU understands.

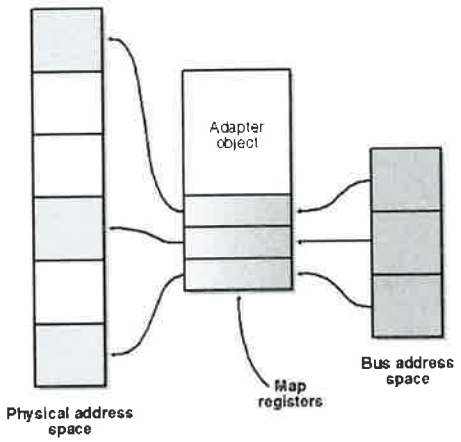


Figure 7-6. *Abstract computer model for DMA transfers.*

Some CPUs, such as the Alpha, have actual hardware map registers. One of the steps in initializing a DMA transfer—specifically, the *MapTransfer* step I'll discuss presently—reserves some of these registers for your use. Other CPUs, such as the Intel x86, don't have map registers, but you write your driver as if they did. The *MapTransfer* step on such a computer might end up reserving use of physical memory buffers that belong to the system, in which case the DMA operation will proceed using the reserved buffer. Obviously, somebody has to copy data to or from the DMA buffer before or after the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 229

transfer. In certain cases—for example, when dealing with a bus-master device that has scatter/gather capability—the *MapTransfer* phase might do nothing at all on an architecture without map registers.

The Windows XP kernel uses a data structure known as an *adapter object* to describe the DMA characteristics of a device and to control access to potentially shared resources, such as system DMA channels and map registers. You get a pointer to an adapter object by calling *IoGetDmaAdapter* during your *StartDevice* processing. The adapter object has a pointer to a structure named *DmaOperations* that, in turn, contains pointers to all the other functions you need to call. See Table 7 - 4. These functions take the place of global functions (such as *IoAllocateAdapter*, *IoMapTransfer*, and the like) that you would have used in previous versions of Windows NT. In fact, the global names are now macros that invoke the *DmaOperations* functions.

DmaOperations Function Pointer	Description
PutDmaAdapter	Destroys adapter object
AllocateCommonBuffer	Allocates a common buffer
FreeCommonBuffer	Releases a common buffer
AllocateAdapterChannel	Reserves adapter and map registers
FlushAdapterBuffers	Flushes intermediate data buffers after transfer
FreeAdapterChannel	Releases adapter object and map registers
FreeMapRegisters	Releases map registers only
MapTransfer	Programs one stage of a transfer
GetDmaAlignment	Gets address alignment required for adapter
ReadDmaCounter	Determines residual count
GetScatterGatherList	Reserves adapter and constructs scatter/gather list
PutScatterGatherList	Releases scatter/gather list

Table 7-4. *DmaOperations Function Pointers for DMA Helper Routines*

7.5.1 Transfer Strategies

How you perform a DMA transfer depends on several factors:

- If your device has bus-mastering capability, it has the necessary electronics to access main memory if you tell it a few basic facts, such as where to start, how many units of data to transfer, whether you're performing an input or an output operation, and so on. You'll consult with your hardware designers to sort out these details, or else you'll be working from a specification that tells you what to do at the hardware level.
- A device with scatter/gather capability can transfer large blocks of data to or from noncontiguous areas of physical memory. Using scatter/gather is advantageous for software because it eliminates the need to acquire large blocks of contiguous page frames. Pages can simply be locked wherever they're found in physical memory, and the device can be told where they are.
- If your device is not a bus master, you'll be using the system DMA controller on the motherboard of the computer. This style of DMA is sometimes called *slave DMA*. The system DMA controller associated with the ISA bus has some limitations on what physical memory it can access and how large a transfer it can perform without reprogramming. The controller for an Extended Industry Standard Architecture (EISA) bus lacks these limits. You won't have to know—at least, not in Windows XP—which type of bus your hardware plugs in to because the operating system is able to take account of these different restrictions automatically.
- Ordinarily, DMA operations involve programming hardware map registers or copying data either before or after the operation. If your device needs to read or write data continuously, you don't want to do either of these steps for each I/O request—they might slow down processing too much to be acceptable in your particular situation. You can, therefore, allocate what's known as a *common buffer*, which your driver and your device can simultaneously access at any time.

Notwithstanding the fact that many details will be different depending on how these four factors interplay, the steps you perform will have many common features. Figure 7-7 illustrates the overall operation of a transfer. You start the transfer in your *StartIo* routine by requesting ownership of your adapter object. Ownership has meaning only if you're sharing a system DMA channel with other devices, but the Windows XP DMA model demands that you perform this step anyway. When the I/O Manager is able to grant you ownership, it allocates some map registers for your temporary use and calls back to an *adapter control* routine you provide. In your adapter control routine, you perform a *transfer mapping* step to arrange the first (maybe the only) stage of the transfer. Multiple stages can be necessary if sufficient map registers aren't available; your device must be capable of handling any delay that might occur between stages.

IPR2024-00292
KASPERSKY EX 1008 Page 230

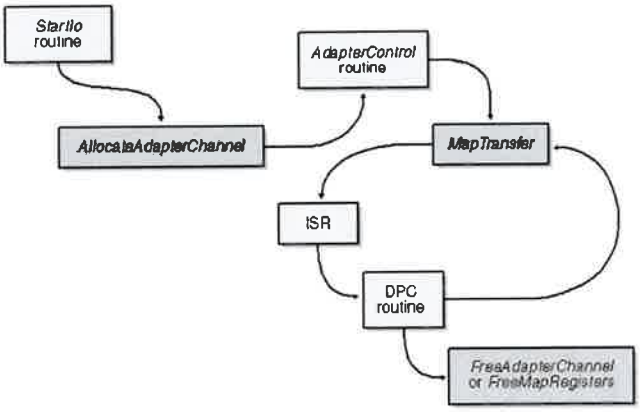


Figure 7-7. *Flow of ownership during DMA.*

Once your adapter control routine has initialized the map registers for the first stage, you signal your device to begin operation. Your device will instigate an interrupt when this initial transfer completes, whereupon you'll schedule a DPC. The DPC routine will initiate another staged transfer if necessary, or else it will complete the request.

Somewhere along the way, you'll release the map registers and the adapter object. The timing of these two events is one of the details that differ based on the factors I summarized earlier in this section.

7.5.2 Performing DMA Transfers

Now I'll go into detail about the mechanics of what's often called a *packet-based* DMA transfer, wherein you transfer a discrete amount of data by using the data buffer that accompanies an I/O request packet. Let's start simply and suppose that you face what will be a very common case nowadays: your device is a PCI bus master but doesn't have scatter/gather capability.

To start with, when you create your device object, you'll ordinarily indicate that you want to use the direct method of data buffering by setting the *DO_DIRECT_IO* flag. You choose the direct method because you'll eventually be passing the address of a memory descriptor list as one of the arguments to the *MapTransfer* function you'll be calling. This choice poses a bit of a problem with regard to buffer alignment, though. Unless the application uses the *FILE_FLAG_NO_BUFFERING* flag in its call to *CreateFile*, the I/O Manager won't enforce the device object's *AlignmentRequirement* on user-mode data buffers. (It doesn't enforce the requirement for a kernel-mode caller at all except in the checked build.) If your device or the HAL requires DMA buffers to begin on some particular boundary, therefore, you might end up copying a small portion of the user data to a correctly aligned internal buffer to meet the alignment requirement—either that or cause to fail any request that has a misaligned buffer.

In your *StartDevice* function, you create an adapter object by using code like the following:

```
DEVICE_DESCRIPTION dd;
RtlZeroMemory(&dd, sizeof(dd));
dd.Version = DEVICE_DESCRIPTION_VERSION;
dd.Master = TRUE;
dd.InterfaceType = InterfaceTypeUndefined;
dd.MaximumLength = MAXTRANSFER;
dd.Dma32BitAddresses = TRUE;

pdx->AdapterObject = IoGetDmaAdapter(pdx->Pdo, &dd, &pdx->nMapRegisters);
```

The last statement in this code fragment is the important one. *IoGetDmaAdapter* will communicate with the bus driver or the HAL to create an adapter object, whose address it returns to you. The first parameter (*pdx->Pdo*) identifies the physical device object (PDO) for your device. The second parameter points to a *DEVICE_DESCRIPTION* structure that you initialize to describe the DMA characteristics of your device. The last parameter indicates where the system should store the maximum number of map registers you'll ever be allowed to attempt to reserve during a single transfer. You'll notice that I reserved two fields in the device extension (*AdapterObject* and *nMapRegisters*) to receive the two outputs from this function.

TIP
If you specify *InterfaceTypeUndefined* for the *InterfaceType* member of the *DEVICE_DESCRIPTION* structure, the I/O Manager will internally query the bus driver to find out what type of bus your device happens to be connected to. This relieves you of the burden of hard-coding the bus type or calling *IoGetDeviceProperty* to determine it yourself.

IPR2024-00292
KASPERSKY EX 1008 Page 231

In your *StopDevice* function, you destroy the adapter object with this call:

```
VOID StopDevice(...)
  {
  .
  .
  .
  if (pdx->AdapterObject)
    (*pdx->AdapterObject->DmaOperations->PutDmaAdapter)
      (pdx->AdapterObject);
  pdx->AdapterObject = NULL;
  .
  .
  .
  }
```

☑ You can request DMA verification in the Driver Verifier settings. If you do, the verifier will make sure that you follow the correct protocol, as described here, all the way from creating the adapter object through finally deleting it with a call to *PutDmaAdapter*. If you're porting a driver from Windows NT version 4, expect to encounter several verifier bug checks as you switch to using the newer protocol in Windows XP.

You won't expect to receive an official DMA resource when your device is a bus master. That is, your resource extraction loop won't need a *CmResourceTypeDma* case label. The PnP Manager doesn't assign you a DMA resource because your hardware itself contains all the necessary electronics for performing DMA transfers, so nothing additional needs to be assigned to you.

Previous versions of Windows NT relied on a service function named *HalGetAdapter* to acquire the DMA adapter object. That function still exists for compatibility, but new WDM drivers should call *IoGetDmaAdapter* instead. The difference between the two is that *IoGetDmaAdapter* first issues an *IRP_MN_QUERY_INTERFACE* Plug and Play IRP to determine whether the physical device object supports the *GUID_BUS_INTERFACE_STANDARD* direct call interface. If so, *IoGetDmaAdapter* uses that interface to allocate the adapter object. If not, it simply calls *HalGetAdapter*.

Table 7-5 summarizes the fields in the *DEVICE_DESCRIPTION* structure you pass to *IoGetDmaAdapter*. The only fields that are relevant for a bus-master device are those shown in the preceding *StartDevice* code fragment. The HAL might or might not need to know whether your device recognizes 32-bit or 64-bit addresses—the Intel x86 HAL uses this flag only when you allocate a common buffer or when the machine employs Physical Memory Extensions (PME), for example—but you should indicate that capability anyway to retain portability. By zeroing the entire structure, we set *ScatterGather* to *FALSE*. Since we won't be using a system DMA channel, none of *DmaChannel*, *DmaPort*, *DmaWidth*, *DemandMode*, *AutoInitialize*, *IgnoreCount*, and *DmaSpeed* will be examined by the routine that creates our adapter object.

Field Name	Description	Relevant to Device
Version	Version number of structure—initialize to *DEVICE_DESCRIPTION_VERSION*	All
Master	Bus-master device—set based on your knowledge of device	All
ScatterGather	Device supports scatter/gather list—set based on your knowledge of device	All
DemandMode	Use system DMA controller's demand mode—set based on your knowledge of device	Slave
AutoInitialize	Use system DMA controller's autoinitialize mode—set based on your knowledge of device	Slave
Dma32BitAddresses	Can use 32-bit physical addresses	All
IgnoreCount	Controller doesn't maintain an accurate transfer count—set based on your knowledge of device	Slave
Reserved1	Reserved—must be *FALSE*	
Dma64BitAddresses	Can use 64-bit physical addresses	All
DoNotUse2	Reserved—must be 0	
DmaChannel	DMA channel number—initialize from Channel attribute of resource descriptor	Slave
InterfaceType	Bus type—initialize to *InterfaceTypeUndefined*	All
DmaWidth	Width of transfers—set based on your knowledge of device to Width8Bits, *Width16Bits*, or *Width32Bits*	Slave
DmaSpeed	Speed of transfers—set based on your knowledge of device to Compatible, *TypeA*, *TypeB*, *TypeC*, or *TypeF*	Slave
MaximumLength	Maximum length of a single transfer—set based on your knowledge of device (and round up to a multiple of *PAGE_SIZE*)	All
DmaPort	*Microchannel*-type bus port number—initialize from Port attribute of resource descriptor	Slave

Table 7-5. *Device Description Structure Used with IoGetDmaAdapter*

To initiate an I/O operation, your *StartIo* routine first has to reserve the adapter object by calling the object's

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 232

AllocateAdapterChannel routine. One of the arguments to *AllocateAdapterChannel* is the address of an adapter control routine that the I/O Manager will call when the reservation has been accomplished. Here's an example of code you would use to prepare and execute the call to *AllocateAdapterChannel*:

```
typedef struct  DEVICE EXTENSION {
:
1
   PADAPTER OBJECT AdapterObject; // device's adapter object
   ULONG nMapRegisters; // max # map registers
   ULONG nMapRegistersAllocated; // # allocated for this xfer
   ULONG numxfer;       // # bytes transferred so far
   ULONG xfer;          // # bytes to transfer during this stage
   ULONG nbytes;        // # bytes remaining to transfer
   PVOID vaddr;         // virtual address for current stage
   PVOID regbase;       // map register base for this stage
:
   } DEVICE EXTENSION, *PDEVICE EXTENSION;

VOID StartIo(PDEVICE OBJECT fdo, PIRP Irp)
   {
   PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;

2
   PMDL mdl = Irp->MdlAddress;
   pdx->numxfer = 0;
   pdx->xfer = pdx->nbytes = MmGetMdlByteCount(mdl);
   pdx->vaddr = MmGetMdlVirtualAddress(mdl);

3
   ULONG nregs = ADDRESS AND SIZE TO SPAN PAGES(pdx->vaddr, pdx->nbytes);
   if (nregs > pdx->nMapRegisters)
     {
     nregs = pdx->nMapRegisters;
     pdx->xfer = nregs * PAGE SIZE - MmGetMdlByteOffset(mdl);
     }
   pdx->nMapRegistersAllocated = nregs;

4
   NTSTATUS status = (*pdx->AdapterObject->DmaOperations
     ->AllocateAdapterChannel)(pdx->AdapterObject, fdo, nregs,
     (PDRIVER CONTROL) AdapterControl, pdx);
   if (!NT SUCCESS(status))
     {
     CompleteRequest(Irp, status, 0);
     StartNextPacket(&pdx->dqReadWrite, fdo);
     }
   }
```

1. Your device extension needs several fields related to DMA transfers. The comments indicate the uses for these fields.

2. These few statements initialize fields in the device extension for the first stage of the transfer.

3. Here we calculate how many map registers we'll ask the system to reserve for our use during this transfer. We begin by calculating the number required for the entire transfer. The *ADDRESS_AND_SIZE_TO_SPAN_PAGES* macro takes into account that the buffer might span a page boundary. The number we end up with might, however, exceed the maximum allowed us by the original call to *IoGetDmaAdapter*. In that case, we need to perform the transfer in multiple stages. We therefore scale back the first stage so as to use only the allowable number of map registers. We also need to remember how many map registers we're allocating (in the *nMapRegistersAllocated* field of the device extension) so that we can release exactly the right number later on.

4. In this call to *AllocateAdapterChannel*, we specify the address of the adapter object, the address of our own device object, the calculated number of map registers, and the address of our adapter control procedure. The last argument (*pdx*) is a context parameter for the adapter control procedure.

In general, several devices can share a single adapter object. Adapter object sharing happens in real life only when you rely on the system DMA controller; bus-master devices own dedicated adapter objects. But because you don't need to know how the system decides when to create adapter objects, you shouldn't make any assumptions about it. In general, then, the adapter object might be busy when you call *AllocateAdapterChannel*, and your request might therefore be put in a queue until the adapter object becomes available. Also, all DMA devices on the computer share a set of map registers. Further delay can ensue until the requested number of registers becomes available. Both of these delays occur inside *AllocateAdapterChannel*, which calls your adapter control procedure when the adapter object and all the map registers you asked for are available.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 233

Even though a PCI bus-mastering device owns its own adapter object, if the device doesn't have scatter/gather capability, it requires the use of map registers. On CPUs such as Alpha that have map registers, *AllocateAdapterChannel* will reserve them for your use. On CPUs such as Intel that don't have map registers, *AllocateAdapterChannel* will reserve use of a software surrogate, such as a contiguous area of physical memory.

What Gets Queued in *AllocateAdapterChannel*?
The object that AllocateAdapterChannel puts in queues to wait for the adapter object or the necessary number of map registers is your device object. Some device architectures allow you to perform more than one DMA transfer simultaneously. Since you can put only one device object in an adapter object queue at a time (without crashing the system, that is), you need to create dummy device objects to take advantage of that multiple-DMA capability.

As I've been discussing, *AllocateAdapterChannel* eventually calls your adapter control routine (at *DISPATCH_LEVEL*, just as your *StartIo* routine does). You have two tasks to accomplish. First you should call the adapter object's *MapTransfer* routine to prepare the map registers and other system resources for the first stage of your I/O operation. In the case of a bus-mastering device, *MapTransfer* will return a logical address that represents the starting point for the first stage. This logical address might be the same as a CPU physical memory address, and it might not be. All you need to know about it is that it's the right address to program into your hardware. *MapTransfer* might also trim the length of your request to fit the map registers it's using, which is why you need to supply the address of the variable that contains the current stage length as an argument.

Your second task is to perform whatever device-dependent steps are required to inform your device of the physical address and to start the operation on your hardware:

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
  PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
  {
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
1
  PMDL mdl = Irp->MdlAddress;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
2
  BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;
3
  pdx->regbase = regbase;
4
  KeFlushIoBuffers(mdl, isread, TRUE);
5
  PHYSICAL_ADDRESS address =
    (*pdx->AdapterObject->DmaOperations->MapTransfer)
    (pdx->AdapterObject, mdl, regbase, pdx->vaddr, pdx->xfer,
    !isread);
6
  .
  .
7
  return DeallocateObjectKeepRegisters;
  }
```

1. The second argument—which I named *junk*—to *AdapterControl* is whatever was in the *CurrentIrp* field of the device object when you called *AllocateAdapterChannel*. When you use a *DEVQUEUE* for IRP queuing, you need to ask the *DEVQUEUE* object which IRP is current. If you use the Microsoft queuing routines *IoStartPacket* and *IoStartNextPacket* to manage the queue, *junk* would be the right IRP. In that case, I'd have named it *Irp* instead.

2. There are few differences between code to handle input and output operations using DMA, so it's often convenient to handle both operations in a single subroutine. This line of code examines the major function code for the IRP to decide whether a read or write is occurring.

3. The *regbase* argument to this function is an opaque handle that identifies the set of map registers that have been reserved for your use during this operation. You'll need this value later, so you should save it in your device extension.

4. *KeFlushIoBuffers* makes sure that the contents of all processor memory caches for the memory buffer you're using are flushed to memory. The third argument (*TRUE*) indicates that you're flushing the cache in preparation for a DMA operation. The CPU architecture might require this step because, in general, DMA operations proceed directly to or from memory without necessarily involving the caches.

5. The *MapTransfer* routine programs the DMA hardware for one stage of a transfer and returns the physical address where the transfer should start. Notice that you supply the address of an MDL as the second argument to this function. Because you need an MDL at this point, you would ordinarily have opted for the *DO_DIRECT_IO* buffering method when you first created your device object, and the I/O Manager would therefore have automatically created the MDL for you. You also pass along the map register base address (*regbase*). You indicate which portion of the MDL is involved in this stage of the operation by supplying a virtual address (*pdx->vaddr*) and a byte count (*pdx->xfer*). *MapTransfer* will use the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 234

virtual address argument to calculate an offset into the buffer area, from which it can determine the physical page numbers containing your data.

6. This is the point at which you program your hardware in the device-specific way that is required. You might, for example, use one of the *WRITE_Xxx* HAL routines to send the physical address and byte count values to registers on your card, and you might thereafter strobe a command register to begin transferring data.

7. We return the constant *DeallocateObjectKeepRegisters* to indicate that we're done using the adapter object but are still using the map registers. In this particular example (PCI bus master), there will never be any contention for the adapter object in the first place, so it hardly matters that we've released the adapter object. In other bus-mastering situations, though, we might be sharing a DMA controller with other devices. Releasing the adapter object allows those other devices to begin transfers by using a disjoint set of map registers from the ones we're still using.

An interrupt usually occurs shortly after you start the transfer, and the interrupt service routine usually requests a DPC to deal with completion of the first stage of the transfer. Your DPC routine will look something like this:

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE OBJECT fdo,
  PIRP junk, PDEVICE EXTENSION pdx)
  {
1
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  PMDL mdl = Irp->MdlAddress;
  BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)
    ->MajorFunction == IRP MJ READ;
2
  (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)
    (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
3
  pdx->nbytes -= pdx->xfer;
  pdx->numxfer += pdx->xfer;
  NTSTATUS status = STATUS SUCCESS;
  :
4
  if (pdx->nbytes && NT SUCCESS(status))
    {
5
    pdx->vaddr = (PVOID) ((PUCHAR) pdx->vaddr + pdx->xfer);
    pdx->xfer = pdx->nbytes;
6
    ULONG nregs = ADDRESS AND SIZE TO SPAN PAGES(pdx->vaddr, pdx->nbytes);
    if (nregs > pdx->nMapRegistersAllocated)
      {
      nregs = pdx->nMapRegistersAllocated;
      pdx->xfer = nregs * PAGE SIZE;
      }
    PHYSICAL ADDRESS address =
      (*pdx->AdapterObject->DmaOperations->MapTransfer)
      (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
    :
    }
  else
    {
    ULONG numxfer = pdx->numxfer;
7
    (*pdx->AdapterObject->DmaOperations->FreeMapRegisters)
      (pdx->AdapterObject, pdx->regbase,
      pdx->nMapRegistersAllocated);
8
    StartNextPacket(&pdx->dqReadWrite, fdo);
    CompleteRequest(Irp, status, numxfer);
    }
  }
```

1. When you use a *DEVQUEUE* for IRP queuing, you rely on the queue object to keep track of the current IRP.

2. The *FlushAdapterBuffers* routine handles the situation in which the transfer required use of intermediate buffers owned by the system. If you've done an input operation that spanned a page boundary, the input data is now sitting in an intermediate buffer and needs to be copied to the user-mode buffer.

3. Here we update the residual and cumulative data counts after the transfer stage that just completed.

IPR2024-00292
KASPERSKY EX 1008 Page 235

4. At this point, you determine whether the current stage of the transfer completed successfully or with an error. You might, for example, read a status port or inspect the results of a similar operation performed by your interrupt routine. In this example, I set the *status* variable to *STATUS_SUCCESS* with the expectation that you'd change it if you discovered an error here.

5. If the transfer hasn't finished yet, you need to program another stage. The first step in this process is to calculate the virtual address of the next portion of the user-mode buffer. Bear in mind that this calculation is merely working with a number—we're not actually trying to access memory by using this virtual address. Accessing the memory would be a bad idea, of course, because we're currently executing in an arbitrary thread context.

6. The next few statements are almost identical to the ones we performed in the first stage for *StartIo* and *AdapterControl*. The end result will be a logical address that can be programmed into your device. It might or might not correspond to a physical address as understood by the CPU. One slight wrinkle is that we're constrained to use only as many map registers as were allocated by the adapter control routine; *StartIo* saved that number in the *nMapRegistersAllocated* field of the device extension.

7. If the entire transfer is now complete, we need to release the map registers we've been using.

8. The remaining few statements in the DPC routine handle the mechanics of completing the IRP that got us here in the first place.

Transfers Using Scatter/Gather Lists

If your hardware has scatter/gather support, the system has a much easier time doing DMA transfers to and from your device. The scatter/gather capability permits the device to perform a transfer involving pages that aren't contiguous in physical memory.

Your *StartDevice* routine creates its adapter object in just about the same way I've already discussed, except (of course) that you'll set the *ScatterGather* flag to *TRUE*.

The traditional method—that is, the method you would have used in previous versions of Windows NT—to program a DMA transfer involving scatter/gather functionality is practically identical to the packet-based example considered in the section on Performing DMA Transfers. The only difference is that instead of making one call to *MapTransfer* for each stage of the transfer, you need to make multiple calls. Each call gives you the information you need for a single element in a *scatter/gather list* that contains a physical address and length. When you're done with the loop, you can send the scatter/gather list to your device by using some device-specific method, and you can then initiate the transfer.

I'm going to make some assumptions about the framework into which you'll fit the construction of a scatter/gather list. First, I'll assume that you've defined a manifest constant named *MAXSG* that represents the maximum number of scatter/gather list elements your device can handle. To make life as simple as possible, I'm also going to assume that you can just use the *SCATTER_GATHER_LIST* structure defined in WDM.H to construct the list:

```
typedef struct  SCATTER GATHER ELEMENT {
  PHYSICAL ADDRESS Address;
  ULONG Length;
  ULONG PTR Reserved;
  } SCATTER GATHER ELEMENT, *PSCATTER GATHER ELEMENT;

typedef struct  SCATTER GATHER LIST {
  ULONG NumberOfElements;
  ULONG PTR Reserved;
  SCATTER GATHER ELEMENT Elements[];
  } SCATTER_GATHER_LIST, *PSCATTER_GATHER_LIST;
```

Finally, I'm going to suppose that you can simply allocate a maximum-size scatter/gather list in your *AddDevice* function and leave it lying around for use whenever you need it:

```
pdx->sglist = (PSCATTER GATHER LIST)
  ExAllocatePool(NonPagedPool, sizeof(SCATTER GATHER LIST) +
  MAXSG * sizeof(SCATTER_GATHER_ELEMENT));
```

With this infrastructure in place, your *AdapterControl* procedure will look like this:

```
IO ALLOCATION ACTION AdapterControl(PDEVICE OBJECT fdo,
  PIRP junk, PVOID regbase, PDEVICE EXTENSION pdx)
{
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  PMDL mdl = Irp->MdlAddress;
  BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)
```


IPR2024-00292
KASPERSKY EX 1008 Page 236

```
    ->MajorFunction == IRP_MJ_READ;
  pdx->regbase = regbase;
  KeFlushIoBuffers(mdl, isread, TRUE);
  PSCATTER_GATHER_LIST sglist = pdx->sglist;

2
  ULONG xfer = pdx->xfer;
  PVOID vaddr = pdx->vaddr;
  pdx->xfer = 0;
  ULONG isg = 0;
3
  while (xfer && isg < MAXSG)
    {
    ULONG elen = xfer;

4
    sglist->Elements[isg].Address =
      (*pdx->AdapterObject->DmaOperations->MapTransfer)
      (pdx->AdapterObject, mdl, regbase, pdx->vaddr, &elen, !isread);
    sglist->Elements[isg].Length = elen;

5
    xfer -= elen;
    pdx->xfer += elen;
    vaddr = (PVOID) ((PUCHAR) vaddr + elen);
6
    ++isg;
    }
  sglist->NumberOfElements = isg;
7

8
  return DeallocateObjectKeepRegisters;
  }
```

1. See the earlier discussion of how to get a pointer to the correct IRP in an adapter control procedure.

2. We previously (in *StartIo*) calculated *pdx->xfer* based on the allowable number of map registers. We're going to try to transfer that much data now, but the allowable number of scatter/gather elements might further limit the amount we can transfer during this stage. During the following loop, *xfer* will be the number of bytes that we haven't yet mapped, and we'll recalculate *pdx->xfer* as we go.

3. Here's the loop I promised you, where we call *MapTransfer* to construct scatter/gather elements. We'll continue the loop until we've mapped the entire stage of this transfer or until we run out of scatter/gather elements, whichever happens first.

4. When we call *MapTransfer* for a scatter/gather device, it will modify the length argument (*elen*) to indicate how much of the MDL starting at the given virtual address (*vaddr*) is physically contiguous and can therefore be mapped by a single scatter/gather list element. It will also return the physical address of the beginning of the contiguous region.

5. Here's where we update the variables that describe the current stage of the transfer. When we leave the loop, *xfer* will be down to 0 (or else we'll have run out of scatter/gather elements), *pdx->xfer* will be up to the total of all the elements we were able to map, and *vaddr* will be up to the byte after the last one we mapped. We don't update the *pdx->vaddr* field in the device extension—we're doing that in our DPC routine. Just another one of those pesky details....

6. Here's where we increment the scatter/gather element index to reflect the fact that we've just used one up.

7. At this point, we have *isg* scatter/gather elements that we should program into our device in whatever hardware-dependent way is appropriate. Then we should start the device working on the request.

8. Returning *DeallocateObjectKeepRegisters* is appropriate for a bus-mastering device. You can theoretically have a nonmaster device with scatter/gather capability, and it would return *KeepObject* instead.

Your device now performs its DMA transfer and, presumably, interrupts to signal completion. Your ISR requests a DPC, and your DPC routine initiates the next stage in the operation. The DPC routine will perform a *MapTransfer* loop like the one I just showed you as part of that initiation process. I'll leave the details of that code as an exercise for you.

Using *GetScatterGatherList*

Windows 2000 and Windows XP provide a shortcut to avoid the relatively cumbersome loop of calls to *MapTransfer* in the common case in which you can accomplish the entire transfer by using either no map registers or no more than the maximum number of map registers returned by *IoGetDmaAdapter*. The shortcut, which is illustrated in the SCATGATH sample in the companion content, involves calling the *GetScatterGatherList* routine instead of *AllocateAdapterChannel*. Your *StartIo* routine

IPR2024-00292
KASPERSKY EX 1008 Page 237

looks like this:

```
VOID StartIo(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  NTSTATUS status;
  PMDL mdl = Irp->MdlAddress;
  ULONG nbytes = MmGetMdlByteCount(mdl);
  PVOID vaddr = MmGetMdlVirtualAddress(mdl);
  BOOLEAN isread = stack->MajorFunction == IRP MJ READ;
  pdx->numxfer = 0;
  pdx->nbytes = nbytes;
  status =
    (*pdx->AdapterObject->DmaOperations->GetScatterGatherList)
    (pdx->AdapterObject, fdo, mdl, vaddr, nbytes,
    (PDRIVER LIST CONTROL) DmaExecutionRoutine, pdx, !isread);
  if (!NT SUCCESS(status))
    {
    CompleteRequest(Irp, status, 0);
    StartNextPacket(&pdx->dqReadWrite, fdo);
    }
  }
```

The call to *GetScatterGatherList*, shown in bold in the preceding code fragment, is the main difference between this *StartIo* routine and the one we looked at in the preceding section. *GetScatterGatherList* waits if necessary until you can be granted use of the adapter object and all the map registers you need. Then it builds a *SCATTER_GATHER_LIST* structure and passes it to the *DmaExecutionRoutine*. You can then program your device by using the physical addresses in the scatter/gather elements and initiate the transfer:

```
VOID DmaExecutionRoutine(PDEVICE OBJECT fdo, PIRP junk,
  PSCATTER GATHER LIST sglist, PDEVICE EXTENSION pdx)
  {
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
1
  pdx->sglist = sglist;
2
  :
  }
```

1. You'll need the address of the scatter/gather list in the DPC routine, which will release the list by calling *PutScatterGatherList*.

2. At this point, program your device to do a read or write using the address and length pairs in the scatter/gather list. If the list has more elements than your device can handle at one time, you'll need to perform the whole transfer in stages. If you can program a stage fairly quickly, I'd recommend adding logic to your interrupt service routine to initiate the additional stages. If you think about it, your *DmaExecutionRoutine* is probably going to be synchronizing with your ISR anyway to start the first stage, so this extra logic is probably not large. I programmed the SCATGATH sample with this idea in mind.

When the transfer finishes, call the adapter object's *PutScatterGatherList* to release the list and the adapter:

```
VOID DpcForIsr(PKDPC Dpc, PDEVICE OBJECT fdo, PIRP junk,
  PVOID Context)
  {
  :
  (*pdx->AdapterObject->DmaOperations->PutScatterGatherList)
    (pdx->AdapterObject, pdx->sglist, !isread);
  :
  }
```

To decide whether you can use *GetScatterGatherList*, you need to be able to predict whether you'll meet the preconditions for its use. First of all, your driver will have to run on Windows 2000 or a later system only because this function isn't available in Windows 98/Me. On an Intel 32-bit platform, scatter/gather devices on a PCI or EISA bus can be sure of not needing any map registers. Even on an ISA bus, you'll be allowed to request up to 16 map register surrogates (8 if you're also a bus-mastering device) unless physical memory is so tight that the I/O system can't allocate its intermediate I/O buffers. In that case, you won't be able to do DMA using the traditional method either, so there's no point in worrying about it.

If you can't predict with certainty at the time you code your driver that you'll be able to use *GetScatterGatherList*, my advice is to just fall back on the traditional loop of *MapTransfer* calls. You'll need to put that code in place anyway to deal with cases

IPR2024-00292
KASPERSKY EX 1008 Page 238

in which *GetScatterGatherList* won't work, and having two pieces of logic in your driver is just unnecessary complication.

Transfers Using the System Controller

If your device is not a bus master, DMA capability requires that it use the system DMA controller. As I've said, people often use the phrase *slave DMA*, which emphasizes that such a device is not master of its own DMA fate. The system DMA controllers have several characteristics that affect the internal details of how DMA transfers proceed:

- There are a limited number of DMA *channels* that all slave devices must share. *AllocateAdapterChannel* has real meaning in a sharing situation because only one device can be using a particular channel at a time.
- You can expect to find a *CmResourceTypeDma* resource in the list of I/O resources delivered to you by the PnP Manager.
- Your hardware is wired, either physically or logically, to the particular channel it uses. If you can configure the DMA channel connection, you'll need to send the appropriate commands at *StartDevice* time.
- The system DMA controllers for an ISA bus computer are able to access data buffers in only the first 16 megabytes of physical memory. Four channels for transferring data 8 bits at a time and three channels for transferring data 16 bits at a time exist. The controller for 8-bit channels doesn't correctly handle a buffer that crosses a 64-KB boundary; the controller for 16-bit channels doesn't correctly handle a buffer that crosses a 128-KB boundary.

Notwithstanding these factors, your driver code will be similar to the bus-mastering code we've just discussed. Your *StartDevice* routine just works a little harder to set up its call to *IoGetDmaAdapter*, and your *AdapterControl* and DPC routines apportion the steps of releasing the adapter object and map registers differently.

In *StartDevice*, you have a little bit of additional code to determine which DMA channel the PnP Manager has assigned for you, and you also need to initialize more of the fields of the *DEVICE_DESCRIPTION* structure for *IoGetDmaAdapter*:

```
NTSTATUS StartDevice(...)
  {
  ULONG dmachannel;    // system DMA channel #
  ULONG dmaport;       // MCA bus port number

  for (ULONG i = 0; i < nres; ++i, ++resource)
    {
    switch (resource->Type)
      {
    case CmResourceTypeDma:
1     dmachannel = resource->u.Dma.Channel;
      dmaport = resource->u.Dma.Port;
      break;
      }
    }

  DEVICE DESCRIPTION dd;
  RtlZeroMemory(&dd, sizeof(dd));
  dd.Version = DEVICE DESCRIPTION VERSION;
  dd.InterfaceType = InterfaceTypeUndefined;
  dd.MaximumLength = MAXTRANSFER;

2 dd.DmaChannel = dmachannel;
  dd.DmaPort = dmaport;
  dd.DemandMode = ??;
  dd.AutoInitialize = ??;
  dd.IgnoreCount = ??;
  dd.DmaWidth = ??;
  dd.DmaSpeed = ??;

  pdx->AdapterObject = IoGetDmaAdapter(...);
  }
```

1. The I/O resource list will have a DMA resource, from which you need to extract the channel and port numbers. The channel number identifies one of the DMA channels supported by a system DMA controller. The port number is relevant only on a Micro Channel Architecture (MCA)-bus machine.
2. Beginning here, you have to initialize several fields of the *DEVICE_DESCRIPTION* structure based on your knowledge of your device. See Table 7-5.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 239

Everything about your adapter control and DPC procedures will be identical to the code we looked at earlier for handling a bus-mastering device without scatter/gather capability except for two small details. First, *AdapterControl* returns a different value:

```
IO_ALLOCATION_ACTION AdapterControl(...)
  {
  .
  .
  return KeepObject;
  }
```

The return value *KeepObject* indicates that we want to retain control over the map registers *and* the DMA channel we're using. Second, since we didn't release the adapter object when *AdapterControl* returned, we have to do so in the DPC routine by calling *FreeAdapterChannel* instead of *FreeMapRegisters*:

```
VOID DpcForIsr(...)
  {
  .
  .
  (*pdx->AdapterObject->DmaOperations->FreeAdapterChannel)
    (pdx->AdapterObject);
  .
  .
  }
```

7.5.3 Using a Common Buffer

As I mentioned in "Transfer Strategies," you might want to allocate a common buffer for your device to use in performing DMA transfers. A common buffer is an area of nonpaged, physically contiguous memory. Your driver uses a fixed virtual address to access the buffer. Your device uses a fixed logical address to access the same buffer.

You can use the common buffer area in several ways. You can support a device that continuously transfers data to or from memory by using the system DMA controller's autoinitialize mode. In this mode of operation, completion of one transfer triggers the controller to immediately reinitialize for another transfer.

Another use for a common buffer area is as a means to avoid extra data copying. The *MapTransfer* routine often copies the data you supply into auxiliary buffers owned by the I/O Manager and used for DMA. If you're stuck with doing slave DMA on an ISA bus, it's especially likely that *MapTransfer* will copy data to conform to the 16-MB address and buffer alignment requirements of the ISA DMA controller. But if you have a common buffer, you'll avoid the copy steps.

Allocating a Common Buffer

You'd normally allocate your common buffer at *StartDevice* time after creating your adapter object:

```
typedef struct  DEVICE_EXTENSION {
  .
  .
  PVOID vaCommonBuffer;
  PHYSICAL_ADDRESS paCommonBuffer;
  .
  .
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;

dd.Dma32BitAddresses = ??;
dd.Dma64BitAddresses = ??;
pdx->AdapterObject = IoGetDmaAdapter(...);
pdx->vaCommonBuffer =
  (*pdx->AdapterObject->DmaOperations->AllocateCommonBuffer)
  (pdx->AdapterObject, <length>, &pdx->paCommonBuffer, FALSE);
```

Prior to calling *IoGetDmaAdapter*, you set the *Dma32BitAddresses* and *Dma64BitAddresses* flags in the *DEVICE_DESCRIPTION* structure to state the truth about your device's addressing capabilities. That is, if your device can address a buffer using any 32-bit physical address, set *Dma32BitAddresses* to *TRUE*. If it can address a buffer using any 64-bit physical address, set *Dma64BitAddresses* to *TRUE*.

In the call to *AllocateCommonBuffer*, the second argument is the byte length of the buffer you want to allocate. The fourth argument is a *BOOLEAN* value that indicates whether you want the allocated memory to be capable of entry into the CPU cache (*TRUE*) or not (*FALSE*).

AllocateCommonBuffer returns a virtual address. This address is the one you use within your driver to access the allocated buffer area. *AllocateCommonBuffer* also sets the *PHYSICAL_ADDRESS* pointed to by the third argument to be the logical address used by your device for its own buffer access.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 240

NOTE
The DDK carefully uses the term *logical address* to refer to the address value returned by *MapTransfer* and the address value returned by the third argument of *AllocateCommonBuffer*. On many CPU architectures, a logical address will be a physical memory address that the CPU understands. On other architectures, it might be an address that only the I/O bus understands. Perhaps *bus address* would have been a better term.

Slave DMA with a Common Buffer

If you're going to be performing slave DMA, you must create an MDL to describe the virtual addresses you receive. The actual purpose of the MDL is to occupy an argument slot in an eventual call to *MapTransfer*. *MapTransfer* won't end up doing any copying, but it requires the MDL to discover that it doesn't need to do any copying! You'll normally create the MDL in your *StartDevice* function just after allocating the common buffer:

```
pdx->vaCommonBuffer = ...;
pdx->mdlCommonBuffer = IoAllocateMdl(pdx->vaCommonBuffer,
  <length>, FALSE, FALSE, NULL);
MmBuildMdlForNonPagedPool(pdx->mdlCommonBuffer);
```

To perform an output operation, first make sure by some means (such as an explicit memory copy) that the common buffer contains the data you want to send to the device. The other DMA logic in your driver will be essentially the same as I showed you earlier (in "Performing DMA Transfers"). You'll call *AllocateAdapterChannel*. It will call your adapter control routine, which will call *KeFlushIoBuffers*—if you allocated a cacheable buffer—and then call *MapTransfer*. Your DPC routine will call *FlushAdapterBuffers* and *FreeAdapterChannel*. In all of these calls, you'll specify the common buffer's MDL instead of the one that accompanied the read or write IRP you're processing. Some of the service routines you call won't do as much work when you have a common buffer as when you don't, but you must call them anyway. At the end of an input operation, you might need to copy data out of your common buffer to some other place.

To fulfill a request to read or write more data than fits in your common buffer, you might need to periodically refill or empty the buffer. The adapter object's *ReadDmaCounter* function allows you to determine the progress of the ongoing transfer to help you decide what to do.

Bus-Master DMA with a Common Buffer

If your device is a bus master, allocating a common buffer allows you to dispense with calling *AllocateAdapterChannel*, *MapTransfer*, and *FreeMapRegisters*. You don't need to call those routines because *AllocateCommonBuffer* also reserves the map registers, if any, needed for your device to access the buffer. Each bus-master device has an adapter object that isn't shared with other devices and for which you therefore need never wait. Because you have a virtual address you can use to access the buffer at any time, and because your device's bus-mastering capability allows it to access the buffer by using the physical address you've received back from *AllocateCommonBuffer*, no additional work is required.

Cautions About Using Common Buffers

A few cautions are in order with respect to common buffer allocation and usage. Physically contiguous memory is scarce in a running system—so scarce that you might not be able to allocate the buffer you want unless you stake your claim quite early in the life of a new session. The Memory Manager makes a limited effort to shuffle memory pages around to satisfy your request, and that process can delay the return from *AllocateCommonBuffer* for a period of time. But the effort might fail, and you must be sure to handle the failure case. Not only does a common buffer tie up potentially scarce physical pages, but it can also tie up map registers that could otherwise be used by other devices. For both these reasons, you should use a common-buffer strategy advisedly.

Another caution about common buffers arises from the fact that the Memory Manager necessarily gives you one or more full pages of memory. Allocating a common buffer that's just a few bytes long is wasteful and should be avoided. On the other hand, it's also wasteful to allocate several pages of memory that don't actually need to be physically contiguous. As the DDK suggests, therefore, it's better to make several requests for smaller blocks if the blocks don't have to be contiguous.

Releasing a Common Buffer

You would ordinarily release the memory occupied by your common buffer in your *StopDevice* routine just before you destroy the adapter object:

```
(*pdx->AdapterObject->DmaOperations->FreeCommonBuffer)
  (pdx->AdapterObject, <length>, pdx->paCommonBuffer, pdx->vaCommonBuffer, FALSE);
```

The second parameter to *FreeCommonBuffer* is the same length value you used when you allocated the buffer. The last parameter indicates whether the memory is cacheable, and it should be the same as the last argument you used in the call to *AllocateCommonBuffer*.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 241

7.5.4 A Simple Bus-Master Device

The PKTDMA sample driver in the companion content illustrates how to perform bus-master DMA operations without scatter/gather support using the AMCC S5933 PCI matchmaker chip. I've already discussed details of how this driver initializes the device in *StartDevice* and how it initiates a DMA transfer in *StartIo*. I've also discussed nearly all of what happens in this driver's *AdapterControl* and *DpcForIsr* routines. I indicated earlier that these routines would have some device-dependent code for starting an operation on the device; I wrote a helper function named *StartTransfer* for that purpose:

```
VOID StartTransfer(PDEVICE_EXTENSION pdx,
  PHYSICAL_ADDRESS address, BOOLEAN isread)
  {
  ULONG mcsr = READ_PORT_ULONG((PULONG)(pdx->portbase + MCSR);
  ULONG intcsr =
    READ_PORT_ULONG((PULONG)(pdx->portbase + INTCSR);
  if (isread)
    {
    mcsr |= MCSR_WRITE_NEED4 | MCSR_WRITE_ENABLE;
    intcsr |= INTCSR_WTCI_ENABLE;
1
    WRITE_PORT_ULONG((PULONG)(pdx->portbase + MWTC), pdx->xfer);
    WRITE_PORT_ULONG((PULONG)(pdx->portbase + MWAR),
      address.LowPart);
    }
  else
    {
    mcsr |= MCSR_READ_NEED4 | MCSR_READ_ENABLE;
    intcsr |= INTCSR_RTCI_ENABLE;
1
    WRITE_PORT_ULONG((PULONG)(pdx->portbase + MRTC), pdx->xfer);
    WRITE_PORT_ULONG((PULONG)(pdx->portbase + MRAR),
      address.LowPart);
    }
2
  WRITE_PORT_ULONG((PULONG)(pdx->portbase + INTCSR), intcsr);
3
  WRITE_PORT_ULONG((PULONG)(pdx->portbase + MCSR), mcsr);
  }
```

This routine sets up the S5933 operations registers for a DMA transfer and then starts the transfer running. The steps in the process are as follows:

1. Program the address (*MxAR*) and transfer count (*MxTC*) registers appropriate to the direction of data flow. AMCC chose to use the term *read* to describe an operation in which data moves from memory to the device. Therefore, when we're implementing an *IRP_MJ_WRITE*, we program a read operation at the chip level. The address we use is the logical address returned by *MapTransfer*.
2. Enable an interrupt when the transfer count reaches 0 by writing to the INTCSR.
3. Start the transfer by setting one of the transfer-enable bits in the MCSR.

It's not obvious from this fragment of code, but the S5933 is actually capable of doing a DMA read and a DMA write at the same time. I wrote PKTDMA in such a way that only one operation (either a read or a write) can be occurring. To generalize the driver to allow both kinds of operation to occur simultaneously, you would need to (a) implement separate queues for read and write IRPs and (b) create *two* device objects and *two* adapter objects—one pair for reading and the other for writing—so as to avoid the embarrassment of trying to queue the same object twice inside *AllocateAdapterChannel*. I thought putting that additional complication into the sample would end up confusing you. (I know I'm being pretty optimistic about my expository skills to imply that I haven't *already* confused you, but it could have been worse.)

Handling Interrupts in PKTDMA

PCI42 included an interrupt routine that did a small bit of work to move some data. PKTDMA's interrupt routine is a little simpler:

```
BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PDEVICE_EXTENSION pdx)
  {
  ULONG intcsr = READ_PORT_ULONG((PULONG) (pdx->portbase + INTCSR));
  if (!(intcsr & INTCSR_INTERRUPT_PENDING))
    return FALSE;
  ULONG mcsr = READ_PORT_ULONG((PULONG) (pdx->portbase + MCSR));
```


IPR2024-00292
KASPERSKY EX 1008 Page 242

```
1
  WRITE_PORT_ULONG((PULONG) (pdx->portbase + MCSR),
    mcsr & ~(MCSR_WRITE_ENABLE | MCSR_READ_ENABLE));

2
  intcsr &= ~(INTCSR_WTCI_ENABLE | INTCSR_RTCI_ENABLE);

  BOOLEAN dpc = GetCurrentIrp(&pdx->dqReadWrite) != NULL;

  while (intcsr & INTCSR_INTERRUPT_PENDING)
    {
    InterlockedOr(&pdx->intcsr, intcsr);
    WRITE_PORT_ULONG((PULONG) (pdx->portbase + INTCSR), intcsr);
    intcsr = READ_PORT_ULONG((PULONG) (pdx->portbase + INTCSR));
    }

  if (dpc)
    IoRequestDpc(pdx->DeviceObject, NULL, NULL);

  return TRUE;
  }
```

I'll discuss only the ways in which this ISR differs from the one in PCI42:

1. The S5933 will keep trying to transfer data—subject to the count register, that is—so long as the enable bits are set in the MCSR. This statement clears both bits. If your driver were handling simultaneous reads and writes, you'd determine which kind of operation had just finished by testing the interrupt flags in the INTCSR, and then you'd disable just the transfer in that direction.

2. We'll shortly write back to the INTCSR to clear the interrupt. This statement ensures that we'll also *disable* the transfer-count-0 interrupts so that they can't occur anymore. Once again, a driver that handles simultaneous reads and writes would disable only the interrupt that just occurred.

Testing PKTDMA

You can test PKTDMA if you have an S5933DK1 development board. If you ran the PCI42 test, you already installed the S5933DK1.SYS driver to handle the ISA add-on interface card. If not, you'll need to install that driver for this test. Then install PKTDMA.SYS as the driver for the S5933 development board itself. You can then run the TEST.EXE test program that's in the PKTDMA\TEST\DEBUG directory. TEST will perform a write for 8192 bytes to PKTDMA. It will also issue a *DeviceIoControl* to S5933DK1 to read the data back from the add-on side, and it will verify that it read the right values.

7.6 Windows 98/Me Compatibility Notes

MmGetSystemAddressForMdlSafe is a macro that invokes a function (*MmMapLockedPagesSpecifyCache*) that Windows 98/Me doesn't export. The older macro, *MmGetSystemAddressForMdl*, is now deprecated. The Driver Verifier will flag a runtime call to the older macro. The difference between the two is that *MmGetSystemAddressForMdl* will bug check if there aren't enough page table entries to map the specified memory, whereas *MmGetSystemAddressForMdlSafe* will simply return a NULL pointer.

There is a portable workaround to the problem posed by *MmGetSystemAddressForMdlSafe*:

```
CSHORT oldfail = mdl->MdlFlags & MDL_MAPPING_CAN_FAIL;
mdl->MdlFlags |= MDL_MAPPING_CAN_FAIL;
PVOID address = MmMapLockedPages(mdl, KernelMode);
if (!oldfail)
  mdl->MdlFlags &= ~MDL_MAPPING_CAN_FAIL;
```

Setting the *MDL_MAPPING_CAN_FAIL* flag causes Windows 2000 and XP to take the same internal code path as does *MmMapLockedPagesSpecifyCache*, thereby fulfilling the spirit of the injunction to use the new macro. Windows 98/Me ignore the flag (and they've always returned NULL in the failure case anyway, so there was never a need for the flag or the new macro).

If you're using my GENERIC.SYS, simply call *GenericGetSystemAddressForMdl*, which contains the foregoing code. I did not attempt to add *MmMapLockedPagesSpecifyCache* to WDMSTUB.SYS (see Appendix A) because Windows 98/Me doesn't provide all the infrastructure needed to fully support that function.

IPR2024-00292
KASPERSKY EX 1008 Page 243

Chapter 8

Power Management

Technophobes may take solace in the fact that they retain ultimate control over their electronic servants so long as they control the power switch. Power is, of course, the sine qua non of computing, but personal computers haven't done an especially good job of managing it until quite recently.

More effective power management is important for at least three reasons. First, as a matter of sound ecology, using less power helps minimize the impact of computing on the environment. Not only do computers require power, but so do the air-conditioning systems for the rooms where the computers reside. A second reason better power management is needed is familiar to many travelers: battery technology simply hasn't kept pace with the demand for mobile computing of all kinds. And finally, greater consumer acceptance of PCs as home appliances depends on improving power management. Current machines have noisy fans and squealing disk drives when they're on, and they take a long time to start up from the power-off state. Decreasing the power-up latency and eliminating unnecessary noise—which also means minimizing power consumption so that less cooling is required—will be necessary before PCs can comfortably occupy consumer niches.

In this chapter, I'll discuss the role WDM drivers play in power management in the MicrosoftWindows XP and Microsoft Windows 98/Me operating systems. The first major section of the chapter, "The WDM Power Model," presents an overview of the concepts you need to know about. The second section, "Managing Power Transitions," is the meat of the chapter: I'll describe there the very complicated tasks a typical function driver carries out. I'll finish the chapter with a discussion of some ancillary responsibilities a WDM function driver has with respect to power management.

8.1 The WDM Power Model

In Windows XP and Windows 98/Me, the operating system takes over most of the job of managing power. This makes sense because only the operating system really knows what's going on, of course. A system BIOS charged with power management, for example, can't tell the difference between an application's use of the screen and a screen saver's. But the operating system can tell the difference and thus can determine whether it's OK to turn off the display.

As the global *power policy owner* for the computer, the operating system supports user interface elements that give the end user ultimate control over power decisions. These elements include the control panel, commands in the Start menu, and APIs for controlling device wake-up features. The Power Manager component of the kernel implements the operating system's power policies by sending I/O request packets (IRPs) to devices. WDM drivers have the primarily passive role of responding to these IRPs, although you'll probably find this passivity to incorporate a lot of active motion when I show you how much code is involved.

8.1.1 The Roles of WDM Drivers

One of the drivers for a device acts as the power policy owner for the device. Since the function driver most often fills this role, I'll continue discussing power management as though that were invariably the case. Just bear in mind that your device might have unique requirements that mandate giving the responsibilities of policy owner to a filter driver or to the bus driver instead.

The function driver receives IRPs (system IRPs) from the Power Manager that pertain to changes in the overall power state of the system. Acting as policy owner for the device, it translates these instructions into device terms and originates new IRPs (device IRPs). When responding to the device IRPs, the function driver worries about the details that pertain to the device. Devices might carry onboard *context information* that you don't want to lose during a period of low power. Keyboard drivers, for example, might hold the state of locking keys (such as Caps Lock, Num Lock, and Scroll Lock), LEDs, and so on. The function driver is responsible for saving and restoring that context. Some devices have a *wake-up feature* that allows them to wake up a sleeping system when external events occur; the function driver works together with the end user to make sure that the wake-up feature is available when needed. Many function drivers manage queues of substantive IRPs—that is, IRPs that read or write data to the device, and they need to stall or release those queues as power wanes and waxes.

The bus driver at the bottom of the device stack is responsible for controlling the flow of current to your device and for performing the electronic steps necessary to arm or disarm your device's wake-up feature.

A filter driver normally acts as a simple conduit for power requests, passing them down to lower-level drivers by using the special protocol I'll describe a bit further on.

8.1.2 Device Power and System Power States

The Windows Driver Model uses the same terms to describe power states as does the Advanced Configuration and Power Interface (ACPI) specification. (See http://www.acpi.info.) Devices can assume the four states illustrated in Figure 8-1. In the

IPR2024-00292
KASPERSKY EX 1008 Page 245

D0 state, the device is fully functional. In the D3 state, the device is using no (or minimal) power and is therefore not functioning (or is functioning at a very low level). The intermediate D1 and D2 states denote two different somnolent states for the device. As a device moves from D0 to D3, it consumes less and less power. In addition, it remembers less and less *context* information about its current state. Consequently, the *latency* period needed for the device's transition back to D0 increases.

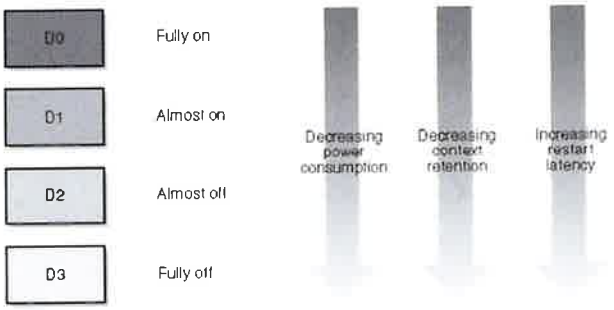


Figure 8-1. *ACPI device power states.*

Microsoft has formulated class-specific requirements for different types of devices. I found these requirements on line at http://www.microsoft.com/hwdev/resources/specs/PMref/. The specifications mandate, for example, that every device support at least the D0 and D3 states. Input devices (keyboards, mice, and so on) should also support the D1 state. Modem devices, moreover, should additionally support D2. These differences in specifications for device classes stem from likely usage scenarios and industry practice.

The operating system doesn't deal directly with the power states of devices—that's exclusively the province of device drivers. Rather, the system controls power by using a set of system power states that are analogous to the ACPI device states. See Figure 8-2. The Working state is the full-power, fully functional state of the computer. Programs are able to execute only when the system is in the Working state.

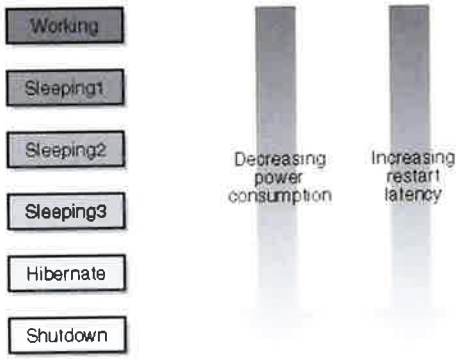


Figure 8-2. *System power states.*

The other system power states correspond to reduced power configurations in which no instructions execute. The Shutdown state is the power-off state. (Discussing the Shutdown state seems like discussing an unanswerable question such as, "What's inside a black hole?" Like the event horizon surrounding a black hole, though, the *transition* to Shutdown is something you'll need to know about as your device spirals in.) The Hibernate state is a variant of Shutdown in which the entire state of the computer is recorded on disk so that a live session can be restarted when power comes back. The three sleeping states between Hibernate and Working encompass gradations in power consumption.

8.1.3 Power State Transitions

The system initializes in the Working state. This almost goes without saying because the computer is, by definition, in the Working state whenever it's executing instructions. Most devices start out in the D0 state, although the policy owner for the device might put it into a lower power state when it's not actually in use. After the system is up and running, then, it reaches a steady state in which the system power level is Working and devices are in various states depending on activity and capability.

End user actions and external events cause subsequent transitions between power states. A common transition scenario arises when the user uses the Stand By command in the Turn Off Computer dialog box to put the machine into standby. In response, the Power Manager first asks each driver whether the prospective loss of power will be OK by sending an *IRP_MJ_POWER* request with the minor function code *IRP_MN_QUERY_POWER*. If all drivers acquiesce, the Power Manager sends a second power IRP with the minor function code *IRP_MN_SET_POWER*. Drivers put their devices into lower power states in response

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 246

to this second IRP. If any driver vetoes the query, the Power Manager still sends an *IRP_MN_SET_POWER* request, but it usually specifies the *current* power level instead of the one originally proposed.

The system doesn't always send *IRP_MN_QUERY_POWER* requests, by the way. Some events (such as the end user unplugging the computer or the battery expiring) must be accepted without demur, and the operating system won't issue a query when they occur. But when a query is issued, and when a driver accepts the proposed state change by passing the request along, the driver undertakes that it won't start any operation that might interfere with the expected set-power request. A tape driver, for example, will make sure that it's not currently retensioning a tape—the interruption of which might break the tape—before allowing a query for a low-power state to succeed. In addition, the driver will reject any subsequent retension command until (and unless) a countervailing set-power request arrives to signal abandonment of the state change.

8.1.4 Handling *IRP_MJ_POWER* Requests

The Power Manager communicates with drivers by means of an *IRP_MJ_POWER* I/O request packet. Four minor function codes are currently possible. See Table 8-1.

Minor Function Code	Description
IRP_MN_QUERY_POWER	Determines whether prospective change in power state can safely occur
IRP_MN_SET_POWER	Instructs driver to change power state
IRP_MN_WAIT_WAKE	Instructs bus driver to arm wake-up feature; provides way for function driver to know when wake-up signal occurs
IRP_MN_POWER_SEQUENCE	Provides optimization for context saving and restoring

Table 8-1. *Minor Function Codes for IRP_MJ_POWER*

The *Power* substructure in the *IO_STACK_LOCATION*'s *Parameters* union has four parameters that describe the request, of which only two will be of interest to most WDM drivers. See Table 8-2.

Field Name	Description
SystemContext	A context value used internally by the Power Manager
Type	DevicePowerState or SystemPowerState (values of POWER_STATE_TYPE enumeration)
State	Power state—either a DEVICE_POWER_STATE enumeration value or a SYSTEM_POWER_STATE enumeration value
ShutdownType	A code indicating the reason for a transition to PowerSystemShutdown

Table 8-2. *Fields in the Parameters.Power Substructure of an IO_STACK_LOCATION*

All drivers—both filter drivers and the function driver—generally pass every power request down the stack to the driver underneath them. The only exceptions are an *IRP_MN_QUERY_POWER* request that the driver wants to cause to fail and an IRP that arrives while the device is being deleted.

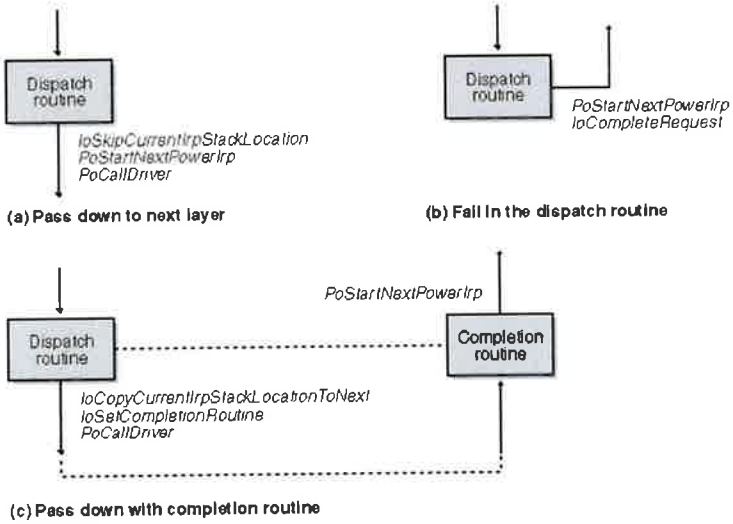


Figure 8-3. *Handling IRP_MJ_POWER requests.*

Special rules govern how you pass power requests down to lower-level drivers. Refer to Figure 8-3 for an overview of the process in the three possible variations you might use. First, before releasing control of a power IRP, you must call *PoStartNextPowerIrp*. You do so even if you are completing the IRP with an error status. The reason for this call is that the

IPR2024-00292
KASPERSKY EX 1008 Page 247

Power Manager maintains its own queue of power requests and must be told when it will be OK to dequeue and send the next request to your device. In addition to calling *PoStartNextPowerIrp*, you must call the special routine *PoCallDriver* (instead of *IoCallDriver*) to send the request to the next driver.

NOTE
Not only does the Power Manager maintain a queue of power IRPs for each device, but it maintains *two* such queues. One queue is for system power IRPs (that is, *IRP_MN_SET_POWER* requests that specify a system power state). The other queue is for device power IRPs (that is, *IRP_MN_SET_POWER* requests that specify a device power state). One IRP of each kind can be simultaneously active. Your driver might also be handling a Plug and Play (PnP) request and any number of substantive IRPs at the same time too, by the way.

The following function illustrates the mechanical aspects of passing a power request down the stack:

```
NTSTATUS DefaultPowerHandler(IN PDEVICE OBJECT fdo, IN PIRP Irp)
{
1
  PoStartNextPowerIrp(Irp);
2
  IoSkipCurrentIrpStackLocation(Irp);
3
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  return PoCallDriver(pdx->LowerDeviceObject, Irp);
}
```

1. *PoStartNextPowerIrp* tells the Power Manager that it can dequeue and send the next power IRP. You must make this call for every power IRP you receive at a time when you own the IRP. In other words, the call must occur either in your dispatch routine before you send the request to *PoCallDriver* or in a completion routine.

2. We use *IoSkipCurrentIrpStackLocation* to retard the IRP's stack pointer by one position in anticipation that *PoCallDriver* will immediately advance it. This is the same technique I've already discussed for passing a request down and ignoring what happens to it afterwards.

3. You use *PoCallDriver* to forward power requests. Microsoft implemented this function to forestall the minimal, but nonetheless measurable, impact on performance that might result from adding conditional logic to *IoCallDriver* to handle power management.

The function driver takes the two steps of passing the IRP down and performing its device-specific action in a neatly nested order, as shown in Figure 8-4: When *removing* power—that is, when changing to a lower power state—it performs the device-dependent step first and then passes the request down. When *adding* power—when changing to a higher power state—it passes the request down and performs the device-dependent step in a completion routine. This neat nesting of operations guarantees that the pathway leading to the hardware has power while the driver manipulates the hardware.

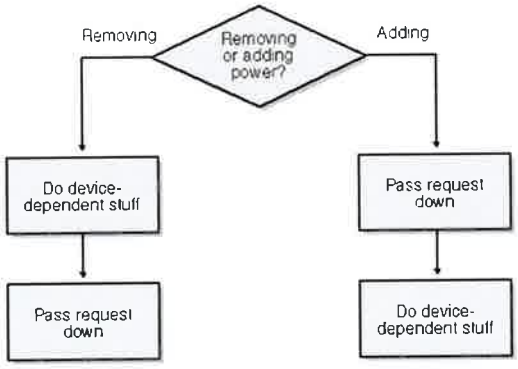


Figure 8-4. *Handling system power requests.*

Power IRPs come to you in the context of a system thread that you must not block. There are several reasons why you can't block the thread. If your device has the *INRUSH* characteristic, or if you've cleared the *DO_POWER_PAGABLE* flag in your device object, the Power Manager will send you IRPs at *DISPATCH_LEVEL*. You remember, of course, that you can't block a thread while executing at *DISPATCH_LEVEL*. Even if you've set *DO_POWER_PAGABLE*, however, so that you get power IRPs at *PASSIVE_LEVEL*, you can cause a deadlock by requesting a device power IRP while servicing a system IRP and then blocking: the Power Manager might not send you the device IRP until your system IRP dispatch routine returns, so you'll wait forever.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 248

The function driver normally needs to perform several steps that require time to finish as part of handling some power requests. The DDK points out that you can *delay* the completion of power IRPs by periods that the end user won't find perceptible under the circumstances, but being able to delay doesn't mean being able to block. The requirement that you can't block while these operations finish means lavish use of completion routines to make the steps asynchronous.

Implicit in the notion that *IRP_MN_QUERY_POWER* poses a question for you to answer yes or no is the fact that you can cause an IRP to fail with that minor function code. Having the IRP fail is how you say no. You don't have any such freedom with *IRP_MN_SET_POWER* requests, however: you must carry out the instructions they convey.

8.2 Managing Power Transitions

Performing power-management tasks correctly requires very accurate coding, and there are many complicating factors. For example, your device might have the ability to wake up the system from a sleeping state. Deciding whether to have a query succeed or fail, and deciding which device power state corresponds to a given new system power state, depend on whether your wake-up feature is currently armed. You may have powered down your own device because of inactivity, and you need to provide for restoring power when a substantive IRP comes along. Maybe your device is an "inrush" device that needs a large spike of current to power on, in which case the Power Manager treats you specially. And so on.

I regret being unable to offer a simple explanation of how to handle power management in a driver. It seems to me that a feature like this, which every driver must implement for the system to work properly, ought to have a simple explanation that any programmer can understand. But it doesn't. I recommend, therefore, that you simply adopt someone else's proven power-management code wholesale. If you build your driver to use GENERIC.SYS, you can delegate *IRP_MJ_POWER* requests like this:

```
NTSTATUS DispatchPower(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx =
    (PDEVICE EXTENSION) fdo->DeviceExtension;
  return GenericDispatchPower(pdx->pgx, Irp);
  }
```

GENERIC will then call back to your driver to handle certain device-dependent details. The power callback functions, all of which are entirely optional, are shown in Table 8-3.

Callback Function	Purpose
FlushPendingIo	"Encourage" any pending operations to finish.
GetDevicePowerState	Get the device power state to use for a specified system power state.
QueryPowerChange	Determine whether a proposed change in device power state will be permissible.
RestoreDeviceContext	Initiate nonblocking process to prepare device for use following restoration of power.
SaveDeviceContext	Initiate nonblocking process to prepare device for loss of power.

Table 8-3. *Power-Management Callbacks Used by GENERIC.SYS*

If you use my WDMWIZ wizard to generate a skeleton driver that doesn't use GENERIC, you'll get a source file (POWER.CPP) that implements the same power-handling model as GENERIC. Instead of callback routines, however, this code contains several "if (*FALSE*)" blocks with TODO comments where you should insert code to do what the GENERIC callbacks do.

In the following sections, I'll describe the requirements for handling *IRP_MN_QUERY_POWER* and *IRP_MN_SET_POWER* requests, and I'll also describe the finite state machine I built to carry out those requirements. I'm not going to show you all of the code, but I will show you enough fragments to cover the many fine points you must be aware of to understand anybody's code. If you wanted to implement your own power code, you wouldn't have to use a finite state machine as I've done. You could, for example, use a traditional set of I/O completion routines. I believe, however, that you would pretty much have to arrange those completion routines in such a way that they would actually be equivalent to the state machine I'll describe. That's why I feel justified in describing what might otherwise seem like an idiosyncratic implementation.

IPR2024-00292
KASPERSKY EX 1008 Page 249

Debugging Power Management
Debugging power-management code in a driver is uncommonly hard. The problems start with the fact that many desktop machines don't actually support standby power states. The reason for that isn't hard to understand: most driver programmers are unable to successfully implement power management (it's just too hard!), and their drivers are running on your development machines and getting in the way. Not only do the machines not support the power states you need for testing, but the operating system also silently decides to suppress the power-management choices on the start menu. There's no way to find out *why* your machine won't support standby or hibernate choices—you just find out that it won't.

For Windows 98/Me only, Microsoft publishes a tool named PMTSHOOT that will help you find out why your machine doesn't support standby. When I ran this program on one of my machines, I discovered that I needed to disable my network cards—never mind how they got a logo if their drivers didn't support power management correctly—and to shut down Internet Connection Sharing. Now, really! I think I could just tell the other users on my home office network (that would be me, myself, and I on a typical day) that the Internet will be temporarily inaccessible, couldn't I? This seems to me to be a case in which automation has run rampant into an area better left to humans.

You may have to purchase a recent vintage laptop if you want to do serious power-management testing. This is a lousy alternative if you're dealing with a Peripheral Component Interconnect (PCI) card because laptops typically lack expansion slots. Power management will work differently, and might not work at all, when the laptop is docked. Thus, any expansion capabilities offered by your docking station might still fall short of what you need. Universal serial bus (USB) host controllers vary widely in their support of power management too, so you may find it hard to buy a laptop that lets you test USB devices.

Supposing you manage to find suitable hardware for testing, you'll then run into problems trying to actually debug your driver. On my previous laptop, the serial port was inaccessible to WinDbg and Soft-Ice alike, apparently because of power-management features (!) associated with the infrared port that I didn't want originally and never used. Whenever there was a problem in my driver, it seemed that the serial port, network adapter, and display had all been turned off already, so I couldn't use any of my normal debugging techniques to investigate failures. Writing messages to log files was also useless because of when the file systems decided to shut down. All failures looked alike from this perspective: the system hung going into standby or resuming afterward, and the screen was black. Good luck drawing deductions from that!

(Notwithstanding that the disk system may not be powered up at the times you need to capture debug prints related to power management, the POWTRACE filter driver in the companion content will log to a disk file. Check out POWTRACE.HTM for hints on how to use it.)

I've finally stumbled into a laborious procedure that nevertheless has let me debug power management in my drivers. I use Soft-Ice, so I'm not dependent on having very many peripherals working. The keyboard and the parallel port appear to keep power longer than my own devices—I don't know whether this is on purpose or just accidental, but I'm grateful that it's true. So I wait for the system to hang, and I press the Print Screen key. If I'm lucky, Soft-Ice has gained control because of some bug check or *ASSERT*, and I can use the printer as a surrogate for the display. I can type a command and do another Print Screen to see the result. And so on. Tedious, but better than trial and error.

A much better solution to the whole problem of power testing and debugging would be a test program that would feed IRPs through a driver to perform a black box test. I've been lobbying Microsoft to write such a tool for a while now, and I hope that persistence eventually pays off.

8.2.1 Required Infrastructure

Whoever actually processes power requests in your driver needs to maintain several data values in the device extension structure:

```
typedef struct  DEVICE EXTENSION {
1
  DEVICE POWER STATE devpower;
2
  SYSTEM POWER STATE syspower;
3
  BOOLEAN StalledForPower;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

1. The value *devpower* is the current device power state. You probably initialize this to *PowerDeviceD0* at *AddDevice* time.
2. The value *syspower* is the current system power state. You always initialize this to *PowerSystemWorking* at *AddDevice* time because the computer is self-evidently in the Working state. (You're executing instructions.)
3. *StalledForPower* will be *TRUE* if your power-management code has stalled your substantive IRP queues across a period

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 250

of low power.

Note that GENERIC.SYS, if you use it, maintains these variables in its own private portion of the device extension structure; you won't need to declare them yourself.

8.2.2 Initial Triage

Your *IRP_MJ_POWER* dispatch function should initially distinguish between *IRP_MN_QUERY_POWER* and *IRP_MN_SET_POWER* requests on the one hand and other minor function codes on the other. In nearly every case, you will pend the query and set requests by calling *IoMarkIrpPending* and returning *STATUS_PENDING*. This is how you obey the rule, stated earlier, that you can't block the system thread in which you receive power IRPs. You will also want to examine the stack parameters to distinguish three basic cases:

- A system power IRP that increases system power. That is, an IRP for which *Parameters.Power.Type* is *SystemPowerState* and for which *Parameters.Power.State.SystemState* is numerically less than the remembered *syspower* value. Note that increasing values of the *SYSTEM_POWER_STATE* enumeration denote less power.
- A system power IRP that decreases system power or leaves it the same.
- A device power IRP. That is, an IRP for which *Parameters.Power.Type* is *DevicePowerState*, regardless of whether it indicates more or less power than your device currently has.

You'll handle power set and query operations in just about the same way up to a certain point, which is why you needn't initially distinguish between them.

Since I'm going to be showing you diagrams of my state machine, I have to explain just a bit more terminology. I put the code for the state machine into a subroutine (*HandlePowerEvent*), the arguments to which are a context structure containing all of the state information about the machine and an event code that indicates what has happened to cause the state machine to be invoked. There are only three event codes. The dispatch routine uses *NewIrp* in the first call to the state machine. *MainIrpComplete* indicates that an I/O completion routine is invoking the state machine to resume processing after a lower-level driver has completed an IRP, and *AsyncNotify* means that some other asynchronous process has completed. The state machine performs one or more actions based on the state and event codes. The machine initially occupies the *InitialState* state, and it performs the *TriageNewIrp* action when called to process a *NewIrp* event, for example. I'll explain the other states and actions as they come up in the following narrative.

8.2.3 System Power IRPs That Increase Power

If a system power IRP implies an increase in the system power level, you'll forward it immediately to the next lower driver after installing a completion routine. In the completion routine, you'll request the corresponding device power IRP. You'll select the return value from your completion routine as follows:

- In Windows 2000 and later systems, if the system power IRP is *IRP_MN_SET_POWER* for *PowerSystemWorking*, you'll return *STATUS_SUCCESS* to allow the Power Manager to immediately send similar IRPs to other drivers. Doing this speeds up restart from suspend by allowing drivers to overlap their device IRP processing.
- In every other case, you'll return *STATUS_MORE_PROCESSING_REQUIRED* to defer the completion of the system IRP. You'll complete the IRP after the device IRP completes.

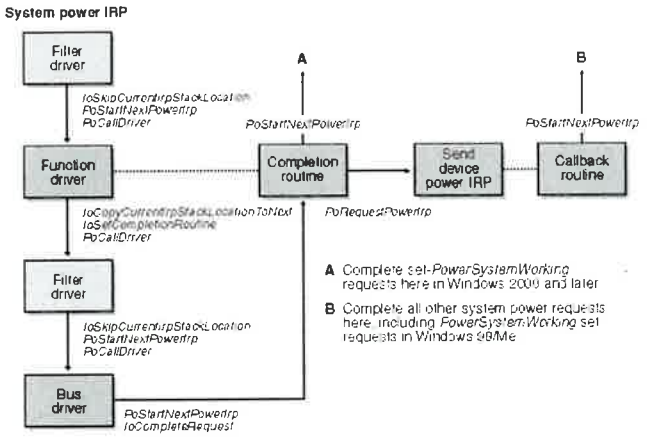


Figure 8-5. *IRP flow when increasing system power.*

Figure 8-5 diagrams the flow of the IRP through all of the drivers.

IPR2024-00292
KASPERSKY EX 1008 Page 251

Figure 8-6 illustrates the path taken by my finite state machine.

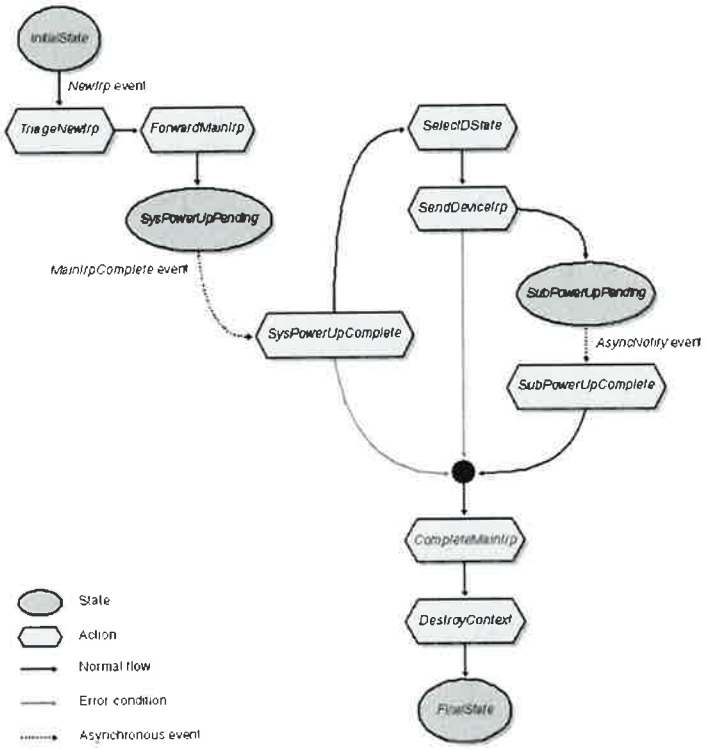


Figure 8-6. *State transitions when increasing system power.*

- *TriageNewIrp*, as noted earlier, is the first action for every power IRP. It discovers that we're dealing with a system power IRP that increases power, and it therefore arranges to perform the *ForwardMainIrp* action next.
- *ForwardMainIrp* installs an I/O completion routine and sends the system IRP down the driver stack after changing the machine state to *SysPowerUpPending*. At this point, the state machine subroutine returns to the *IRP_MJ_POWER* dispatch routine, which returns *STATUS_PENDING*.
- When the bus driver completes the system power IRP, the I/O completion routine reinvokes the state machine with the *MainIrpComplete* event code.
- The *SysPowerUpComplete* action first tests the completion status for the IRP. If the bus driver failed the IRP, we arrange to return *STATUS_SUCCESS* from the completion routine. That allows the IRP to finish completing with an error status. This is also the point in the code where we test to see whether we're dealing with an *IRP_MN_SET_POWER* for the *PowerSystemWorking* state in Windows 2000 or later. If so, we'll also allow the IRP to finish completing.
- Unless the system IRP has failed in the bus driver, we go on to perform *SelectDState* to select the device power state that corresponds to this IRP's system power state and to perform *SendDeviceIrp* to request a device power IRP with the same minor function code. I'll discuss the mechanics of doing both of these actions in detail in a moment. It's possible for the *SendDeviceIrp* step to fail, in which case we'll want to change the ending status for the system IRP to a failure code and allow that IRP to complete. We then exit the finite state machine, whereupon our I/O completion routine will return whatever status (*STATUS_SUCCESS* or *STATUS_MORE_PROCESSING_REQUIRED*) we've decided on after putting the state machine into the *SubPowerUpPending* state.
- Time will pass while our own driver handles the device power IRP we've just requested. Eventually, the Power Manager will call a callback routine in our driver to inform us that the device power IRP has completed. The callback routine in turn reinvokes the state machine with the *AsyncNotify* event code.
- The *SubPowerUpComplete* action doesn't actually do much in the retail build of my state machine except chain to the *CompleteMainIrp* event.
- *CompleteMainIrp* arranges to complete the system IRP if we haven't already done that when performing *SysPowerUpComplete*. Because the state machine has been invoked this time by an asynchronous event instead of an I/O completion routine, we have to actually call *IoCompleteRequest*. We might, however, be running at *DISPATCH_LEVEL*. In Windows 98/Me, we must be at *PASSIVE_LEVEL* when we complete power IRPs, so we might need to schedule a

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 252

work item (see Chapter 14) and return without destroying the state machine. The work-item callback routine then reinvokes the state machine at *PASSIVE_LEVEL* to finish up.

- *DestroyContext* is the last action performed by the state machine for any given power IRP. It releases the context structure that the machinery has been using to keep track of state information.

I want you to notice what the net result of all of this motion has been: we have requested a device power IRP. I don't want to negatively characterize the kernel power-management architecture because I surely lack encyclopedic knowledge of all the constituencies it serves and all the problems it solves. But all this motion does seem a trifle complex given the net result.

Mapping the System State to a Device State

Our obligation as power policy owner for a device is to originate a device power IRP, either a set or a query, with an appropriate device power state. I broke this into two steps: *SelectDState* and *SendDeviceIrp*. I'll discuss the first of these steps now.

In general, we always want to put our device into the lowest power state that's consistent with current device activity, with our own wake-up feature (if any), with device capabilities, and with the impending state of the system. These factors can interplay in a relatively complex way. To explain them fully, I need to digress briefly and talk about a PnP IRP that I avoided discussing in Chapter 6: *IRP_MN_QUERY_CAPABILITIES*.

The PnP Manager sends a capabilities query shortly after starting your device and perhaps at other times. The parameter for the request is a *DEVICE_CAPABILITIES* structure that contains several fields relevant to power management. Since this is the only time in this book I'm going to discuss this structure, I'm showing you the entire declaration:

```
typedef struct  DEVICE CAPABILITIES {
    USHORT Size;
    USHORT Version;
    ULONG DeviceD1:1;
    ULONG DeviceD2:1;
    ULONG LockSupported:1;
    ULONG EjectSupported:1;
    ULONG Removable:1;
    ULONG DockDevice:1;
    ULONG UniqueID:1;
    ULONG SilentInstall:1;
    ULONG RawDeviceOK:1;
    ULONG SurpriseRemovalOK:1;
    ULONG WakeFromD0:1;
    ULONG WakeFromD1:1;
    ULONG WakeFromD2:1;
    ULONG WakeFromD3:1;
    ULONG HardwareDisabled:1;
    ULONG NonDynamic:1;
    ULONG Reserved:16;

    ULONG Address;
    ULONG UINumber;

    DEVICE POWER STATE DeviceState[PowerSystemMaximum];
    SYSTEM POWER STATE SystemWake;
    DEVICE POWER STATE DeviceWake;
    ULONG D1Latency;
    ULONG D2Latency;
    ULONG D3Latency;
} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;
```

Table 8-4 describes the fields in this structure that relate to power management.

IPR2024-00292
KASPERSKY EX 1008 Page 253

Field	Description
DeviceState	Array of highest device states possible for each system state
SystemWake	Lowest system power state from which the device can generate a wake-up signal for the system—*PowerSystemUnspecified* indicates that the device can't wake up the system
DeviceWake	Lowest power state from which the device can generate a wake-up signal—*PowerDeviceUnspecified* indicates that the device can't generate a wake-up signal
D1Latency	Approximate worst-case time (in 100-microsecond units) required for the device to switch from D1 to D0 state
D2Latency	Approximate worst-case time (in 100-microsecond units) required for the device to switch from D2 to D0 state
D3Latency	Approximate worst-case time (in 100-microsecond units) required for the device to switch from D3 to D0 state
WakeFromD0	Flag indicating whether the device's system wake-up feature is operative when the device is in the indicated state
WakeFromD1	Same as above
WakeFromD2	Same as above
WakeFromD3	Same as above

Table 8-4. *Power-Management Fields in the DEVICE_CAPABILITIES Structure*

You normally handle the query capabilities IRP synchronously by passing it down and waiting for the lower layers to complete it. After the pass-down, you'll make any desired changes to the capabilities recorded by the bus driver. Your subdispatch routine will look like this one:

```
NTSTATUS HandleQueryCapabilities(IN PDEVICE OBJECT fdo, IN PIRP Irp)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  PDEVICE CAPABILITIES pdc = stack-> Parameters.DeviceCapabilities.Capabilities;
1
  if (pdc->Version < 1)
    return DefaultPnpHandler(fdo, Irp);
2
    <stuff>
  NTSTATUS status = ForwardAndWait(fdo, Irp);
  if (NT SUCCESS(status))
    {
    stack = IoGetCurrentIrpStackLocation(Irp);
    pdc = stack->Parameters.DeviceCapabilities.Capabilities;
3
    <stuff>
4
    AdjustDeviceCapabilities(pdx, pdc);
5
    pdx->devcaps = *pdc;
    }
  return CompleteRequest(Irp, status);
  }
```

1. The device capabilities structure has a version number member, which is currently always equal to 1. The structure is designed to always be upward compatible, so you'll be able to work with the version defined in the DDK that you build your driver with *and* with any later incarnation of the structure. If, however, you're confronted with a structure that's older than you're able to work with, you should just ignore this IRP by passing it along.

2. Here's where the DDK says you *add* capabilities.

3. Here's where the DDK says you should *remove* capabilities set by the bus driver.

4. Depending on which platform you're running on and which bus your device is connected to, the bus driver may have incorrectly completed the power-related portions of the capabilities structure. *AdjustDeviceCapabilities* compensates for that shortcoming.

5. It's a good idea to make a copy of the capabilities structure. You'll use the *DeviceState* map when you receive a system power IRP. You might have occasion to consult other fields in the structure too.

It's just not clear what the distinction is between "adding" and "removing" capabilities, unfortunately, and certain bus drivers have had bugs that cause them to erase whatever changes you make in the capabilities structure as the IRP travels down the stack. My advice, therefore, is to make your changes at both points 2 and 3 in the preceding code snippet—that is, both before

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 254

and after passing the IRP down.

You can modify *SystemWake* and *DeviceWake* to specify a higher power state than the bus driver thought was appropriate. You can't specify a lower power state for the wake-up fields, and you can't override the bus driver's decision that your device is incapable of waking the system. If your device is ACPI-compliant, the ACPI filter will set the *LockSupported*, *EjectSupported*, and *Removable* flags automatically based on the ACPI Source Language (ASL) description of the device—you won't need to worry about these capabilities.

You might want to set the *SurpriseRemovalOK* flag. Setting the flag suppresses the dialog box that earlier versions of Windows presented when they detected the sudden and unexpected removal of a device. It's normally OK for the end user to remove a USB or 1394 device without first telling the system, and the function driver should set this flag to avoid annoying the user.

As noted (point 4), some bus drivers don't correctly set the power-related fields in the capabilities structure, so it's up to you to tweak the structure somewhat. My *AdjustDeviceCapabilities* function, which is based on a talk given at WinHEC 2002 and on an example in the DDK TOASTER sample, does the following:

- Examines the *DeviceState* map in the capabilities structure. If D1 or D2 appears in that map, we can infer that the *DeviceD1* and *DeviceD2* flags ought to have been set.
- Uses the reported *DeviceWake* value, and the *DeviceState* value corresponding to the reported *SystemWake* value, to infer the settings of the *WakeFromDx* flags and the *DeviceD1* or *DeviceD2* flag.
- Infers *SystemWake* from the fact that some entry in the *DeviceState* map must allow the device to be at least as powered as the lowest D-state from which wake-up is possible.

To return to our discussion of how to select a power state, GENERIC will calculate minimum and maximum values and choose the lower of the two. The minimum is D3 unless you have an enabled wake-up feature and the system is in a state from which your device can wake it, in which case the minimum is the remembered *DeviceWake* capability. The maximum is the remembered *DeviceState* capability for the current system state. Then GENERIC calls your *GetDevicePowerState* callback, if you have one, to allow you to override the selection by picking a higher state. You can decide, for example, to put the device into D0 when system power is restored, but only if an application has a handle open to your device. For example:

```
DEVICE_POWER_STATE GetDevicePowerState(PDEVICE_OBJECT fdo,
  SYSTEM_POWER_STATE sstate, DEVICE_POWER_STATE dstate)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  if (sstate > SystemPowerWorking || !pdx->handles)
    return dstate;
  return PowerDeviceD0;
  }
```

Requesting a Device Power IRP

To submit an *IRP_MN_SET_POWER* or *IRP_MN_QUERY_POWER* request, you call the special function shown here:

```
PSOMETHING_OR_ANOTHER ctx;
POWER_STATE devstate;
devstate.DeviceState = PowerDeviceDx;
NTSTATUS postatus = PoRequestPowerIrp(pdx->Pdo, IRP_MN_XXX_POWER, devstate,
  (PREQUEST_POWER_COMPLETE) PoCallbackRoutine, ctx, NULL);
```

The first argument to *PoRequestPowerIrp* is the address of the physical device object (PDO) at the bottom of your device's PnP stack. The second argument is the minor function code for the IRP we want to submit. Insofar as concerns us right now, this would be the same minor function code as the system power IRP that we happen to be processing. That is, it's either *IRP_MN_QUERY_POWER* or *IRP_MN_SET_POWER*. The third argument is a power state derived as described in the preceding section. *PoCallbackRoutine* is a callback routine (not a standard I/O completion routine), and *ctx* is a context parameter for that function. The final argument (*NULL* in this example) is the address of a variable where *PoRequestPowerIrp* will store the address of the IRP it creates. You don't use this particular capability for SET and QUERY requests.

PoRequestPowerIrp creates a device power IRP of the specified type and with the specified power level, and sends it to the topmost driver in your PnP stack. If *PoRequestPowerIrp* returns with *STATUS_PENDING*, you can infer that it actually created and sent the IRP. The Power Manager will then eventually call your callback routine. If *PoRequestPowerIrp* returns anything except *STATUS_PENDING*, however, the Power Manager will not call your callback routine. This possibility is why the *SendDeviceIrp* action in my finite state machine might exit through the *CompleteMainIrp* action in order to complete the system IRP.

You mustn't request a device power IRP if your device is already in the state you're requesting. There is a bug in Windows 98/Me such that *PoRequestPowerIrp* will appear to succeed in this case, but the CONFIGMG driver won't actually send a configuration event to NTKERN. In this situation, your power code will deadlock waiting for a call to your *PoCallbackRoutine* that will never happen. It has also been my experience that Windows 2000 and Windows XP will hang while resuming from standby if you ask to set the state that currently obtains.

IPR2024-00292
KASPERSKY EX 1008 Page 255

8.2.4 System Power IRPs That Decrease Power

If the system power IRP implies no change or a reduction in the system power level, you'll request a device power IRP with the same minor function code (set or query) and a device power state that corresponds to the system state. When the device power IRP completes, you'll forward the system power IRP to the next lower driver. You'll need a completion routine for the system power IRP so that you can make the requisite call to *PoStartNextPowerIrp* and so that you can perform some additional cleanup. See Figure 8-7 for an illustration of how the IRPs flow through the system in this case.

It's not strictly necessary to issue the device-dependent power request as the system IRP is traveling down the stack. That is, we can issue the request from the I/O completion routine for the system power IRP, just as we did in the case of a system power IRP that increases power (studied earlier). In fact, the DDK says to do exactly this. Performing the steps in the order I suggest has the virtue of having been tested and proven to work in many drivers on many WDM platforms since the first edition was published, however. Here, therefore, I follow the admonition "If it ain't broke, don't fix it."

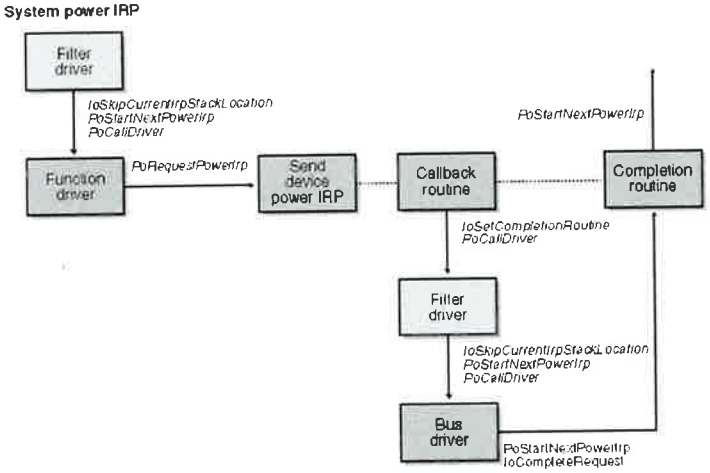


Figure 8-7. *IRP flow when decreasing system power.*

Figure 8-8 diagrams how my finite state machine handles this type of IRP. *TriageNewIrp* puts the state machine into the *SubPowerDownPending* state and jumps to the *SelectDState* action. You already saw that *SelectDState* selects a device power state and leads to a *SendDeviceIrp* action to request a device power IRP. In the system power-down scenario, we'll be specifying a lower power state in this device IRP.

8.2.5 Device Power IRPs

All we actually do with system power IRPs is act as a conduit for them and request a device IRP either as the system IRP travels down the driver stack or as it travels back up. We have more work to do with device power IRPs, however.

To begin with, we don't want our device occupied by any substantive I/O operations while a change in the device power state is under way. As early as we can in a sequence that leads to powering down our device, therefore, we wait for any outstanding operation to finish, and we stop processing new operations. Since we're not allowed to block the system thread in which we receive power IRPs, an asynchronous mechanism is required. Once the current IRP finishes, we'll continue processing the device IRP. Each of the next four state diagrams (Figures 8-11 through 8-14), therefore, begins with the same sequence. *TriageNewIrp* tests the *StalledForPower* flag to see whether the substantive IRP queues have already been stalled for a power operation. If not, GENERIC does two things:

- It calls the *DEVQUEUE* routine named *StallAllRequestsAndNotify*. That routine stalls all your substantive IRP queues and returns an indication of whether your device is currently busy servicing an IRP from one of them. In the latter case, GENERIC will end up deferring further processing of the IRP until someone calls *StartNextPacket* for each currently busy queue.
- It calls your *FlushPendingIo* callback routine if you have specified one. This function solves a problem reported by a reader of the first edition, as follows: Your *StartIo* routine might have started a time-consuming operation that won't finish of its own accord. For example, you might have repackaged the IRP as an *IRP_MJ_INTERNAL_DEVICE_CONTROL* and sent it to the USB bus driver, and you're planning to call *StartNextPacket* from a completion routine when the USB device completes the repackaged IRP. This won't necessarily happen soon without some sort of "encouragement" (which, in this case, would be a pipe abort command) that your callback can supply.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 256

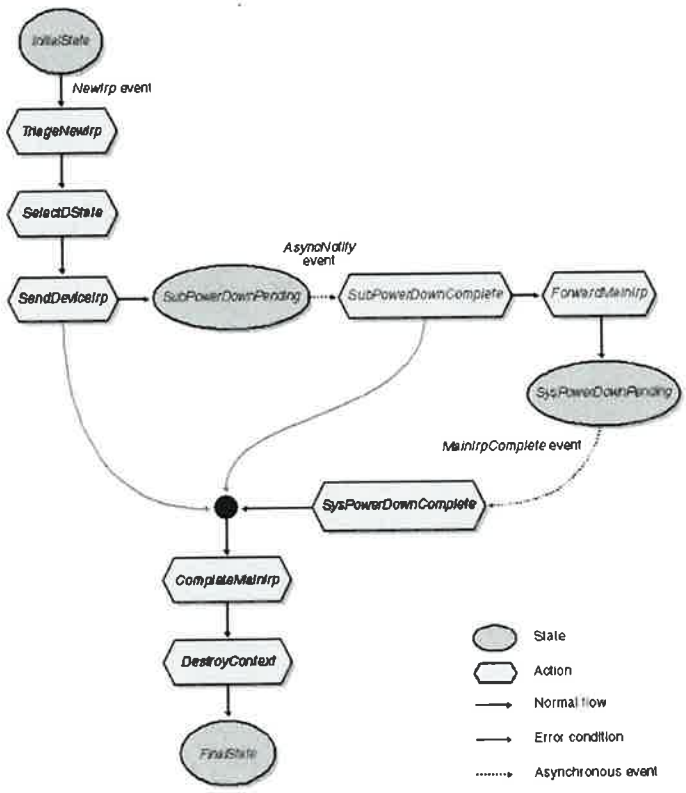


Figure 8-8. *State transitions when decreasing system power.*

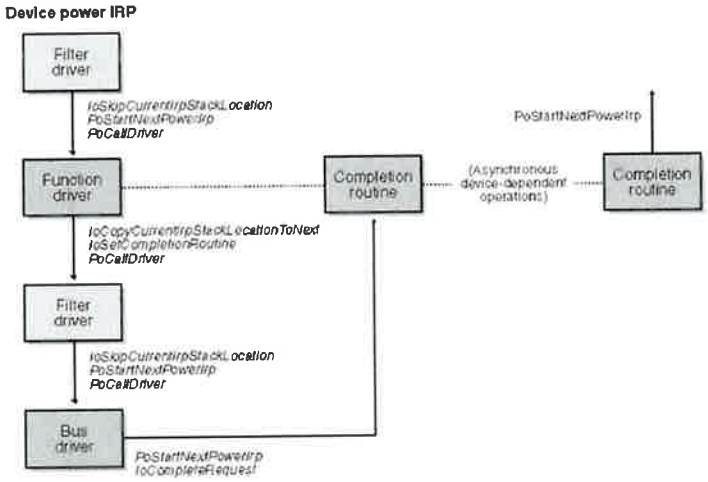


Figure 8-9. *IRP flow when increasing device power.*

If the device power IRP implies an increase in the device power level, we'll forward it to the next lower driver. Refer to Figure 8-9 for an illustration of how the IRP flows through the system. The bus driver will process a device set-power IRP by, for example, using whatever bus-specific mechanism is appropriate to turn on the flow of electrons to your device, and it will complete the IRP. Your completion routine will initiate the operations required to restore context information to the device, and it will return *STATUS_MORE_PROCESSING_REQUIRED* to interrupt the completion process for the device IRP. When the context-restore operation finishes, you'll resume processing substantive IRPs and finish completing the device IRP.

If the device power IRP implies no change or a reduction in the device power level, you perform any device-specific processing (asynchronously, as we've discussed) and then forward the device IRP to the next lower driver. See Figure 8-10.

IPR2024-00292
KASPERSKY EX 1008 Page 257

The "device-specific processing" for a query operation involves stalling your queues and waiting for all current substantive IRPs to finish. The device-specific processing for a set operation includes saving device context information, if any, in memory so that you can restore it later. The bus driver completes the request. In the case of a query operation, you can expect the bus driver to complete the request with *STATUS_SUCCESS* to indicate acquiescence in the proposed power change. In the case of a set operation, you can expect the bus driver to take the bus-dependent steps required to put your device into the specified device power state. Your completion routine cleans up by calling *PoStartNextPowerIrp*, among other things.

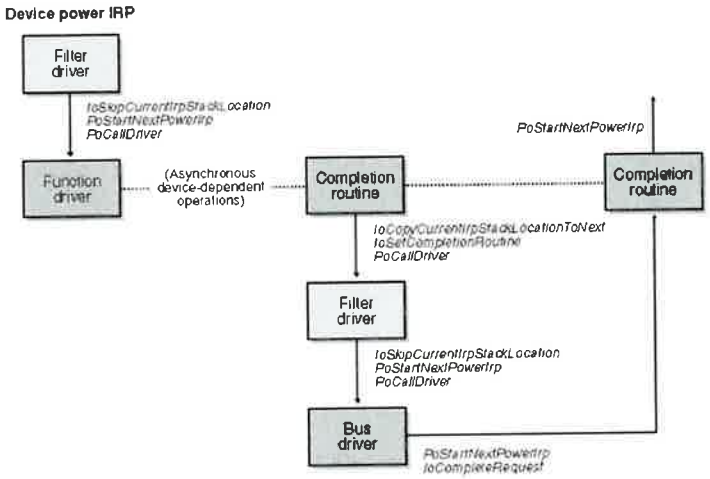


Figure 8-10. *IRP flow when decreasing device power.*

Setting a Higher Device Power State

Figure 8-11 diagrams the state transitions that occur for an *IRP_MN_SET_POWER* that specifies a higher device power state than that which is current.

The state transitions and actions are as follows:

- *TriageNewIrp* makes sure that your substantive IRP queues are stalled. *QueueStallComplete* picks up the processing of the device power IRP once this is accomplished.
- *ForwardMainIrp* sends the device IRP down the PnP stack. The bus driver turns on the flow of current to the device and completes the IRP.
- When the device IRP completes, our completion routine reinvokes the state machine to perform the *DevPowerUpComplete* action. If the device IRP failed (I've never seen this happen, by the way), we'll exit via *CompleteMainIrp*.
- At this point, GENERIC will call your *RestoreDeviceContext* callback routine, if any, to allow you to initiate a nonblocking process to prepare your device for operation in the new, higher power state. I'll discuss this aspect of the processing in more detail shortly.
- When the context-restore operation finishes (or immediately if there's no *RestoreDeviceCallback*), *ContextRestoreComplete* unstalls the substantive IRP queues (which have presumably been stalled while power was off) and hands off control to *CompleteMainIrp*.
- *CompleteMainIrp* arranges to complete the device IRP. We sometimes get here from the I/O completion routine we've installed for the device IRP, in which case we only need to return *STATUS_SUCCESS* to allow the completion process to continue. In other cases, our I/O completion routine long ago returned *STATUS_MORE_PROCESSING_REQUIRED*, and we need to call *IoCompleteRequest* to resume the completion process. In either case, since we're usually processing a device IRP that we requested while handling a system IRP, the next thing that will happen is that the Power Manager will call our *PoCompletionRoutine* to indicate that the device IRP is truly finished. We then go on to destroy this instance of the state machine, and another (earlier) instance picks up its own processing of a system IRP.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 258

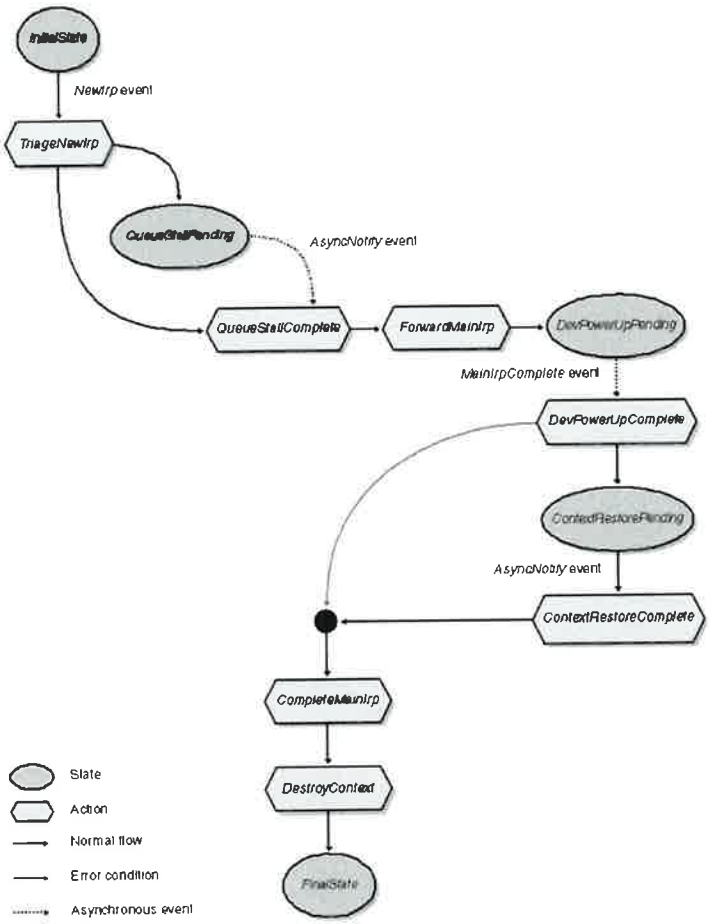


Figure 8-11. *State transitions when setting a higher device power state.*

The *RestoreDeviceContext* callback is an important part of how GENERIC helps you manage power for your device. As indicated, you have the opportunity in this routine to initiate any sort of nonblocking process that you need to perform before your device will be ready to operate with the new, higher power state. When GENERIC calls this routine, the bus driver has already restored power to your device. The function has this skeleton:

```
VOID RestoreDeviceContext(PDEVICE OBJECT fdo,
  DEVICE POWER STATE oldstate, DEVICE POWER STATE newstate,
  PVOID context)
  {

  }
```

Here *oldstate* and *newstate* are the previous and new states of your device, and *context* is an opaque parameter that you'll supply in an eventual call to *GenericSaveRestoreComplete*. Inside this function, you can perform any nonblocking activity needed to prepare your device. That is, you can read and write hardware registers or call any other kernel routine that doesn't block the current thread. You can't do something such as send a *synchronous* IRP to another driver because you would need to block the current thread until that IRP completes. You can, however, send *asynchronous* IRPs to other drivers. When your device is completely ready to go, make the following call back to GENERIC:

```
GenericSaveRestoreComplete(context);
```

where context is the context parameter you got in your *RestoreDeviceContext* call. GENERIC will then resume handling the device power IRP as previously discussed. Note that you can call *GenericSaveRestoreComplete* from within your *RestoreDeviceContext* function if you've finished all desired power-up operations.

IPR2024-00292
KASPERSKY EX 1008 Page 259

NOTE
For whatever reason, I've written many drivers for SmartCard readers. My *RestoreDeviceContext* function for these drivers completes any outstanding card-absent tracking IRP (a requirement for safely dealing with the possibility that someone has replaced a card while power was off). In addition, with devices that require continuous polling to detect card insertion and removal events, I restart the polling thread.

Note that you don't need to supply a *RestoreDeviceContext* function if there's no work to do at power-up time.

The net result of all the motion for a device set higher power request is that the bus driver repowers our device and we thereafter prepare it for reuse. This is still a lot of motion, but at least something useful comes from it.

Querying for a Higher Device Power State

You shouldn't expect to receive an *IRP_MN_QUERY_POWER* that refers to a higher power state than your device is already in, but you shouldn't crash the system if you do happen to receive one. Figure 8-12 illustrates the state changes my finite state machine goes through in such a case. The machine simply stalls the IRP queues if they didn't happen to have been stalled when power was removed earlier.

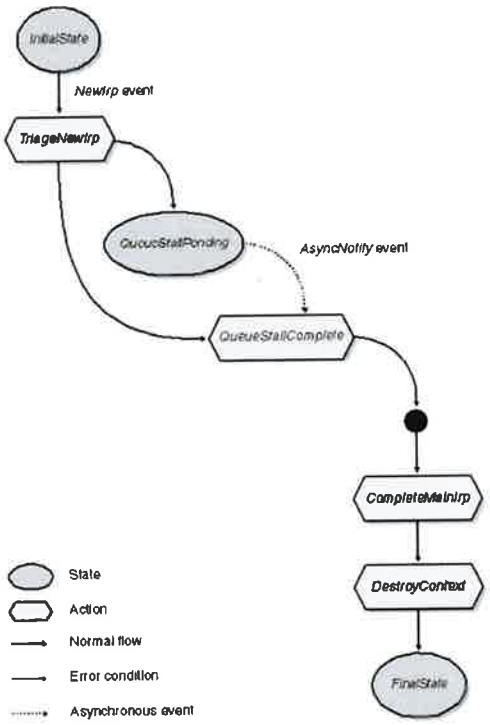


Figure 8-12. *State transitions for a query about a higher device power state.*

Setting a Lower Device Power State

If the IRP is an *IRP_MN_SET_POWER* for the same or a lower device power state than current, the finite state machine goes through the state transitions diagrammed in Figure 8-13.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 260

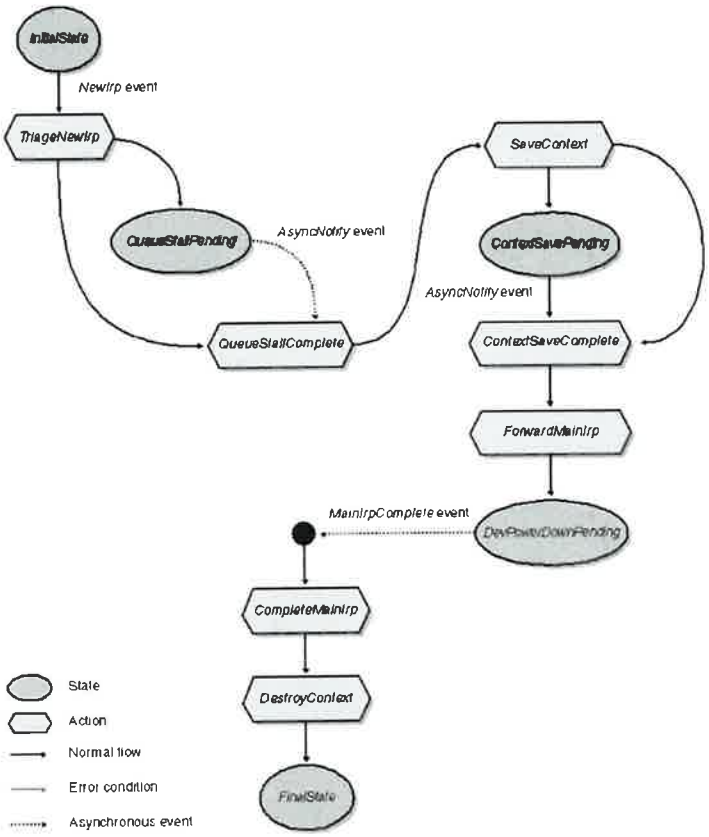


Figure 8-13. *State transitions when setting a lower device power state.*

The actions and state transitions in this case are as follows:

- *TriageNewIrp* makes sure that your substantive IRP queues are stalled. *QueueStallComplete* picks up the processing of the device power IRP once this is accomplished.
- At this point, GENERIC will call your *SaveDeviceContext* callback routine, if any, to allow you to initiate a nonblocking process to prepare your device for operation in the new, lower power state. I'll discuss this aspect of the processing in more detail shortly.
- When the context-save operation finishes (or immediately if there's no *SaveDeviceCallback*), *ContextSaveComplete* hands off control to *ForwardMainIrp*.
- *ForwardMainIrp* sends the device IRP down the PnP stack. The bus driver turns off the flow of current to the device and completes the IRP.
- When the device IRP completes, our completion routine reinvokes the state machine to perform the *CompleteMainIrp* action.
- *CompleteMainIrp* arranges to complete the device IRP. We always get here from the I/O completion routine we've installed for the device IRP, so we only need to return *STATUS_SUCCESS* to allow the completion process to continue. Since we're usually processing a device IRP that we requested while handling a system IRP, the next thing that will happen is that the Power Manager will call our *PoCompletionRoutine* to indicate that the device IRP is truly finished. We then go on to destroy this instance of the state machine, and another (earlier) instance picks up its own processing of a system IRP.

GENERIC's context-save protocol is exactly complementary to the context-restore protocol discussed previously. If you've supplied a *SaveDeviceContext* function, GENERIC will call it:

```
VOID SaveDeviceContext(PDEVICE_OBJECT fdo,
   DEVICE_POWER_STATE oldstate,
   DEVICE_POWER_STATE newstate, PVOID context)
```


IPR2024-00292
KASPERSKY EX 1008 Page 261

```
{
    .
    .
    .
}
```

You initiate any desired nonblocking operation to prepare your device for low power operation, and you call *GenericSaveRestoreComplete* when you finish. GENERIC then resumes handling the device power IRP as just described. Note that your device still has power at the time GENERIC calls your callback routine.

You don't need to supply a *SaveDeviceContext* function if there's no work to do at power-down time.

NOTE
To finish the story about my SmartCard drivers, I use *SaveDeviceContext* to halt any polling thread that I might have for detecting card insertions and removals. Since this operation requires blocking the current thread until the polling thread exits, I ordinarily need to schedule a work item that can block a *different* system thread, wait for the polling thread to terminate, and then call *GenericSaveRestoreComplete*.

Querying for a Lower Device Power State

An *IRP_MN_QUERY_POWER* that specifies the same or a lower device power state than current is the basic vehicle by which a function driver gets to vote on changes in power levels. Figure 8-14 shows how my state machine handles such a query.

- *TriageNewIrp* makes sure that your substantive IRP queues are stalled. *QueueStallComplete* picks up the processing of the device power IRP once this is accomplished.
- At this point (*DevQueryDown*), GENERIC will call your *QueryPower* callback routine, if any, to allow you to decide whether to accept the proposed change. If your function returns *FALSE*, GENERIC then short-circuits around a couple of actions to *DevQueryDownComplete* and thence to *CompleteMainIrp*.
- *ForwardMainIrp* sends the device IRP down the PnP stack. The bus driver usually just completes the IRP with a success status.
- When the device IRP completes, our completion routine reinvokes the state machine to perform the *DevQueryDownComplete* action. If the query has failed, we'll unstall our queues just in case we don't later get a set-power IRP to make us do that.
- *CompleteMainIrp* arranges to complete the device IRP. Since we're usually processing a device IRP that we requested while handling a system IRP, the next thing that will happen is that the Power Manager will call our *PoCompletionRoutine* to indicate that the device IRP is truly finished. We then go on to destroy this instance of the state machine, and another (earlier) instance picks up its own processing of a system IRP.

The net effect of these actions is to stall our substantive IRP queues if the query succeeds.

8.2.6 Flags to Set in *AddDevice*

Two flag bits in a device object—see Table 8-5—control various aspects of power management. After you call *IoCreateDevice* in your *AddDevice* function, both of these bits will be set to 0, and you can set one or the other of them depending on circumstances.

Flag	Description
DO_POWER_PAGABLE	Driver's *IRP_MJ_POWER* dispatch routine must run at *PASSIVE_LEVEL*.
DO_POWER_INRUSH	Powering on this device requires a large amount of current.

Table 8-5. *Power-Management Flags in DEVICE_OBJECT*

Set the *DO_POWER_PAGABLE* flag if your dispatch function for *IRP_MJ_POWER* requests must run at *PASSIVE_LEVEL*. The flag has the name it does because, as you know, paging is allowed at *PASSIVE_LEVEL* only. If you leave this flag set to 0, the Power Manager is free to send you power requests at *DISPATCH_LEVEL*. In fact, it always will do so in the current release of Windows XP.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 262

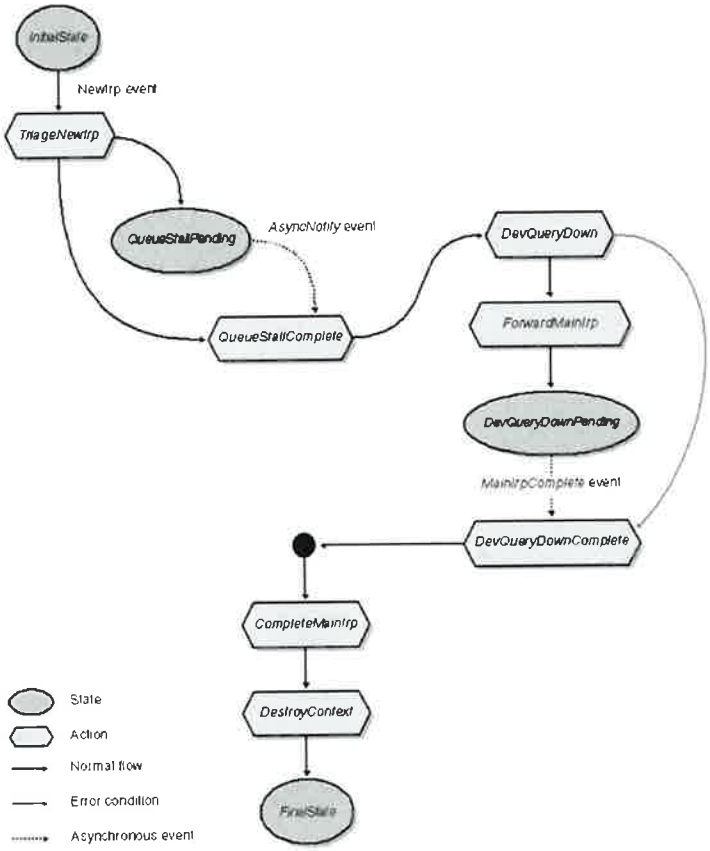


Figure 8-14. *State transitions for a query about a lower device power state.*

8.3 Additional Power-Management Details

In this section, I'll describe some additional details about power management, including flags you might need to set in your device object, controlling your device's wake-up feature, arranging for power-down requests after your device has been idle for a predetermined time, and optimizing context-restore operations.

Set the *DO_POWER_INRUSH* flag if your device draws so much current when powering up that other devices should not be allowed to power up simultaneously. The problem solved by this flag is familiar to people who've experienced multiple simultaneous spikes of electricity demand at the end of a power outage—having all your appliances trying to cycle on at the same time can blow the main breaker. The Power Manager guarantees that only one inrush device at a time will be powered up. Furthermore, it sends power requests to inrush devices at *DISPATCH_LEVEL*, which implies that you should not also set the *DO_POWER_PAGABLE* flag.

The system's ACPI filter driver will set the *INRUSH* flag in the PDO automatically if the ASL description of the device so indicates. All that's required for the system to properly serialize inrush power is that some device object in the stack have the *INRUSH* flag; you won't need to set the flag in your own device object too. If the system can't automatically determine that you require inrush treatment, however, you'll need to set the flag yourself.

The settings of the *PAGABLE* and *INRUSH* flags need to be consistent in all the device objects for a particular device. If the PDO has the *PAGABLE* flag set, every device object should also have *PAGABLE* set. Otherwise, a bug check with the code *DRIVER_POWER_STATE_FAILURE* can occur. (It's legal for a *PAGABLE* device to be layered on top of a non-*PAGABLE* device, just not the other way around.) If a device object has the *INRUSH* flag set, neither it nor any lower device objects should be *PAGABLE*, or else an *INTERNAL_POWER_ERROR* bug check will occur. If you're writing a disk driver, don't forget that you may change back and forth from time to time between pageable and nonpageable status in response to device usage PnP notifications about paging files.

8.3.1 Device Wake-Up Features

Some devices have a hardware wake-up feature, which allows them to wake up a sleeping computer when an external event

IPR2024-00292
KASPERSKY EX 1008 Page 263

occurs. See Figure 8-15. The power switch on the current crop of PCs is such a device. So are many modems and network cards, which are able to listen for incoming calls and packets, respectively. USB devices ordinarily claim wake-up capability, and many hubs and host controllers implement the wake-up signaling needed to support that claim.

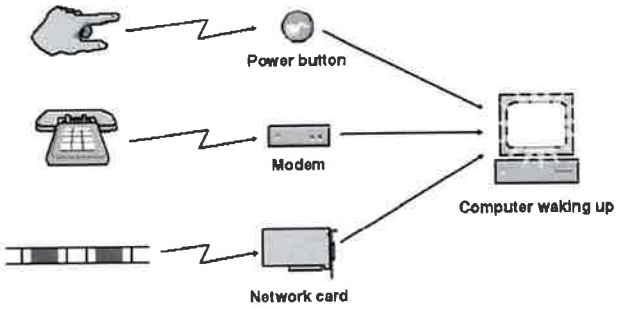


Figure 8-15. *Examples of devices that wake the system.*

If your device has a wake-up feature, your function driver has additional power-management responsibilities beyond the ones we've already discussed. The additional responsibilities revolve around the *IRP_MN_WAIT_WAKE* flavor of *IRP_MJ_POWER*:

- You'll inspect the device capabilities to determine whether and under what circumstances, in the opinion of the bus driver and the ACPI filter driver, your device will be capable of waking the system.
- You'll maintain some persistent setting to indicate whether the end user wants you to arm your device's wake-up feature.
- At a time when your device stack isn't in the middle of a power transition, you'll use *PoRequestPowerIrp* to originate an *IRP_MN_WAIT_WAKE* request, which the bus driver will ordinarily pend.
- When you have a choice about which power state to put your device into, you'll try to choose the lowest available state consistent with your device's wake-up capabilities.
- If your device causes the system to wake from a standby state, the bus driver will complete your *WAIT_WAKE* request. Thereafter, the Power Manager will call your power callback routine, from within which you should originate a device set-power request to restore your device to the D0 state.

NOTE
The WAKEUP sample in the companion content illustrates how to implement wake-up functionality using GENERIC.SYS. GENERIC itself, also in the companion content, contains the code discussed throughout this section. Note that WDMWIZ will generate exactly the same code (but with slightly different function names) if you use it to generate a skeleton project that doesn't use GENERIC.

Giving the End User Control

The end user has ultimate control over whether your device's wake-up feature, if any, should actually be armed. The standard way to provide this control is to support a Windows Management Instrumentation (WMI) class named *MSPower_DeviceWakeEnable*. (See Chapter 10 for more information about WMI.) The Device Manager automatically generates a Power Management tab in the device properties if the driver supports either *MSPower_DeviceWakeEnable* or *MSPower_DeviceEnable*. See Figure 8-16.

Your driver should remember the current state of *MSPower_DeviceWakeEnable*'s *Enable* member in both your device extension structure and in a registry key. You'll probably want to initialize your device extension variable from the registry at *AddDevice* time.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 264

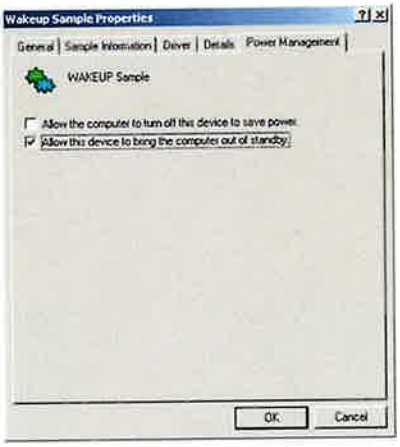


Figure 8-16. *Power Management tab in device properties.*

WAIT_WAKE Mechanics

As the power policy owner for your device, your driver originates a *WAIT_WAKE* request by calling *PoRequestPowerIrp*:

```
if (InterlockedExchange(&pdx->wwoutstanding, 1))
  <skip remaining statements>pdx->wwcancelled = 0;
POWER STATE junk;
junk.SystemState = pdx->devcaps.SystemWake
status = PoRequestPowerIrp(pdx->Pdo, IRP MN WAIT WAKE,
  junk, (PREQUEST POWER COMPLETE) WaitWakeCallback,
  pdx, &pdx->WaitWakeIrp);if (!NT SUCCESS(status))
  {
  pdx->WakeupEnabled = FALSE;
  pdx->wwoutstanding = 0;
  .
  .
  }
```

PoRequestPowerIrp creates an *IRP_MJ_POWER* with the minor function *IRP_MN_WAIT_WAKE* and sends it to the topmost driver in your own PnP stack. (I'll explain the other statements shortly.) This fact means that your own POWER dispatch function will see the IRP as it travels down the stack. You should install a completion routine and pass it further down the stack:

```
IoCopyCurrentIrpStackLocationToNext(Irp);
IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE)
  WaitWakeCompletionRoutine, pdx, TRUE, TRUE, TRUE);
PoStartNextPowerIrp(Irp);
status = PoCallDriver(pdx->LowerDeviceObject, Irp);
.
.
return status;
```

The bus driver will normally pend the IRP and return *STATUS_PENDING*. That status code will percolate back up through the dispatch routines of all the drivers in your stack and will eventually cause *PoRequestPowerIrp* to return *STATUS_PENDING*. Several other actions are possible, however:

- If you send more than one *WAIT_WAKE* request, the bus driver will complete the second and subsequent ones with *STATUS_DEVICE_BUSY*. In other words, you can have only one *WAIT_WAKE* outstanding.
- If the device is already in too low a power state (less than the *DeviceWake* capability, in other words), the bus driver completes a *WAIT_WAKE* with *STATUS_INVALID_DEVICE_STATE*.
- If the device capabilities indicate that your device doesn't support wake-up in the first place, the bus driver completes a *WAIT_WAKE* with *STATUS_NOT_SUPPORTED*.

Note that if the bus driver immediately causes the IRP to fail, your I/O completion routine (*WaitWakeCompletionRoutine* in the preceding example) will be called. Your power callback routine (*WaitWakeCallback*) will not.

IPR2024-00292
KASPERSKY EX 1008 Page 265

Completing *IRP_MN_WAIT_WAKE*

In the normal case, when the bus driver returns *STATUS_PENDING*, you simply leave the IRP sitting there, waiting for one of several things to happen:

- The system goes into standby and, still later, wakes up because your device asserts its wake-up signal. The computer hardware repowers automatically. The bus driver detects that your device is responsible and completes your *WAIT_WAKE* with *STATUS_SUCCESS*. Later on, you and all other drivers receive a *PowerSystemWorkingSET_POWER* request. In response, you originate a device power IRP to put your device into an appropriate power state. Since your device is attempting to communicate with the computer, you'll probably want to put the device into the D0 state at this point.
- The system stays in (or returns to) the *PowerSystemWorking* state, but your device ends up in a low power state. Thereafter, your device asserts its wake-up signal, and the bus driver completes your *WAIT_WAKE*. In this case, the Power Manager won't send you a system power IRP because the system is already in the working state, but your device is still in its low power state. To handle this case properly, your power callback routine (*WaitWakeCallback* in the example) needs to originate a device power IRP to put your device into (probably) the D0 state.
- Your device or the system enters a power state inconsistent with your wake-up signaling. That is, your device goes to a state less powered than *DeviceWake*, or the system goes to a state less powered than *SystemWake*. The bus driver realizes that your wake-up signal can no longer occur, and it completes your *WAIT_WAKE* with *STATUS_INVALID_DEVICE_STATE*.
- You decide to abandon the *WAIT_WAKE* yourself, so you call *IoCancelIrp*. The bus driver's cancel routine completes the IRP with *STATUS_CANCELLED*. You should cancel your *WAIT_WAKE* (at least) when you process an *IRP_MN_STOP_DEVICE*, *IRP_MN_SURPRISE_REMOVAL*, or *IRP_MN_REMOVE_DEVICE* request.

In all of these situations, the I/O Manager calls your I/O completion routine (*WaitWakeCompletionRoutine*) as a normal part of completing the IRP. In addition, the Power Manager calls your power callback routine (*WaitWakeCallback*) once the IRP is totally complete.

Your I/O Completion Routine

Cancelling an *IRP_MN_WAIT_WAKE* exposes the same race condition we discussed in Chapter 5 between your call to *IoCancelIrp* and the bus driver's call to *IoCompleteRequest*. To safely cancel the IRP, you can use a variation of the technique I showed you for cancelling asynchronous IRPs. This technique relies on interlocking your cancel logic with your completion routine in a special way, as shown here:

```
VOID CancelWaitWake(PDEVICE EXTENSION pdx)
  {
  PIRP Irp = (PIRP) InterlockedExchangePointer((PVOID*) &pdx->WaitWakeIrp, NULL);
  if (Irp)
    {
    IoCancelIrp(Irp);
    if (InterlockedExchange(&pdx->wwcancelled, 1))
      IoCompleteRequest(Irp, IO NO INCREMENT);
    }
  }

NTSTATUS WaitWakeCompletionRoutine(PDEVICE OBJECT junk, PIRP Irp,
  PDEVICE EXTENSION pdx)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
  if (InterlockedExchangePointer(
    (PVOID*) &pdx->WaitWakeIrp, NULL))
    return STATUS SUCCESS;
  if (InterlockedExchange(&pdx->wwcancelled, 1))
    return STATUS SUCCESS;
  return STATUS MORE PROCESSING REQUIRED;
  }
```

In the example I showed you in Chapter 5, we were dealing with an asynchronous IRP that we had created. We had the responsibility for calling *IoFreeIrp* to delete the IRP when it was finally complete. In this situation, the Power Manager has created the *WAIT_WAKE* IRP and will make its own call to *IoFreeIrp* when the IRP completes. To avoid the cancel/complete race condition, therefore, we need to delay completion until we're past the point where we might want to cancel the request.

The cancel logic depends partly on an undocumented but essential side effect of how *PoRequestPowerIrp* works internally. Recall that the last argument to *PoRequestPowerIrp* is the address of a PIRP variable that will receive the address of the IRP that gets created. *PoRequestPowerIrp* sets that variable before it sends the IRP to the topmost driver. In the example, I used *pdx->WaitWakeIrp* for that parameter. This is the same data member used in the I/O completion routine and in the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 266

CancelWaitWake routine. I'm deliberately relying on the fact that *PoRequestPowerIrp* will set this member before sending the IRP rather than after so that my completion and cancel logic will work correctly, even if invoked before the topmost dispatch routine returns to *PoRequestPowerIrp*. Note that it would be a mistake to write your code this way:

```
PIRP foo;
status = PoRequestPowerIrp(pdx->Pdo, IRP MN WAIT WAKE, junk,
  (PREQUEST POWER COMPLETE) WaitWakeCallback, pdx, &foo);
pdx->WaitWakeIrp = foo; // <== don't do this
```

This sequence risks setting *WaitWakeIrp* to the address of a completed IRP. Havoc would occur were you to later try to cancel that IRP.

Your Callback Routine

Your power callback routine for *WAIT_WAKE* will look something like this one:

```
VOID WaitWakeCallback(PDEVICE OBJECT junk, UCHAR MinorFunction,
  POWER STATE state, PDEVICE EXTENSION pdx,
  PIO STATUS BLOCK pstatus)
  {
  InterlockedExchange(&pdx->wwoutstanding, 0);
  if (!NT SUCCESS(pstatus->Status))
    return;
  else
    {
    SendDeviceSetPower(pdx, PowerDeviceD0, FALSE);
    }
  }
```

Here I imagine that you've written (or copied!) a subroutine named *SendDeviceSetPower* that will originate a device *SET_POWER* request. I don't want to show you that function here because it needs to dovetail with your other power logic. There's a routine by that name in GENERIC.SYS or in any driver you build using WDMWIZ, so you can check out my implementation of that function. You call *SendDeviceSetPower* to put your device into the D0 state to deal with the possibility that the system is already in the working state when your wake-up signal occurs. As I discussed earlier, you need to bring your own device out of its sleep state because you won't get a system *SET_POWER* in the near future.

A Modest Proposal

System wake-up appears to have been afflicted with very many bugs across the different releases of WDM operating system platforms, different buses, and different chip sets. Here is some anecdotal evidence that wake-up is pretty well broken on platforms earlier than Windows XP:

- Not long ago, I tested several brands and models of laptop computer, all of them running Windows XP Home or Windows XP Professional, at a superstore before finding one that actually supported system wake-up from a USB device. This feature is supposed to be standardized in the USB specification, and all the machines should have behaved the same way. Not only did just one computer actually work, but it also *stopped* working when I upgraded from Windows XP Home to Windows XP Professional. My developer contacts at Microsoft said, basically, "Gee, that shouldn't have happened."

- I have an old Advanced Power Management (APM)-based laptop on which I installed Windows 2000 as an upgrade. If I plug in a USB mouse and put the machine into standby, the system hangs trying to resume. This occurs because the USB hub driver forces the system into a power state that the BIOS doesn't actually honor, in order not to dishonor a pending *WAIT_WAKE* that wouldn't work anyway because the USB controller on the machine doesn't support wake-up signaling. There are no user interface choices that would let me disable the default behavior of issuing the *WAIT_WAKE* either. Nothing in the DDK suggests cancelling a pending *WAIT_WAKE* before sending a system power query down the stack, but that's the only action that would actually forestall the incorrect choice of standby state. Of course, it wasn't my driver requesting the wake-up either: it was the MOUCLASS driver crafted by a large software company near Seattle. I'm sure that there's plenty of blame to share between the operating system vendor, the laptop vendor, and the various manufacturers of components in the laptop.

- On the same laptop, Windows 98 Second Edition (the operating system installed by the manufacturer) doesn't hang when resuming from standby. Instead, it (surprise!) removes the driver for the mouse, reenumerates the USB bus, and reloads the same driver. It does the same with any USB device that has wake-up enabled when the machine was put into standby. Not issuing the *WAIT_WAKE* causes the system to behave sensibly and not reload the driver. Note that if you had an application using the device when the machine went into standby, the surprise removal would orphan the application's handle. I can predict a flurry of *very* confusing support calls about something like this.

- On a different machine, a defect in the ACPI description of the machine causes *SystemWakeup* to be set to *PowerSystemWorking*, which in turn causes USBHUB to assign a *DeviceState* mapping of *PowerDeviceD2* for *PowerSystemWorking* to *any* USB device. In other words, any driver that believes the device capabilities will find it

IPR2024-00292
KASPERSKY EX 1008 Page 267

impossible to power the device on since it can apparently never rise above D2, even when the system is working!

- Windows 98 Second Edition never seems to complete a *WAIT_WAKE*. Windows Me does, but only long after the wake-up has occurred.
- Prior to Windows XP, HIDCLASS, the standard driver for human input devices such as mice and keyboards (see Chapter 13) issued its *WAIT_WAKE* with the silly *PowerState* value *PowerSystemWorking*. Taken at face value, this means that the system shouldn't wake from any of the standby states. Obviously, no one lower down the stack is paying any attention because the system manifestly will wake up from lower states. Apropos of this observation is a comment in the DDK toaster sample to the effect that the *PowerState* parameter of *WAIT_WAKE* is ignored. So what's a driver writer to do? Set the parameter to *PowerSystemWorking*, set it to the *SystemWake* value from the capabilities, or simply ignore it? Anyway, how could you tell that a mistake in this regard affected the behavior of your driver without exhaustively testing on every conceivable permutation of hardware?

Faced with an apparently insoluble puzzle, I have formulated the following bit of advice: Never arm your device's wake-up feature without providing a user interface element (such as the Device Manager's Power Management tab) that gives the user control over whether you're actually going to use the feature. Set the default for this feature to off (that is, no wake-up) in all platforms prior to Windows XP. In Windows 98/Me, you'll need to provide your own user interface for controlling your wake-up feature since the Device Manager doesn't generate the property page even if you support the WMI controls. At the very least, provide for a registry setting that your tech support people know about.

There! I feel much better having gotten that off my chest!

8.3.2 Powering Off When Idle

As a general matter, the end user would prefer that your device not draw any power if it isn't being used. Your driver might use two schemes (at least) to implement such a policy. You can register with the Power Manager to be sent a low-power device IRP when your device remains idle for a specified period. Alternatively, you can decide to keep your device in a low power state if no handles happen to be open.

Powering Off After an Idle Period

The mechanics of the time-based idle detection scheme involve two service functions: *PoRegisterDeviceForIdleDetection* and *PoSetDeviceBusy*.

To register for idle detection, make this service function call:

```
pdx->idlecount = PoRegisterDeviceForIdleDetection(pdx->Pdo,
  ulConservationTimeout, ulPerformanceTimeout, PowerDeviceD3);
```

The first argument to *PoRegisterDeviceForIdleDetection* is the address of the PDO for your device. The second and third arguments specify timeout periods measured in seconds. The conservation period will apply when the system is trying to conserve power, such as when running on battery power. The performance period will apply when the system is trying to maximize performance, such as when running on AC power. The fourth argument specifies the device power state into which you want your device to be forced if it's idle for longer than whichever of the timeout periods applies.

Indicating That You're Not Idle

The return value from *PoRegisterDeviceForIdleDetection* is the address of a long integer that the system uses as a counter. Every second, the Power Manager increments that integer. If it reaches the appropriate timeout value, the Power Manager sends you a device set-power IRP indicating the power state you registered. At various places in your driver, you'll reset this counter to 0 to restart the idle detection period:

```
if (pdx->idlecount)
  PoSetDeviceBusy(pdx->idlecount);
```

PoSetDeviceBusy is a macro in the WDM.H header file that uncritically dereferences its pointer argument to store a 0. It turns out that *PoRegisterDeviceForIdleDetection* can return a *NULL* pointer, so you should check for *NULL* before calling *PoSetDeviceBusy*.

Now that I've described what *PoSetDeviceBusy* does, you can see that its name is slightly misleading. It doesn't tell the Power Manager that your device is "busy," in which case you'd expect to have to make another call later to indicate that your device is no longer "busy." Rather, it indicates that, at the particular instant you use the macro, your device isn't idle. I'm not making this point as a mere semantic quibble. If your device is busy with some sort of active request, you'll want to have logic that forestalls idle detection. So you might want to call *PoSetDeviceBusy* from many places in your driver: from various dispatch routines, from your *StartIo* routine, and so on. Basically, you want to make sure that the detection period is longer than the longest time that can elapse between the calls to *PoSetDeviceBusy* that you make during the normal processing of a request.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 268

NOTE
PoRegisterSystemState allows you to prevent the Power Manager from changing the system power state, but you can't use it to forestall idle timeouts. Besides, it isn't implemented in Windows 98/Me, so calling it is contraindicated for drivers that need to be portable between Windows XP and Windows 98/Me.

Choosing Idle Timeouts

Picking the idle timeout values isn't necessarily simple. Certain kinds of devices can specify -1 to indicate the standard power policy timeout for their class of device. At the time of this writing, only *FILE_DEVICE_DISK* and *FILE_DEVICE_MASS_STORAGE* devices are in this category. While you'll probably want to have default values for the timeout constants, their values should ultimately be under end user control. Underlying the method by which a user gives you these values is a tale of considerable complexity.

Unless your device is one for which the system designers planned a generic idle detection scheme, you'll need to provide a user-mode component that allows the end user to specify timeout values. To fit in best with the rest of the operating system, that piece should be a property page extension of the Power control panel applet. That is, you should provide a user-mode DLL that implements the *IShellPropSheetExt* and *IShellExtInit* COM interfaces. This DLL will fit the general description of a shell extension DLL, which is the topic you'll research if you want to learn all the ins and outs of writing this particular piece of user interface software.

NOTE
Learning about COM in general and shell extension DLLs in particular seems to me like a case of the tail wagging the dog insofar as driver programming goes. You can download a free Visual Studio application wizard from my Web site (http://www.oneysoft.com) and use it to construct a property-page extension DLL for the Power applet in the Control Panel. You could define a private IOCTL interface between your DLL and your driver for specifying idle timeout constants and other policy values. Alternatively, you could define a custom WMI schema that includes idle timeout functionality. As you'll see in Chapter 10, it's exceptionally easy to use WMI from a scripting language.

Restoring Power

If you implement idle detection, you'll also have to provide a way to restore power to your device at some later time—for example, when you next receive an IRP that requires power. You'll need some relatively complex coding to make this feature work:

- Often you receive substantive IRPs in an arbitrary thread or at an elevated interrupt request level (IRQL), so you can't block the dispatch thread while you restore power to the device.
- Other power or PnP operations can be going on when you receive an IRP that would require you to restore power to your device.
- You might receive more than one IRP requiring power in quick succession. You want to restore power only once, and you don't want to handle the IRPs out of order.
- The IRP that triggers you to restore power might be cancelled while you're waiting for power to come back.

I think the best way to handle all of these complications is to always route IRPs that require power through a *DEVQUEUE*. A skeletal dispatch routine might look like this one:

```
NTSTATUS DispatchReadWrite(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  if (pdx->powerstate > PowerDeviceD0)
    SendDeviceSetPower(pdx, PowerDeviceD0, FALSE);
  IoMarkIrpPending(Irp);
  StartPacket(&pdx->dqReadWrite, fdo, Irp, OnCancel);
  return STATUS PENDING;
  }
```

The idea is to unconditionally queue the IRP after starting a power operation that will finish asynchronously. The power-management code elsewhere in your driver will unstall the queue when power finally comes back, and that will release the IRP.

Powering Off When Handles Close

The other basic strategy for idle power management involves keeping your device in a low power state except while an application has a handle open. Your driver should honor the *MSPower_DeviceEnable* WMI control to decide whether to implement this strategy and should maintain a persistent registry entry that records the end user's most recent specification of

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 269

the value of this control. Let's suppose you define a member of your device extension structure to record the value of *MSPower_DeviceEnable* and another to record the number of open handles:

```
typedef struct  DEVICE EXTENSION {
:
  LONG handles;
  BOOLEAN autopower;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;
Your IRP MJ CREATE and IRP MJ CLOSE dispatch functions will do something like this:
NTSTATUS DispatchCreate(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DevicExtension;
  if (InterlockedIncrement(&pdx->handles) == 1)
    SendDeviceSetPower(fdo, PowerDeviceD0, TRUE);
:
  }

NTSTATUS DispatchClose(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DevicExtension;
  if (InterlockedDecrement(&pdx->handles) == 0 && pdx->autopower)
    SendDeviceSetPower(fdo, PowerDeviceD3, TRUE);
:

}
```

These are *synchronous* calls to *SendDeviceSetPower*, so we don't have to worry about race conditions between the power-up operation initiated by *DispatchCreate* and the power-down operation initiated by the matching call to *DispatchClose*.

When you employ this strategy, you rely on the *DEVQUEUE* or an equivalent package to stall delivery of substantive IRPs to a *StartIo* routine while power is off.

NOTE
A USB driver running in Windows XP or later should use the Selective Suspend protocol instead of directly sending a device IRP to idle the device. Refer to Chapter 12 for details about Selective Suspend. The only change to the sample code shown above is that you don't just call *SendDeviceSetPower*. Instead, you send a special idle registration IRP down the PnP stack with a pointer to a callback routine. The parent driver calls you back when it's time for you to actually put your device into a low power state, whereupon you make the call to *SendDeviceSetPower*. The WAKEUP sample driver illustrates the necessary mechanics.

8.3.3 Using Sequence Numbers to Optimize State Changes

You might want to use an optimization technique in connection with removing and restoring power to your device. Two background facts will help you make sense of the optimization technique. First, the bus driver doesn't always power down a device, even when it receives a device set-power IRP. This particular bit of intransigence arises because of the way computers are wired together. There might be one or more power channels, and any random collection of devices might be wired to any given channel. These devices are said to share a *power relation*. A particular device can't be powered down unless all the other devices on the same power channel are powered down as well. So to use the macabre example that I sometimes give my seminar students, suppose the modem *you* want to power down happens to share a power channel with your computer's heart-lung machine—the system can't power down your modem until the bypass operation is over.

The second background fact is that some devices require a great deal of time to change power. To return to the preceding example, suppose your modem were such a device. At some point, you received and passed along a device set-power request to put your modem to sleep. Unbeknownst to you, however, the bus driver didn't actually power down the modem. When the time came to restore power, you could have saved some time if you had known that your modem hadn't lost power. That's where this particular optimization comes into play.

At the time you remove power, you can create and send a power request with the minor function code *IRP_MN_POWER_SEQUENCE* to the drivers underneath yours. Even though this IRP is technically an *IRP_MJ_POWER*, you use *IoBuildAsynchronousFsdRequest* instead of *PoRequestPowerIrp* to create it. You still use *PoStartNextPowerIrp* and *PoCallDriver* when you handle it, though. The request completes after the bus driver stores three sequence numbers in an array you provide. The sequence numbers indicate how many times your device has been put into the D1, D2, and D3 states. When you're later called upon to restore power, you create and send another *IRP_MN_POWER_SEQUENCE* request to obtain a new set of sequence numbers. If the new set is the same as the set you captured at power-down time, you know that no state change has occurred and that you can bypass whatever expensive process would be required to restore power.

Since *IRP_MN_POWER_SEQUENCE* simply optimizes a process that will work without the optimization, you needn't use it. Furthermore, the bus driver needn't support it, and you shouldn't treat failure of a power-sequence request as indicative of any sort of error. The GENERIC sample in the companion content actually includes code to use the optimization, but I didn't want

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 270

to further complicate the textual discussion of the state machine by showing it here.

8.4 Windows 98/Me Compatibility Notes

Windows 98/Me incompletely implements many power-management features. Consequently, the Windows 98/Me environment will forgive your mistakes more readily than Windows XP will, facilitating the initial development of a driver. But since Windows 98/Me tolerates mistakes that Windows XP won't tolerate, you must be sure to test all of your driver's power functionality under Windows XP.

8.4.1 The Importance of *DO_POWER_PAGABLE*

The *DO_POWER_PAGABLE* flag has additional and unexpected significance in Windows 98/Me. Unless every device object, including the PDO and all filter devices, in your particular stack has this flag set, the I/O Manager tells the Windows 98/Me Configuration Manager that the device supports only the D0 power state and is incapable of waking the system. Thus, an additional consequence of not setting the *DO_POWER_PAGABLE* flag is that any idle notification request you make by calling *PoRegisterDeviceForIdleDetection* is effectively ignored—that is, you'll never receive a power IRP as a result of being idle too long. Another consequence is that your device's wake-up feature, if any, won't be used.

8.4.2 Completing Power IRPs

You must complete power set and query requests in Windows 98/Me at *PASSIVE_LEVEL* only. If you look carefully at the power-management code for GENERIC in the companion content, you'll find it scheduling a work item instead of completing the IRP at *DISPATCH_LEVEL*.

If you copy my code, or if you use GENERIC.SYS, you may have a bit of explaining to do when you submit your driver to Windows Hardware Quality Lab (WHQL). Windows 98/Me does not support the *IoXxxWorkItem* functions that Windows 2000 and later systems provide, so it's necessary to use the older *ExXxxWorkItem* functions. Unfortunately, the WHQL tests spot the symbol import from your driver instead of performing a run-time test to see which function you actually call. My sample code has a run-time test and therefore completely meets the spirit of the test, just not the letter of the law. If enough of my readers ask for exceptions, maybe WHQL will change the tests. See Chapter 14 for information about work items and Chapter 15 for information about WHQL.

8.4.3 Requesting Device Power IRPs

As previously discussed, Windows 98/Me has a bug whereby *PoRequestPowerIrp* can appear to succeed—that is, it returns *STATUS_PENDING*—without actually causing you to receive a device set-power IRP. The problem arises when you ask for a set-power IRP that specifies the same device state that your device is already in—the Windows 98/Me Configuration Manager "knows" that there's no news to report by sending a configuration event to the configuration function that NTKERN operates on your behalf. Mind you, if you're waiting for a device IRP to complete, your device will simply stop responding at this point.

I used an obvious workaround to overcome this problem: if we detect that we're about to request a device power IRP for the same power state that the device already occupies, I simply pretend that the device IRP succeeded. In terms of the state transitions that *HandlePowerEvent* goes through, I jump from *SendDeviceIrp* directly to whichever action (*SubPowerUpComplete* or *SubPowerDownComplete*) is appropriate.

8.4.4 PoCallDriver

PoCallDriver just calls *IoCallDriver* in Windows 98/Me. Consequently, it would be easy for you to make the mistake of using *IoCallDriver* to forward power IRPs. There is, however, an even worse problem in Windows 98/Me.

The Windows XP version of *PoCallDriver* makes sure that it sends power IRPs to *DO_POWER_PAGABLE* drivers at *PASSIVE_LEVEL* and to *INRUSH* or nonpaged drivers at *DISPATCH_LEVEL*. I took advantage of that fact in GENERIC to forward power IRPs in situations in which *HandlePowerEvent* is called at *DISPATCH_LEVEL* from an I/O completion routine. The Windows 98/Me version, since it's just *IoCallDriver* under a different name, doesn't switch IRQL. As it happens, *all* power IRPs in Windows 98/Me should be sent at *PASSIVE_LEVEL*. So I wrote a helper routine named *SafePoCallDriver* for use in GENERIC that queues an executive work item to send the IRP at *PASSIVE_LEVEL*. The implications of using a work item in this situation are the same as discussed just above in connection with completing Power IRPs.

8.4.5 Other Differences

You should know about a few other differences between the way Windows 98/Me and Windows XP handle power-management features. I'll describe them briefly and indicate how they might affect the development of your drivers.

When you call *PoRegisterDeviceForIdleDetection*, you must supply the address of the PDO rather than your own device object. That's because, internally, the system needs to find the address of the *DEVNODE* that the Windows 98/Me Configuration Manager works with, and that's accessible only from the PDO. You can also use the PDO as the argument in Windows XP, so you might as well write your code that way in the first place.

IPR2024-00292
KASPERSKY EX 1008 Page 271

The *PoSetPowerState* support routine is a no-operation in Windows 98/Me. Furthermore, although it's documented as returning the previous device or system power state, the Windows 98/Me version returns whatever state argument you happen to supply. This is the *new* state rather than the old state—or maybe just a random number that occupies an uninitialized variable that you happened to use as an argument to the function: no one checks.

PoStartNextPowerIrp is a no-operation in Windows 98/Me, so it's easy for you to forget to call it if you do your development in Windows 98/Me.

The service routines having to do with device power relations (*PoRegisterDeviceNotify* and *PoCancelDeviceNotify*) aren't defined in Windows 98/Me. As far as I can tell, Windows 98/Me also doesn't issue a *PowerRelations* query to gather the information needed to support the callbacks in the first place. The service routines *PoRegisterSystemState*, *PoSetSystemState*, and *PoUnregisterSystemState* are also not implemented in Windows 98/Me. To load a driver in Windows 98/Me that calls these or other undefined service functions, you'll need to employ a technique, like the WDMSTUB.SYS filter driver described in Appendix A, for defining the missing functions.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 272

Chapter 9

I/O Control Operations

If you look at the various types of requests that come to a device, most of them involve reading or writing data. On occasion, however, an application needs to communicate "out of band" with the driver. For most types of device, the application can use the standard MicrosoftWin32 API function *DeviceIoControl*. On the driver side, an application's call to *DeviceIoControl* turns into an I/O request packet (IRP) with the major function code *IRP_MJ_DEVICE_CONTROL*.

I'll discuss the user-mode and kernel-mode sides of *DeviceIoControl* in this chapter. With several specific types of device, however, an application isn't supposed to (or can't) use *DeviceIoControl* to talk to a driver. See Table 9-1.

Driver Type	Alternative to DeviceIoControl
Human Interface Device (HID) minidriver (see Chapter 13)	*HidD_GetFeature, HidD_SetFeature*
SCSI miniport driver	*IOCTL_SCSI_PASS_THROUGH*
Network Driver Interface Specification (NDIS) miniport driver	WMI request using custom GUID. You're on your own for Win98 Gold, where WMI doesn't work.
SmartCard reader driver (Interface Device [IFD] Handler in PC/SC terms)	*ScardControl*

Table 9-1. *Alternatives to DeviceIoControl for Certain Types of Driver*

There is also a special problem associated with using *DeviceIoControl* to communicate with a filter driver. I'll discuss that problem and its solution in Chapter 16.

9.1 The *DeviceIoControl* API

The user-mode DeviceIoControl API has the following prototype:

```
result = DeviceIoControl(Handle, Code, InputData, InputLength,
  OutputData, OutputLength, &Feedback, &Overlapped);
```

Handle (*HANDLE*) is an open handle open to the device. You obtain this handle by calling *CreateFile* in the following manner:

```
Handle = CreateFile("\\\\.\\IOCTL", GENERIC READ | GENERIC WRITE,
  0, NULL, OPEN EXISTING, flags, NULL);
if (Handle == INVALID HANDLE VALUE)
  <error>:
CloseHandle(Handle);
```

The *flags* argument to *CreateFile* is either *FILE_FLAG_OVERLAPPED* or 0 to indicate whether you'll be performing asynchronous operations with this file handle. While you have the handle open, you can make calls to *ReadFile*, *WriteFile*, or *DeviceIoControl*. When you're done accessing the device, you should explicitly close the handle by calling *CloseHandle*. Bear in mind, though, that the operating system automatically closes any handles that are left open when your process terminates.

The *Code* (*DWORD*) argument to *DeviceIoControl* is a control code that indicates the control operation you want to perform. I'll discuss how you define these codes a bit further on (in "Defining I/O Control Codes"). The *InputData* (*PVOID*) and *InputLength* (*DWORD*) arguments describe a data area that you're sending *to* the device driver. (That is, this data is input from the perspective of the driver.) The *OutputData* (*PVOID*) and *OutputLength* (*DWORD*) arguments describe a data area that the driver can completely or partially fill with information that it wants to send back to you. (That is, this data is output from the perspective of the driver.) The driver will update the *Feedback* variable (a *DWORD*) to indicate how many bytes of output data it gave you back. Figure 9-1 illustrates the relationship of these buffers with the application and the driver. The *Overlapped* (*OVERLAPPED*) structure is used to help control an asynchronous operation, which is the subject of the next section. If you specified *FILE_FLAG_OVERLAPPED* in the call to *CreateFile*, you must specify the *OVERLAPPED* structure pointer. If you didn't specify *FILE_FLAG_OVERLAPPED*, you might as well supply *NULL* for this last argument because the system is going to ignore it anyway.

IPR2024-00292
KASPERSKY EX 1008 Page 273

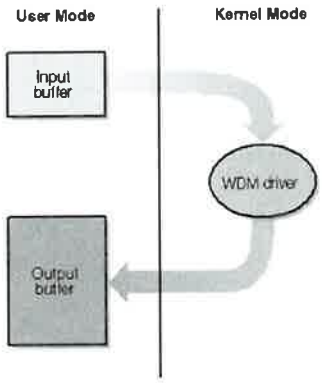


Figure 9-1. *Input and output buffers for DeviceIoControl.*

Whether a particular control operation requires an input buffer or an output buffer depends on the function being performed. For example, an I/O control (IOCTL) that retrieves the driver's version number probably requires an output buffer only. An IOCTL that merely notifies the driver of some fact pertaining to the application probably requires only an input buffer. You can imagine still other operations that require either both or neither of the input and output buffers—it all depends on what the control operation does.

The return value from *DeviceIoControl* is a Boolean value that indicates success (if *TRUE*) or failure (if *FALSE*). In a failure situation, the application can call *GetLastError* to find out why the call failed.

9.1.1 Synchronous and Asynchronous Calls to *DeviceIoControl*

When you make a synchronous call to *DeviceIoControl*, the calling thread blocks until the control operation completes. For example:

```
HANDLE Handle = CreateFile("\\\\.\\IOCTL", ..., 0, NULL);
DWORD version, junk;
if (DeviceIoControl(Handle, IOCTL GET VERSION BUFFERED,
  NULL, 0, &version, sizeof(version), &junk, NULL))
  printf("IOCTL.SYS version %d.%2.2d\n", HIWORD(version),
    LOWORD(version));
else
  printf("Error %d in IOCTL_GET_VERSION_BUFFERED call\n", GetLastError());
```

Here we open the device handle without the *FILE_FLAG_OVERLAPPED* flag. Our subsequent call to *DeviceIoControl* therefore doesn't return until the driver supplies the answer we're asking for.

When you make an asynchronous call to *DeviceIoControl*, the calling thread doesn't block immediately. Instead, it continues processing until it reaches the point where it requires the result of the control operation. At that point, it calls an API that will block the thread until the driver completes the operation. For example:

```
HANDLE Handle = CreateFile("\\\\.\\IOCTL", ...,
  FILE FLAG OVERLAPPED, NULL);
DWORD version, junk;
OVERLAPPED Overlapped;

Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
DWORD code;

if (DeviceIoControl(Handle, ..., &Overlapped))
  code = 0;
else
  code = GetLastError();

<continue processing>
if (code == ERROR IO PENDING)
  {
  if (GetOverlappedResult(Handle, &Overlapped, &junk, TRUE))
    code = 0;
  else
    code = GetLastError();
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 274

```
  }
CloseHandle(Overlapped.hEvent);
if (code != 0)
  <error>
```

Two major differences exist between this asynchronous example and the earlier synchronous example. First, we specify the *FILE_FLAG_OVERLAPPED* flag in the call to *CreateFile*. Second, the call to *DeviceIoControl* specifies the address of an *OVERLAPPED* structure, within which we've initialized the *hEvent* event handle to describe a manual reset event. (For more information about events and thread synchronization in general, see Jeffrey Richter's *Programming Applications for Microsoft Windows*, Fourth Edition [Microsoft Press, 1999].)

The asynchronous call to *DeviceIoControl* will have one of three results. First, it might return *TRUE*, meaning that the device driver's dispatch routine was able to complete the request right away. Second, it might return *FALSE*, and *GetLastError* might retrieve the special error code *ERROR_IO_PENDING*. This result indicates that the driver's dispatch routine returned *STATUS_PENDING* and will complete the control operation later. Note that *ERROR_IO_PENDING* isn't really an error—it's one of the two ways in which the system indicates that everything is proceeding normally. The third possible result from the asynchronous call to *DeviceIoControl* is a *FALSE* return value coupled with a *GetLastError* value other than *ERROR_IO_PENDING*. Such a result would be a real error.

At the point at which the application needs the result of the control operation, it calls one of the Win32 synchronization primitives—*GetOverlappedResult*, *WaitForSingleObject*, or the like. *GetOverlappedResult*, the synchronization primitive I use in this example, is especially convenient because it also retrieves the bytes-transferred feedback value and sets the *GetLastError* result to indicate the result of the I/O operation. Although you can call *WaitForSingleObject* or a related API—passing the *Overlapped.hEvent* event handle as an argument—you won't be able to learn the results of the *DeviceIoControl* operation; you'll just learn that the operation has finished.

9.1.2 Defining I/O Control Codes

The *Code* argument to *DeviceIoControl* is a 32-bit numeric constant that you define using the *CTL_CODE* preprocessor macro that's part of both the DDK and the Platform SDK. Figure 9-2 illustrates the way in which the operating system partitions one of these 32-bit codes into subfields.



Figure 9-2. *Fields in an I/O control code.*

The fields have the following interpretation:

- The device type (16 bits, first argument to *CTL_CODE*) indicates the type of device that implements this control operation. You should use the same value (for example, *FILE_DEVICE_UNKNOWN*) that you use in the driver when you call *IoCreateDevice*. (File system device type codes cause the I/O Manager to use a different major function code for the IRP it sends you.)
- The access code (2 bits, fourth argument to *CTL_CODE*) indicates the access rights an application needs to its device handle to issue this control operation.
- The function code (12 bits, second argument to *CTL_CODE*) indicates precisely which control operation this code describes. Microsoft reserves the first half of the range of this field—that is, values 0 through 2047—for standard control operations. You and I therefore assign values in the range 2048 through 4095. The main purpose of this convention is to allow you to define private control operations for standard devices.
- The buffering method (2 bits, third argument to *CTL_CODE*) indicates how the I/O Manager is to handle the input and output buffers supplied by the application. I'll have a great deal to say about this field in the next section, when I describe how to implement *IRP_MJ_DEVICE_CONTROL* in a driver.

I want to clarify one point of possible confusion. When you create your driver, you're free to design a series of IOCTL operations that applications can use in talking to your driver. Although some other driver author might craft a set of IOCTL operations that uses exactly the same numeric values for control codes, the system will never be confused by the overlap because IOCTL codes are interpreted by only the driver to which they're addressed. Mind you, if you opened a handle to a device belonging to that hypothetical other driver and then tried to send what you thought was one of your own IOCTLs to it, confusion would definitely ensue.



The access code in an I/O control code gives you the ability to divide the world of users into four parts, based on a security descriptor you attach to your device object:

- Users to whom the security descriptor denies all access can't open a handle, so they can't issue any IOCTLs at all.
- Users whom the security descriptor allows to open handles for reading but not writing can issue IOCTLs whose function code specifies *FILE_READ_ACCESS* or *FILE_ANY_ACCESS*, but not those whose code specifies *FILE_WRITE_ACCESS*.

IPR2024-00292
KASPERSKY EX 1008 Page 275

- Users whom the security descriptor allows to open handles for writing but not reading can issue IOCTLs whose function code specifies *FILE_WRITE_ACCESS* or *FILE_ANY_ACCESS*, but not those whose code specifies *FILE_READ_ACCESS*.
- Users whom the security descriptor allows to open handles for both reading and writing can issue any IOCTL.

Mechanically, your life and the life of application programmers who need to call your driver will be easier if you place all of your IOCTL definitions in a dedicated header file. In the samples in the companion content, the projects each have a header named IOCTLS.H that contains these definitions. For example:

```
#ifndef CTL_CODE
  #pragma message ( \
    "CTL_CODE undefined. Include winioctl.h or wdm.h")
#endif

#define IOCTL_GET_VERSION_BUFFERED \
  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, \
  FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_DIRECT \
  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_OUT_DIRECT, \
  FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_NEITHER \
  CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER, \
  FILE_ANY_ACCESS)
```

The reason for the message *#pragma*, by the way, is that I'm forever forgetting to include the header file (WINIOCTL.H) that defines *CTL_CODE* for user-mode programs, and I also tend to forget the name. Better a message that will tell me what I'm doing wrong than a few minutes grep'ing through the include directory, I always say.

9.2 Handling IRP_MJ_DEVICE_CONTROL

Each user-mode call to *DeviceIoControl* causes the I/O Manager to create an IRP with the major function code *IRP_MJ_DEVICE_CONTROL* and to send that IRP to the driver dispatch routine at the top of the stack for the addressed device. The top stack location contains the parameters listed in Table 9-2. Filter drivers might interpret some private codes themselves but will—if correctly coded, that is—pass all others down the stack. A dispatch function that understands how to handle the IOCTL will reside somewhere in the driver stack—most likely in the function driver, in fact.

Parameters.DeviceIoControl Field	Description
OutputBufferLength	Length of the output buffer—sixth argument to *DeviceIoControl*
InputBufferLength	Length of the input buffer—fourth argument to *DeviceIoControl*
IoControlCode	Control code—second argument to *DeviceIoControl*
Type3InputBuffer	User-mode virtual address of input buffer for *METHOD_NEITHER*

Table 9-2. *Stack Location Parameters for IRP_MJ_DEVICE_CONTROL*

A skeletal dispatch function for control operations looks like this:

```
#pragma PAGEDCODE

NTSTATUS DispatchControl(PDEVICE_OBJECT fdo, PIRP Irp)
{
1
  PAGED_CODE();
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = STATUS_SUCCESS;
  ULONG info = 0;

2
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG cbin =
    stack->Parameters.DeviceIoControl.InputBufferLength;
  ULONG cbout =
    stack->Parameters.DeviceIoControl.OutputBufferLength;
  ULONG code =
    stack->Parameters.DeviceIoControl.IoControlCode;

  switch (code)
    {
```


IPR2024-00292
KASPERSKY EX 1008 Page 276

3

```
    default:
      status = STATUS_INVALID_DEVICE_REQUEST;
      break;

    }

    return CompleteRequest(Irp, status, info);
    }
```

1. You can be sure of being called at *PASSIVE_LEVEL*, so there's no particular reason for a simple dispatch function to be anywhere but paged memory.
2. The next few statements extract the function code and buffer sizes from the parameters union in the I/O stack. You often need these values no matter which specific IOCTL you're processing, so I find it easier always to include these statements in the function.
3. This is where you get to exercise your own creativity by inserting *case* labels for the various IOCTL operations you support.
4. It's a good idea to return a meaningful status code if you're given an IOCTL operation you don't understand.



IMPORTANT
Always switch on the full 32 bits of the I/O control code to prevent a user-mode program from sneaking past the access checks or causing the I/O Manager to prepare the parameters using the wrong buffering method.

The way you handle each IOCTL depends on two factors. The first and most important of these is the actual purpose of the IOCTL in your scheme of things. (Duh.) The second factor, which is critically important to the mechanics of your code, is the method you selected for buffering user-mode data.

In Chapter 7, I discussed how you work with a user-mode program sending you a buffer load of data for output to your device or filling a buffer with input from your device. As I indicated there, when it comes to read and write requests, you have to make up your mind at *AddDevice* time whether you're going to use the so-called buffered method or the direct method (or neither of them) for accessing user-mode buffers in all read and write requests. Control requests also use one of these addressing methods, but they work a little differently. Rather than specify a global addressing method via device-object flags, you specify the addressing method for each IOCTL by means of the 2 low-order bits of the function code. Consequently, you can have some IOCTLs that use the buffered method, some that use a direct method, and some that use neither method. Moreover, the methods you pick for IOCTLs don't affect in any way how you address buffers for read and write IRPs.

You choose one or the other buffering method based on several factors. Most IOCTL operations transfer much less than a page worth of data in either direction and therefore use the *METHOD_BUFFERED* method. Operations that will transfer more than a page of data should use one of the direct methods. The names of the direct methods seem to oppose common sense: you use *METHOD_IN_DIRECT* if the application is sending data to the driver and *METHOD_OUT_DIRECT* if it's the other way around. If the application needn't transfer any data at all, *METHOD_NEITHER* would be your best choice.

9.3 METHOD_BUFFERED

With *METHOD_BUFFERED*, the I/O Manager creates a kernel-mode copy buffer big enough for the larger of the user-mode input and output buffers. When your dispatch routine gets control, the user-mode input data is sitting in the copy buffer. Before completing the IRP, you fill the copy buffer with the output data you want to send back to the application. When you complete the IRP, you set the *IoStatus.Information* field equal to the number of output bytes you put into the copy buffer. The I/O Manager then copies that many bytes of data back to user mode and sets the feedback variable equal to that same count. Figure 9-3 illustrates these copy operations.

Inside the driver, you access both buffers at the same address—namely, the *AssociatedIrp.SystemBuffer* pointer in the IRP. Once again, this is a kernel-mode virtual address that points to a copy of the input data. It obviously behooves you to finish processing the input data before you overwrite this buffer with output data. (I hardly need to tell you—it's the kind of mistake you'll make only once.)

IPR2024-00292
KASPERSKY EX 1008 Page 277

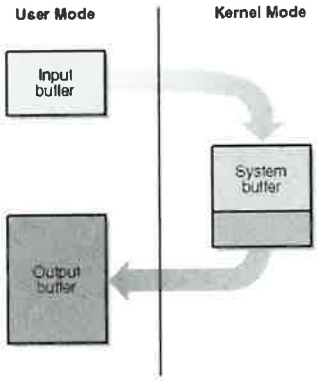


Figure 9-3. *Buffer management with METHOD_BUFFERED.*

Here's a simple example, drawn from the IOCTL sample program, of the code-specific handling for a *METHOD_BUFFERED* operation:

```
case IOCTL_GET_VERSION_BUFFERED:
  {
  if (cbout < sizeof(ULONG))
    {
    status = STATUS_INVALID_BUFFER_SIZE;
    break;
    }
  PULONG pversion = (PULONG) Irp->AssociatedIrp.SystemBuffer;
  *pversion = 0x0004000A;
  info = sizeof(ULONG);
  break;
  }
```

We first verify that we've been given an output buffer at least long enough to hold the doubleword we're going to store there. Then we use the *SystemBuffer* pointer to address the system copy buffer, in which we store the result of this simple operation. The *info* local variable ends up as the *IoStatus.Information* field when the surrounding dispatch routine completes this IRP. The I/O Manager copies that much data from the system copy buffer back to the user-mode buffer.



Always check the length of the buffers you're given with an *IRP_MJ_DEVICE_CONTROL*, at least when the IRP's *RequestorMode* isn't equal to *KernelMode*. With *METHOD_BUFFERED* and the two *METHOD_XXX_DIRECT* methods, the I/O Manager will verify that the address and the length of the input and output buffers are valid, but *you* are the only one who knows how long the buffers should be.

9.3.1 The *DIRECT* Buffering Methods

Both *METHOD_IN_DIRECT* and *METHOD_OUT_DIRECT* are handled the same way in the driver. They differ only in the access rights required for the user-mode buffer. *METHOD_IN_DIRECT* needs read access; *METHOD_OUT_DIRECT* needs read and write access. With both of these methods, the I/O Manager provides a kernel-mode copy buffer (at *AssociatedIrp.SystemBuffer*) for the input data and an MDL for the output data buffer. Refer to Chapter 7 for all the gory details about MDLs and to Figure 9-4 for an illustration of this method of managing the buffers.

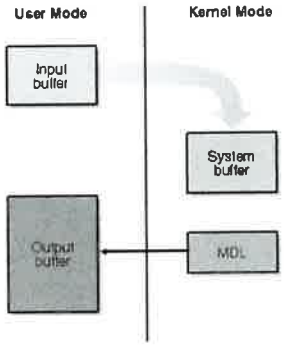


Figure 9-4. *Buffer management with METHOD_XXX_DIRECT.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 278

Here's an example of a simple handler for a *METHOD_XXX_DIRECT* request:

```
case IOCTL_GET_VERSION_DIRECT:
  {
  if (cbout < sizeof(ULONG))
    {
    status = STATUS_INVALID_BUFFER_SIZE;
    break;
    }
  PULONG pversion = (PULONG)
    MmGetSystemAddressForMdlSafe(Irp->MdlAddress);
  *pversion = 0x0004000B;
  info = sizeof(ULONG);
  break;
  }
```

The only substantive difference between this example and the previous one is the bold line. (I also altered the reported version number so that I could easily know I was invoking the correct IOCTL from the test program.) With either *DIRECT*-method request, we use the MDL pointed to by the *MdlAddress* field of the IRP to access the user-mode output buffer. You can do direct memory access (DMA) using this address. In this example, I just called *MmGetSystemAddressForMdlSafe* to get a kernel-mode alias address pointing to the physical memory described by the MDL.

TIP
To achieve binary portability, use the portable workaround for *MmGetSystemAddressForMdlSafe* that I described in Chapter 7.

9.3.2 METHOD_NEITHER

With *METHOD_NEITHER*, the I/O Manager doesn't try to translate the user-mode virtual addresses in any way. Consequently, you most often use *METHOD_NEITHER* when you don't need to transfer any data into or out of the driver. *IOCTL_SERIAL_SET_DTR*, for example, is a standard serial port IOCTL for setting the Data Terminal Ready (DTR) signal line. It's defined to use *METHOD_NEITHER* because it has no data.

It's possible to use *METHOD_NEITHER* when you *do* have data, however, provided you follow some rules about pointer validation. You get (in the *Type3InputBuffer* parameter in the stack location) the user-mode virtual address of the input buffer, and you get (in the *UserBuffer* field of the IRP) the user-mode virtual address of the output buffer. Neither address is of any use unless you know you're running in the same process context as the user-mode caller. If you *do* know you're in the right process context, you can just directly dereference the pointers:

```
case IOCTL_GET_VERSION_NEITHER:
  {
  if (cbout < sizeof(ULONG))
    {
    status = STATUS_INVALID_BUFFER_SIZE;
    break;
    }
  PULONG pversion = (PULONG) Irp->UserBuffer;
  if (Irp->RequestorMode != KernelMode)
    {
    __try
      {
      ProbeForWrite(pversion, sizeof(ULONG), 1);
      *pversion = 0x0004000A;
      }
    __except(EXCEPTION_EXECUTE_HANDLER)
      {
      status = GetExceptionCode();
      break;
      }
    }
  else
    *pversion = 0x0004000A;
  info = sizeof(ULONG);
  break;
  }
```



As shown in the preceding code in boldface, the only real glitch here is that you want to make sure that it's OK to write into any buffer you get from an untrusted source. Refer to Chapter 3 if you're rusty about structured exceptions.

IPR2024-00292
KASPERSKY EX 1008 Page 279

ProbeForWrite is a standard kernel-mode service routine for testing whether a given user-mode virtual address can be written. The second argument indicates the length of the data area you want to probe, and the third argument indicates the alignment you require for the data area. In this example, we want to be sure that we can access 4 bytes for writing, but we're willing to tolerate single-byte alignment for the data area itself. What *ProbeForWrite* (and its companion function *ProbeForRead*) actually tests is whether the given address range has the correct alignment and occupies the user-mode portion of the address space—it doesn't actually try to write to (or read from) the memory in question.

Additionally, you must perform the access within a structured exception frame. If any portion of the buffer happens to belong to nonexistent pages at the time of the access, the memory manager will raise an exception instead of immediately bugchecking. Your exception handler will backstop the exception and prevent the system from crashing.

9.3.3 Designing a Safe and Secure IOCTL Interface

How you design the IOCTL interface to your driver can have a big impact on the security and robustness of systems that eventually run your code. You've probably noticed that this chapter has a lot of the little Security icons we're using to flag potential security problems. That's because it's pretty easy for sloppy coding to open unforeseen security holes or to unintentionally compromise system integrity.

So, in addition to all the other things I'm pointing out in this chapter, here are some more things to consider as you design an IOCTL interface for your driver.

- Don't assume that the only caller of your IOCTL interface will be your own application, which you have (of course!) crafted to supply only valid parameters. Cyberterrorists use the same technique that a housefly uses to break in: they buzz around the screen door until they find a hole. If your driver has a hole, hackers will find it and publicize it so widely that everyone who cares will know.
- Don't pass null-terminated strings as arguments to an IOCTL. Provide a count instead. That way, the driver won't risk walking off the end of a page looking for a null terminator that happens not to be there.
- Don't put pointers inside the structures you use in IOCTL calls. Instead, package all the data for a particular call into a single buffer that contains internal offset pointers. Take the time in the driver to validate the offsets and lengths with respect to the overall length of the parameter structure.
- Don't write the equivalent of *IOCTL_POKE_KERNEL_MEMORY*. That is, don't invent some sort of utility control operation whose job is to write to kernel memory. Really—don't do this, even in the debug version of your driver, because debug versions have been known to make it out of the lab.
- Be careful with hardware pass-through IOCTLs. Maybe your application has a need to directly communicate with dedicated hardware, and maybe some sort of pass-through operation is the best way to provide this functionality. Just be careful how much functionality you open up.
- Avoid making one IOCTL dependent on state information left over from some preceding operation. The one invariant rule about persistent state information is that it isn't. Persistent, that is. Something always goes wrong to clobber the data you were sure would stay put, so try to design IOCTL operations to be as nearly self-contained as you can.
- It's better not to use *METHOD_NEITHER* for control operations that involve data transfer because of two risks. First, you might forget to do all the buffer validation that's required to avoid security holes. Second, somewhere down the road, someone might forget that the IOCTL used *METHOD_NEITHER* and call you in the wrong thread context. If you and the programmers that follow in your footsteps are supremely well organized, these problems won't arise, but one component of successful engineering is to take account of human foibles.
- Above all, don't assume that no one will try to compromise the system through your driver. There need be only one supremely evil person in the world for it to be dangerous to trust everyone, and there are many more than just one. It's not an exaggeration to suggest that actual lives may be on the line when the enemies of civilization systematically exploit weaknesses in the world's most prevalent operating system.

You can use the DEVCTL tool in the DDK to help you test your IOCTL interface for robustness, by the way. This tool will send random IOCTLs and well-formed IOCTLs with bad parameters to your driver in an attempt to provoke a failure. This sort of attack mimics what a script kiddie will do as soon as he or she lays hands on your driver, so you *should* run this test yourself anyway.

Since there are limits on how cunning a generic tool such as DEVCTL can be, I also recommend that you build your own test program to thoroughly explore all the boundary conditions in your IOCTL interface. Here are some ideas for things to test:

- Invalid function codes. Don't go nutty here since DEVCTL will do this type of testing exhaustively.
- Function codes with mistakes in one or more of the four fields (device type, access mask, function code, and buffering method).
- Missing or extraneous input or output buffers.
- Buffers that are too short but not altogether missing.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 280

- Simultaneous operations from different threads using the same handle and using different handles.

9.4 Internal I/O Control Operations

The system uses *IRP_MJ_DEVICE_CONTROL* to implement a *DeviceIoControl* call from user mode. Drivers sometimes need to talk to each other too, and they use the related *IRP_MJ_INTERNAL_DEVICE_CONTROL* to do so. A typical code sequence is as follows:

```
ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
KEVENT event;
KeInitializeEvent(&event, NotificationEvent, FALSE);
IO_STATUS_BLOCK iostatus;
PIRP Irp = IoBuildDeviceIoControlRequest(IoControlCode,
  DeviceObject, pInBuffer, cbInBuffer, pOutBuffer, cbOutBuffer,
  TRUE, &event, &iostatus);
NTSTATUS status = IoCallDriver(DeviceObject, Irp);
if (NT_SUCCESS(status))
  KeWaitForSingleObject(&event, Executive, KernelMode,
    FALSE, NULL);
```

Being at *PASSIVE_LEVEL* is a requirement for calling *IoBuildDeviceIoControlRequest* as well as for blocking on the event object as shown here.

The *IoControlCode* argument to *IoBuildDeviceIoControlRequest* is a control code expressing the operation you want the target device driver to perform. This code is the same kind of code you use with regular control operations. *DeviceObject* is a pointer to the *DEVICE_OBJECT* whose driver will perform the indicated operation. The input and output buffer parameters serve the same purpose as their counterparts in a user-mode *DeviceIoControl* call. The seventh argument, which I specified as *TRUE* in this fragment, indicates that you're building an internal control operation. (You could say *FALSE* here to create an *IRP_MJ_DEVICE_CONTROL* instead.) I'll describe the purpose of the *event* and *iostatus* arguments in a bit.

IoBuildDeviceIoControlRequest builds an IRP and initializes the first stack location to describe the operation code and buffers you specify. It returns the IRP pointer to you so that you can do any additional initialization that might be required. In Chapter 12, for example, I'll show you how to use an internal control request to submit a USB request block (URB) to the USB driver. Part of that process involves setting a stack parameter field to point to the URB. You then call *IoCallDriver* to send the IRP to the target device. If the return value passes the NT_SUCCESS test, you wait on the *event* object you specified as the eighth argument to *IoBuildDeviceIoControlRequest*. The I/O Manager will set the event when the IRP finishes, and it will also fill in your *iostatus* structure with the ending status and information values. Finally it will call *IoFreeIrp* to release the IRP. Consequently, you don't want to access the IRP pointer at all after you call *IoCallDriver*.



CAUTION

When you use automatic variables for the event and status-block arguments to *IoBuildDeviceIoControlRequest* or *IoBuildSynchronousFsdRequest*, you must wait for the event to be signaled if *IoCallDriver* returns STATUS_PENDING. Otherwise, you risk allowing the event and status block to pass out of scope before the I/O Manager finishes completing the IRP. You can wait if *IoCallDriver* returns a success code, but the wait ought to complete immediately because, in that case, the IRP has already been totally completed. Don't wait, however, if *IoCallDriver* returns an error code because, in an error case, the I/O Manager doesn't set the event and doesn't touch the status block when it completes the IRP.

Since internal control operations require cooperation between two drivers, fewer rules about sending them exist than you'd guess from what I've just described. You don't have to use *IoBuildDeviceIoControlRequest* to create one of them, for example: you can just call *IoAllocateIrp* and perform your own initialization. Provided the target driver isn't expecting to handle internal control operations solely at *PASSIVE_LEVEL*, you can also send one of these IRPs at *DISPATCH_LEVEL*, say, from inside an I/O completion or a deferred procedure call (DPC) routine. (Of course, you can't use *IoBuildDeviceIoControlRequest* in such a case, and you can't wait for the IRP to finish. But you can *send* the IRP because *IoAllocateIrp* and *IoCallDriver* can run at *DISPATCH_LEVEL* or below.) You don't even have to use the I/O stack parameter fields exactly as you would for a regular IOCTL. In fact, calls to the USB driver use the field that would ordinarily be the output buffer length to hold the URB pointer. So if you're designing an internal control protocol for two of your own drivers, just think of *IRP_MJ_INTERNAL_DEVICE_CONTROL* as being an envelope for whatever kind of message you want to send.



It's not a good idea to use the same dispatch routine for internal and external control operations, by the way, at least not without checking the major function code of the IRP. Here's an example of why not: Suppose your driver has an external control interface that allows an application to query the version number of your driver *and* an internal control interface that allows a trusted kernel-mode caller to determine a vital secret that you don't want to share with user-mode programs. Then suppose you use one routine to handle both interfaces, as in this example:

```
NTSTATUS DriverEntry(...)
  {
```


IPR2024-00292
KASPERSKY EX 1008 Page 281

```
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
    DispatchControl;
  DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] =
    DispatchControl;
  .
  .
  .
  }

NTSTATUS DispatchControl(...)
  {
  .
  .
  .
  switch (code)
    {
  case IOCTL_GET_VERSION:
  .
  .
  .
  case IOCTL_INTERNAL_GET_SECRET:
            // <== exposed for user-mode calls
    }
  }
```

If an application is able to somehow determine the numeric value of *IOCTL_INTERNAL_GET_SECRET*, it can issue a regular *DeviceIoControl* call and bypass the intended security on that function.

9.5 Notifying Applications of Interesting Events

One extremely important use of IOCTL operations is to give a WDM driver a way to notify an application that an interesting event (however you define "interesting," that is) has occurred. To motivate this discussion, suppose you had an application that needed to work closely with your driver in such a way that whenever a certain kind of hardware event occurred your driver would alert the application so that it could take some sort of user-visible action. For example, a button press on an instrument might trigger an application to begin collecting and displaying data. Whereas Windows 98/Me provides a couple of ways for a driver to signal an application in this kind of situation—namely, asynchronous procedure calls or posted window messages—those methods don't work in Windows XP because the operating system lacks (or doesn't expose) the necessary infrastructure to make them work.

A WDM driver can notify an application about an interesting event in two ways:

- The application can create an event that it shares with the driver. An application thread waits on the event, and the driver sets the event when something interesting happens.
- The application can issue a *DeviceIoControl* that the driver pends by returning *STATUS_PENDING*. The driver completes the IRP when something interesting happens.

In either case, the application typically dedicates a thread to the task of waiting for the notification. That is, when you're sharing an event, you have a thread that spends most of its life asleep on a call to *WaitForSingleObject*. When you use a pending IOCTL, you have a thread that spends most of *its* life waiting for *DeviceIoControl* to return. This thread doesn't do anything else except, perhaps, post a window message to a user interface thread to make something happen that will be visible to the end user.

How to Organize Your Notification Thread
When using either notification method, you can avoid some "plumbing" problems by not having the notification thread block simply on the event or the IOCTL, as the case may be. Instead, proceed as follows: First define a "kill" event that your main application thread will set when it's time for the notification thread to exit.

If you're using the shared event scheme for notification, call *WaitForMultipleObjects* to wait for either the kill event or the event you're sharing with the driver.

If you're using the pending IOCTL scheme, make asynchronous calls to *DeviceIoControl*. Instead of calling *GetOverlappedResult* to block, call *WaitForMultipleObjects* to wait for either the kill event or the event associated with the *OVERLAPPED* structure. If the return code indicates that the *DeviceIoControl* operation has finished, you make a nonblocking call to *GetOverlappedResult* to get the return code and bytes-transferred values. If the return code indicates that the kill event has been signaled, you call *CancelIo* to knock down the *DeviceIoControl*, and you then exit from the thread procedure. Leave out the call to *CancelIo* if your application has to run in Windows 98 Second Edition.

These two methods of solving the notification problem have the relative strengths and weaknesses shown in Table 9-3. Notwithstanding that it appears superficially as though the pending IOCTL method has all sorts of advantages over the shared event method, I recommend you use the shared event method because of the complexity of the race conditions you must handle with the pending IOCTL method.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 282

Sharing an Event	Pending an IRP_MJ_DEVICE_CONTROL
Application needs to create an object by calling CreateEvent.	No object needed.
Driver has to convert handle to object pointer.	No conversion needed.
No cancel logic needed; trivial cleanup.	Cancel and cleanup logic needed; usual horrible race conditions.
Application knows only that "something happened" when event gets signaled.	Driver can provide arbitrary amount of data when it completes the IRP.

9.5.1 Using a Shared Event for Notification

The basic idea behind the event sharing method is that the application creates an event by calling *CreateEvent* and then uses *DeviceIoControl* to send the event handle to the driver:

```
DWORD junk;
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
DeviceIoControl(hdevice, IOCTL_REGISTER_EVENT, &hEvent,
  sizeof(hEvent), NULL, 0, &junk, NULL);
```

NOTE
The EVWAIT sample driver illustrates the shared event method of notifying an application about an interesting event. You generate the "interesting" event by pressing a key on the keyboard, thereby causing the test program to call a back-door IOCTL in the driver. In real life, an actual hardware occurrence would generate the event.

The call to *CreateEvent* creates a kernel-mode *KEVENT* object and makes an entry into the application process's handle table that points to the *KEVENT*. The *HANDLE* value returned to the application is essentially an index into the handle table. The handle isn't directly useful to a WDM driver, though, for two reasons. First of all, there isn't a documented kernel-mode interface for setting an event, given just a handle. Second, and most important, the handle is useful only in a thread that belongs to the same process. If driver code runs in an arbitrary thread (as it often does), it will be unable to reference the event by using the handle.

To get around these two problems with the event handle, the driver has to "convert" the handle to a pointer to the underlying *KEVENT* object. To handle a *METHOD_BUFFERED* control operation that the application uses to register an event with the driver, use code like this:

```
HANDLE hEvent = *(PHANDLE) Irp->AssociatedIrp.SystemBuffer;
PKEVENT pevent;
NTSTATUS status = ObReferenceObjectByHandle(hEvent,
  EVENT_MODIFY_STATE, *ExEventObjectType, Irp->RequestorMode,
  (PVOID*) &pevent, NULL);
```

ObReferenceObjectByHandle looks up *hEvent* in the handle table for the current process and stores the address of the associated kernel object in the *pevent* variable. If the *RequestorMode* in the IRP is *UserMode*, this function also verifies that *hEvent* really is a handle to something, that the something is an event object, and that the handle was opened in a way that includes the *EVENT_MODIFY_STATE* privilege.



Whenever you ask the object manager to resolve a handle obtained from user mode, request access and type checking by indicating *UserMode* for the accessor mode argument to whichever object manager function you're calling. After all, the number you get from user mode might not be a handle at all, or it might be a handle to some other type of object. In addition, avoid the undocumented *ZwSetEvent* function in order not to create the following security hole: even if you've made sure that some random handle is for an event object, your user-mode caller could close the handle and receive back the same numeric handle for a different type of object. You'd then unwittingly cause something bad to happen because you're a *trusted* caller of *ZwSetEvent*.

The application can wait for the event to happen:

```
WaitForSingleObject(hEvent, INFINITE);
```

The driver signals the event in the usual way:

```
KeSetEvent(pevent, EVENT_INCREMENT, FALSE);
```

Eventually, the application cleans up by calling *CloseHandle*. The driver has a separate reference to the event object, which it must release by calling *ObDereferenceObject*. The object manager won't destroy the event object until both these things occur.

IPR2024-00292
KASPERSKY EX 1008 Page 283

9.5.2 Using a Pending IOCTL for Notification

The central idea in the pending IOCTL notification method is that when the application wants to receive event notifications from the driver, it calls *DeviceIoControl*:

```
HANDLE hDevice = CreateFile("\\\\.\\<driver-name>", ...);
BOOL okay = DeviceIoControl(hDevice, IOCTL WAIT NOTIFY,
:);
```

(*IOCTL_WAIT_NOTIFY*, by the way, is the control code I used in the NOTIFY sample in the companion content.)

The driver will pend this IOCTL and complete it later. If other considerations didn't intrude, the code in the driver might be as simple as this:

```
NTSTATUS DispatchControl(...)
  {
  switch (code)
    {
  case IOCTL WAIT NOTIFY:
    IoMarkIrpPending(Irp);
    pdx->NotifyIrp = Irp;
    return STATUS PENDING;
    }
  }

VOID OnInterestingEvent(...)
  {
  CompleteRequest(pdx->NotifyIrp,
    STATUS SUCCESS, 0); // <== don't do this!
  }
```

The "other considerations" I just so conveniently tucked under the rug are, of course, all-important in crafting a working driver. The originator of the IRP might decide to cancel it. The application might call *CancelIo*, or termination of the application thread might cause a kernel-mode component to call *IoCancelIrp*. In either case, we must provide a cancel routine so that the IRP gets completed. If power is removed from our device, or if our device is suddenly removed from the computer, we may want to abort any outstanding IOCTL requests. In general, any number of IOCTLs might need to be aborted. Consequently, we'll need a linked list of them. Since multiple threads might be trying to access this linked list, we'll also need a spin lock so that we can access the list safely.

Helper Routines

To simplify my own life, I wrote a set of helper routines for managing asynchronous IOCTLs. The two most important of these routines are named *CacheControlRequest* and *UncacheControlRequest*. They assume that you're willing to accept only one asynchronous IOCTL having a particular control code per device object and that you can, therefore, reserve a pointer cell in the device extension to point to the IRP that's currently outstanding. In NOTIFY, I call this pointer cell *NotifyIrp*. You accept the asynchronous IRP this way:

```
switch (code)
  {
case IOCTL WAIT NOTIFY:
  if (<parameters invalid in some way>)
    status = STATUS INVALID PARAMETER;
  else
    status = CacheControlRequest(pdx, Irp, &pdx->NotifyIrp);
  break;
  }

return status == STATUS_PENDING ? status : CompleteRequest(Irp, status, info);
```

The important statement here is the call to *CacheControlRequest*, which registers this IRP in such a way that we'll be able to cancel it later if necessary. It also records the address of this IRP in the *NotifyIrp* member of our device extension. We expect it to return *STATUS_PENDING*, in which case we avoid completing the IRP and simply return *STATUS_PENDING* to our caller.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 284

NOTE
You can easily generalize the scheme I'm describing to permit an application to have an IRP of each type outstanding for each open handle. Instead of putting the current IRP pointers in your device extension, put them instead into a structure that you associate with the *FILE_OBJECT* that corresponds to the handle. You'll get a pointer to this *FILE_OBJECT* in the I/O stack location for *IRP_MJ_CREATE*, *IRP_MJ_CLOSE*, and, in fact, all other IRPs generated for the file handle. You can use either the *FsContext* or *FsContext2* field of the file object for any purpose you choose.

Later, when whatever event the application is waiting for occurs, we execute code like this:

```
PIRP nfyirp = UncacheControlRequest(pdx, &pdx->NotifyIrp);if (nfyirp)
  {
  <do something>
  CompleteRequest(nfyirp, STATUS SUCCESS, <info value>);
  }
```

This logic retrieves the address of the pending *IOCTL_WAIT_NOTIFY* request, does something to provide data back to the application, and then completes the pending I/O request packet.

How the Helper Routines Work

I hid a wealth of complications inside the *CacheControlRequest* and *UncacheControlRequest* functions. These two functions provide a thread-safe and multiprocessor-safe mechanism for keeping track of asynchronous IOCTL requests. They use a variation on the techniques we've discussed elsewhere in the book for safely queuing and dequeuing IRPs at times when someone else might be flitting about trying to cancel the IRP. I actually packaged these routines in GENERIC.SYS, and the NOTIFY sample in the companion content shows how to call them. Here's how those functions work (but note that the GENERIC.SYS versions have 'Generic' in their names):

```
typedef struct  DEVICE EXTENSION {
  KSPIN LOCK IoctlListLock;
  LIST ENTRY PendingIoctlList;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

NTSTATUS CacheControlRequest(PDEVICE EXTENSION pdx, PIRP Irp, PIRP* pIrp)
  {
  KIRQL oldirql;
1
  KeAcquireSpinLock(&pdx->IoctlListLock, &oldirql);
  NTSTATUS status;
2
  if (*pIrp)
    status = STATUS UNSUCCESSFUL;
3
  else if (pdx->IoctlAbortStatus)
    status = pdx->IoctlAbortStatus;
  else
    {
4
    IoSetCancelRoutine(Irp, OnCancelPendingIoctl);
    if (Irp->Cancel && IoSetCancelRoutine(Irp, NULL))
      status = STATUS CANCELLED;
    else
      {
5
      IoMarkIrpPending(Irp);
      status = STATUS PENDING;
6
      Irp->Tail.Overlay.DriverContext[0] = pIrp;
      *pIrp = Irp;
      InsertTailList(&pdx->PendingIoctlList, &Irp->Tail.Overlay.ListEntry);
      }
    }
  KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
  return status;
  }

VOID OnCancelPendingIoctl(PDEVICE_OBJECT fdo, PIRP Irp)
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 285

```
    {
    KIRQL oldirql = Irp->CancelIrql;
    IoReleaseCancelSpinLock(DISPATCH_LEVEL);
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    KeAcquireSpinLockAtDpcLevel(&pdx->IoctlListLock);
    RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
    PIRP* pIrp = (PIRP*) Irp->Tail.Overlay.DriverContext[0];
    InterlockedCompareExchange((PVOID*) pIrp, Irp, NULL);
    KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
    Irp->IoStatus.Status = STATUS_CANCELLED;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }

  PIRP UncacheControlRequest(PDEVICE_EXTENSION pdx, PIRP* pIrp)
    {
    KIRQL oldirql;
    KeAcquireSpinLock(&pdx->IoctlListLock, &oldirql);
7   PIRP Irp = (PIRP) InterlockedExchangePointer(pIrp, NULL);
    if (Irp)
      {
8     if (IoSetCancelRoutine(Irp, NULL))
        {
        RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
        }
      else
        Irp = NULL;
      }
    KeReleaseSpinLock(&pdx->IoctlListLock, oldirql);
    return Irp;
    }
```

1. We use a spin lock to guard the list of pending IOCTLs and also to guard all of the pointer cells that are reserved to point to the current instance of each different type of asynchronous IOCTL request.

2. This is where we enforce the rule—it's more of a design decision, really—that only one IRP of each type can be outstanding at one time.

3. This *if* statement accommodates the fact that we may need to start failing incoming IRPs at some point because of PnP or power events.

4. Since we'll pend this IRP for what might be a long time, we should have a cancel routine for it. I've discussed cancel logic so many times in this book that I feel sure you'd rather not read about it once more.

5. Here we've decided to go ahead and cache this IRP so that we can complete it later. Since we're going to end up returning *STATUS_PENDING* from our *DispatchControl* function, we need to call *IoMarkIrpPending*.

6. We need to have a way to *NULL* out the cache pointer cell when we cancel the IRP. Since there's no way to get a context parameter passed to our cancel routine, I decided to co-opt one of the *DriverContext* fields in the IRP to hold a pointer to the cache pointer cell.

7. In the normal course of events, this statement uncaches an IRP.

8. Now that we've uncached our IRP, we don't want it to be cancelled any more. If *IoSetCancelRoutine* returns *NULL*, however, we know that this IRP is currently in the process of being cancelled. We return a *NULL* IRP pointer in that case.

NOTIFY also has an *IRP_MJ_CLEANUP* handler for pending IOCTLs that looks just about the same as the cleanup handlers I've discussed for read and write operations. Finally, it includes an *AbortPendingIoctls* helper function for use at power-down or surprise removal time, as follows:

```
VOID AbortPendingIoctls(PDEVICE_EXTENSION pdx, NTSTATUS status)
  {
  InterlockedExchange(&pdx->IoctlAbortStatus, status);
  CleanupControlRequests(pdx, status, NULL);
  }
```

CleanupControlRequests is the handler for *IRP_MJ_CLEANUP*. I wrote it in such a way that it cancels *all* outstanding IRPs if the third argument—normally a file object pointer—is *NULL*.

NOTIFY is a bit too simple to serve as a complete model for a real-world driver. Here are some additional considerations for

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 286

you to mull over in your own design process:

- A driver might have several types of events that trigger notifications. You could decide to deal with these by using a single IOCTL code, in which case you'd indicate the type of event by some sort of output data, or by using multiple IOCTL codes.
- You might want to allow multiple threads to register for events. If that's the case, you certainly can't have a single IRP pointer in the device extension—you need a way of keeping track of all the IRPs that relate to a particular type of event. If you use only a single type of IOCTL for all notifications, one way to keep track is to queue them on the *PendingIoctlList* shown earlier. Then, when an event occurs, you execute a loop in which you call *ExInterlockedRemoveHeadList* and *IoCompleteRequest* to empty the pending list. (I avoided this complexity in NOTIFY by fiat—I decided I'd run only one instance of the test program at a time.)
- Your IOCTL dispatch routine might be in a race with the activity that generates events. For example, in the USBINT sample I'll discuss in Chapter 12, we have a potential race between the IOCTL dispatch routine and the pseudointerrupt routine that services an interrupt endpoint on a USB device. To avoid losing events or taking inconsistent actions, you need a spin lock. Refer to the USBINT sample in the companion content for an illustration of how to use the spin lock appropriately. (Synchronization wasn't an issue in NOTIFY because by the time a human being is able to perform the keystroke that unleashes the event signal, the notification request is almost certainly pending. If not, the signal request gets an error.)

9.6 Windows 98/Me Compatibility Notes

The VxD service that NTKERN must use to complete an overlapped IOCTL operation (*VWIN32_DIOCCompletionRoutine*) doesn't provide for an error code. Thus, if an application performs an overlapped call to a WDM driver, the eventual call to *GetOverlappedResult* will appear to succeed even if the driver failed the operation.

A Win32 application can use *DeviceIoControl* to communicate with a Windows 98/Me virtual device driver (VxD) as well as a WDM driver. Three subtle and minor differences exist between IOCTLs for WDM drivers and IOCTLs for VxDs. The most important difference has to do with the meaning of the device handle you obtain from *CreateFile*. When you're working with a WDM driver, the handle is for a specific *device*, whereas you get a handle for the *driver* when you're talking to a VxD. In practice, a VxD might need to implement a pseudohandle mechanism (embedded within the IOCTL data flow) to allow applications to refer to specific instances of the hardware managed by the VxD.

Another difference between VxD and WDM control operations concerns the assignment of numeric control codes. As I discussed earlier, you define a control code for a WDM driver by using the *CTL_CODE* macro, and you can't define more than 2048 codes. For a VxD, all 32-bit values except 0 and -1 are available. If you want to write an application that can work with either a VxD or a WDM driver, use *CTL_CODE* to define your control codes because a VxD will be able to work with the resulting numeric values.

The last difference is a pretty minor one: the second-to-last argument to *DeviceIoControl*—a *PDWORD* pointing to a feedback variable—is required when you call a WDM driver but not when you call a VxD. In other words, if you're calling a WDM driver, you must supply a non-*NULL* value pointing to a *DWORD*. If you're calling a VxD, however, you can specify *NULL* if you're not interested in knowing how many data bytes are going into your output buffer. It shouldn't hurt to supply the feedback variable when you call a VxD, though. Furthermore, the fact that this pointer can be *NULL* is something that a VxD writer might easily overlook, and you might provoke a bug if your application takes advantage of the freedom to say *NULL*.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 287

Chapter 10
Windows Management Instrumentation

Microsoft Windows XP supports a facility named Windows Management Instrumentation (WMI) as a way to manage the computer system. WMI is Microsoft's implementation of a broader industry standard called Web-Based Enterprise Management (WBEM). The goal of WMI is to provide a model for system management and the description of management data in an enterprise network that's as independent as possible from a specific API set or data object model. Such independence facilitates the development of general mechanisms for creating, transporting, and displaying data and for exercising control over individual system components.

WDM drivers fit into WMI in three ways. See Figure 10-1. First, WMI responds to requests for data that conveys system management information. Second, controller applications of various kinds can use the facilities of WMI to control generic features of conforming devices. Finally, WMI provides an event-signalling mechanism that allows drivers to notify interested applications of important events. I'll discuss all three of these aspects of driver programming in this chapter.

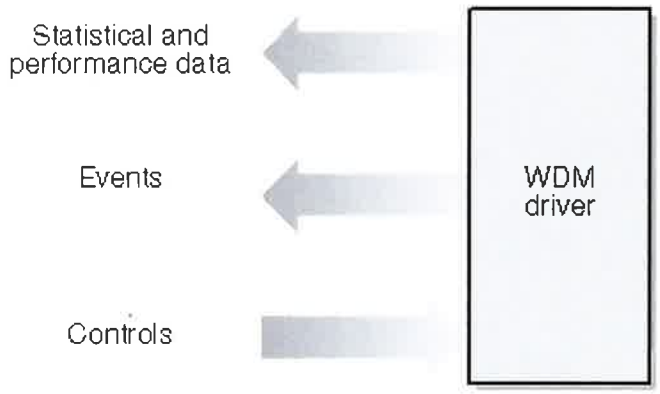


Figure 10-1. *The role of a WDM driver in WMI.*

The WMI and WBEM Names
The Common Information Model (CIM) is a specification for Web-based enterprise management supported by the Distributed Management Task Force (DMTF), formerly named the Desktop Management Task Force. Microsoft named its implementation of the Common Information Model *WBEM*, which was essentially CIM for Windows. The kernel-mode portion of CIM for Windows was called *WMI*. To get CIM more widely adopted, DMTF started a marketing initiative and used WBEM as the name of CIM. Microsoft then renamed its implementation of WBEM to WMI and renamed WMI (the kernel-mode portion) to *WMI extensions for WDM*. That being said, WMI is compliant with the CIM and WBEM specification.

I'm afraid my usage of the various different terms in this chapter won't go very far to resolve the confusion you might feel at this point. I'd suggest that you think *WMI* whenever you see *CIM* or *WBEM* in this book and any documentation Microsoft provides. You'll probably then at least be thinking about the same concept that I and Microsoft are trying to write about—until something with a name like "Windows Basic Extensions for Mortals" or "Completely Integrated Mouse" comes along, that is. Then you're on your own.

10.1 WMI Concepts

Figure 10-2 diagrams the overall architecture of WMI. In the WMI model, the world is divided into *consumers* and *providers* of data and events. Consumers consume, and providers provide, blocks of data that are *instances* of abstract *classes*. The concept involved here is no different from that of a class in the C++ language. Just like C++ classes, WMI classes have data members and *methods* that implement *behaviors* for objects. What goes inside a data block isn't specified by WMI—that depends on who's producing the data and for what purpose.

IPR2024-00292
KASPERSKY EX 1008 Page 289

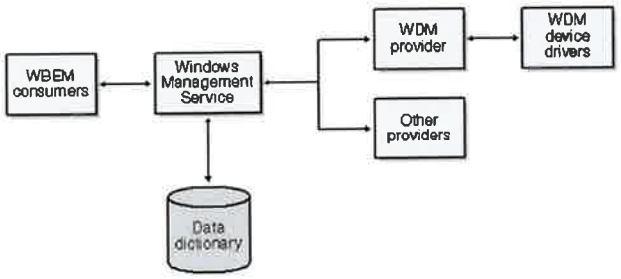


Figure 10-2. *The world of WMI.*

WMI allows for multiple *namespaces*, each of which contains classes belonging to one or more providers. A namespace contains attributes and classes, each of which must have a unique name within that namespace. Each namespace can have its own security descriptor, which an administrator can specify in the Computer Management applet. Consumers and providers alike specify the namespace within which they will operate. Table 10-1 lists some of the namespaces that exist on one of my computers. The WMI namespace is the one we're concerned with in this chapter since it's where the classes are located that drivers work with.

Namespace Name	Description
root\CIMV2	Industry-standard classes
root\DEFAULT	Registry access and monitoring
root\Directory\LDAP	Active Directory objects
root\MSAPPS	Microsoft application classes
root\WMI	WDM device driver classes

Table 10-1. *WMI Namespaces on One of the Author's Computers*

A WDM driver can provide instances of one or more WMI classes in the root\wmi namespace. Many drivers support standard classes defined by Microsoft in the DDK file named WMICORE.MOF. Drivers can also implement a *customschema* that includes vendor-specific or device-specific classes. You define a schema by using a language named the *Managed Object Format*, or MOF. The system maintains a data dictionary known as the *repository* that contains the definitions of all known schemas. Assuming you do all the right things in your driver, the system will automatically put your schema into the repository when it initializes your driver.

More Info
Another way of thinking about WMI is in terms of a classical relational database. A WMI class is the same thing as a table. Instances of a class correspond to records, and the class members to fields in a record. The repository plays the same role as a traditional data dictionary. There is even a query language associated with WMI that includes concepts drawn from Structured Query Language (SQL), which database programmers are familiar with.

10.1.1 A Sample Schema

Later in this chapter, I'll show you a sample named WMI42.SYS, which is available in the companion content. This sample has the following MOF schema:

1
```
[Dynamic, Provider("WMIProv"),
 WMI,
 Description("Wmi42 Sample Schema"),
 guid("A0F95FD4-A587-11d2-BB3A-00C04FA330A6"),
 locale("MS\\0x409")]
```

2
```
class Wmi42
{
    [key, read]
     string InstanceName;

    [read] boolean Active;

    [WmiDataId(1),
     Description("The Answer to the Ultimate Question")
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 290

```
        ]
      uint32 TheAnswer;
};
```

I don't propose to describe all the details of the MOF syntax; that information is available as part of the Platform SDK and WMI SDK documentation. You can either construct your MOF by hand, as I did for this simple example, or use a tool named WBEM CIM Studio that comes with the WMI SDK. Here, however, is a brief explanation of the contents of this MOF file:

1. The provider named *WMIProv* is the system component that knows how to instantiate this class. It understands, for example, how to call into kernel mode and send an I/O request packet (IRP) to an appropriate driver. It can find the right driver by means of the globally unique identifier (GUID) that appears near the beginning of the file.

2. This schema declares a class named *WMI42*, which coincidentally has the same name as our driver. Instances of the class have *properties* named *InstanceName*, *Active*, and *TheAnswer*.

As developers, we run the MOF compiler on this schema definition to produce a binary file that eventually ends up as a resource in our driver executable file. *Resource* in this sense is the same concept that application developers have in mind when they build dialog box templates, string tables, and other things that are part of their project's resource script.

10.1.2 Mapping WMI Classes to C Structures

The *WMI42* class is especially simple because it has just a single data member that happens to be a 32-bit integer. The mapping from the class data structure used by WMI to the C structure used by the driver is therefore completely obvious. The mapping of a more complex WMI class is harder to determine, especially if you declare the data items in a different order than the *WmiDataId* ordinals. Rather than trying to predict how WMI will map a class, I recommend that you ask WMIMOFCK to create a header file containing the declarations you'll need. You can issue commands like these at a command prompt:

```
mofcomp -wmi -b:wmi42.bmf wmi42.mof
wmimofck -hwmi42.h -m wmi42.bmf
```

The *mofcomp* step yields a binary MOF file (wmi42.bmf). Among other things, the *wmimofck* step creates a header file (wmi42.h) like this one:

```
#ifndef  wmi42_h
#define  wmi42_h

// Wmi42 - Wmi42
// Wmi42 Sample Schema
#define Wmi42Guid \
  { 0xa0f95fd4,0xa587,0x11d2, \
  { 0xbb,0x3a,0x00,0xc0,0x4f,0xa3,0x30,0xa6 } }

DEFINE GUID(Wmi42 GUID,
  0xa0f95fd4,0xa587,0x11d2,0xbb,0x3a,
  0x00,0xc0,0x4f,0xa3,0x30,0xa6);


typedef struct  Wmi42
{
  // The Answer to the Ultimate Question of Life,
  // the Universe, and Everything
  ULONG TheAnswer;
  #define Wmi42 TheAnswer SIZE sizeof(ULONG)
  #define Wmi42 TheAnswer ID 1

} Wmi42, *PWmi42;

#define Wmi42 SIZE (FIELD OFFSET(Wmi42, TheAnswer) \
  + Wmi42 TheAnswer SIZE)

#endif
```

Note that the size of the structure is not simply *sizeof(Wmi42)*. The reason for the curious definition of *Wmi42_SIZE* is that WMI class structures aren't padded to a multiple of the most stringent interior alignment as are C structures.

TIP
All of the examples in this chapter rely on the version of WMIMOFCK that is part of beta versions of the Windows .NET Server DDK. If you try to use an earlier DDK to build and test these samples, you'll probably have

IPR2024-00292
KASPERSKY EX 1008 Page 291

to make some source changes to the driver code.

10.2 WDM Drivers and WMI

The kernel-mode support for WMI is based primarily on IRPs with the major code *IRP_MJ_SYSTEM_CONTROL*. You must register your desire to receive these IRPs by making the following call:

```
IoWMIRegistrationControl(fdo, WMI_ACTION_REGISTER);
```

The appropriate time to make the registration call is in the *AddDevice* routine at a point when it would be safe for the system to send the driver a system control IRP. In due course, the system will send you an *IRP_MJ_SYSTEM_CONTROL* request to obtain detailed registration information about your device. You'll balance the registration call with another call at *RemoveDevice* time:

```
IoWMIRegistrationControl(fdo, WMI_ACTION_DEREGISTER);
```

If any WMI requests are outstanding at the time you make the deregistration call, *IoWMIRegistrationControl* waits until they complete. It's therefore necessary to make sure that your driver is still capable of responding to IRPs when you deregister. You can have new IRPs fail with *STATUS_DELETE_PENDING*, but you have to respond.

Before explaining how to service the registration request, I'll describe how you handle system control IRPs in general. An *IRP_MJ_SYSTEM_CONTROL* request can have any of the minor function codes listed in Table 10-2.

Minor Function Code	Description
IRP_MN_QUERY_ALL_DATA	Gets all instances of every item in a data block
IRP_MN_QUERY_SINGLE_INSTANCE	Gets every item in a single instance of a data block
IRP_MN_CHANGE_SINGLE_INSTANCE	Replaces every item in a single instance of a data block
IRP_MN_CHANGE_SINGLE_ITEM	Changes one item in a data block
IRP_MN_ENABLE_EVENTS	Enables event generation
IRP_MN_DISABLE_EVENTS	Disables event generation
IRP_MN_ENABLE_COLLECTION	Starts collecting "expensive" statistics
IRP_MN_DISABLE_COLLECTION	Stops collecting "expensive" statistics
IRP_MN_REGINFO_EX	Gets detailed registration information
IRP_MN_EXECUTE_METHOD	Executes a method function

Table 10-2. *Minor Function Codes for IRP_MJ_SYSTEM_CONTROL*

The *Parameters* union in the stack location includes a *WMI* substructure with parameters for the system control request:

```
struct {
  ULONG PTR ProviderId;
  PVOID DataPath;
  ULONG BufferSize;
  PVOID Buffer;
  } WMI;
```

ProviderId is a pointer to the device object to which the request is directed. *Buffer* is the address of an input/output area where the first several bytes are mapped by the *WNODE_HEADER* structure. *BufferSize* gives the size of the buffer area. Your dispatch function will extract some information from this buffer and will also return results in the same memory area. For all the minor functions except *IRP_MN_REGINFO*, *DataPath* is the address of a 128-bit GUID that identifies a class of data block. The *DataPath* field is either *WMIREGISTER* or *WMIUPDATE* (0 or 1, respectively) for an *IRP_MN_REGINFO* request, depending on whether you're being told to provide initial registration information or just to update the information you supplied earlier.

When you design your driver, you must choose between two ways of handling system control IRPs. One method is relying on the facilities of the WMILIB support "driver." WMILIB is really a kernel-mode DLL that exports services you can call from your driver to handle some of the annoying mechanics of IRP processing. The other method is simply handling the IRPs yourself. If you use WMILIB, you'll end up writing less code but you won't be able to use every last feature of WMI to its fullest—you'll be limited to the subset supported by WMILIB. Furthermore, your driver won't run under the original retail release of Microsoft Windows 98 because WMILIB wasn't available then. *Before you let the lack of WMILIB in original Windows 98 ruin your day, consult the compatibility notes at the end of this chapter.*

WMILIB suffices for most drivers, so I'm going to limit my discussion to using WMILIB. The DDK documentation describes how to handle system control IRPs yourself if you absolutely have to.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 292

10.2.1 Delegating IRPs to WMILIB

In your dispatch routine for system control IRPs, you delegate most of the work to WMILIB with code like the following:

```
WMIGUIDREGINFO guidlist[] = {
  {&Wmi42 GUID, 1, 0},
  };

1
WMILIB CONTEXT libinfo = {
  arraysize(guidlist),
  guidlist,
  QueryRegInfo,
  QueryDataBlock,
  SetDataBlock,
  SetDataItem,
  ExecuteMethod,
  FunctionControl,
  };

NTSTATUS DispatchWmi(IN PDEVICE OBJECT fdo, IN PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status;
  SYSCTL IRP DISPOSITION disposition;

2
  status = WmiSystemControl(&libinfo, fdo, Irp, &disposition);

  switch (disposition)
    {

  case IrpProcessed:

3
    break;

  case IrpNotCompleted:

4
    IoCompleteRequest(Irp, IO NO INCREMENT);
    break;

5
  default:
  case IrpNotWmi:
  case IrpForward:

6
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    break;
    }

  return status;
  }
```

1. The *WMILIB_CONTEXT* structure declared at file scope describes the class GUIDs your driver supports and lists several callback functions that WMILIB uses to handle WMI requests in the appropriate device-dependent and driver-dependent way. It's OK to use a static context structure if the information in it doesn't change from one IRP to the next.

2. This statement calls WMILIB to handle the IRP. We pass the address of our *WMILIB_CONTEXT* structure. *WmiSystemControl* returns two pieces of information: an *NTSTATUS* code and a *SYSCTL_IRP_DISPOSITION* value.

3. Depending on the disposition code, we might have additional work to perform on this IRP. If the code is *IrpProcessed*, the IRP has already been completed and we need do nothing more with it. This case would be the normal one for minor functions other than *IRP_MN_REGINFO*.

4. If the disposition code is *IrpNotCompleted*, completing the IRP is our responsibility. This case would be the normal one for *IRP_MN_REGINFO*. WMILIB has already filled in the *IoStatus* block of the IRP, so we need only call *IoCompleteRequest*.

5. The *default* and *IrpNotWmi* cases shouldn't arise in Windows XP. We'd get to the default label if we weren't handling all possible disposition codes; we'd get to the *IrpNotWmi* case label if we sent an IRP to WMILIB that didn't have one of the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 293

minor function codes that specifies WMI functionality.

6. The *IrpForward* case occurs for system control IRPs that are intended for some other driver. Recall that the *ProviderId* parameter indicates the driver that is supposed to handle this IRP. *WmiSystemControl* compares that value with the device object pointer we supply as the second function argument. If they're not the same, it returns *IrpForward* so that we'll send the IRP down the stack to the next driver.

The way a WMI consumer matches up to your driver in your driver's role as a WMI provider is based on the GUID or GUIDs you supply in the context structure. When a consumer wants to retrieve data, it (indirectly) accesses the data dictionary in the WMI repository to translate a symbolic object name into a GUID. The GUID is part of the MOF syntax I showed you earlier. You specify the same GUID in your context structure, and WMILIB takes care of the matching.

WMILIB will call routines in your driver to perform device-dependent or driver-dependent processing. Most of the time, the callback routines will perform the requested operation synchronously. However, except in the case of *IRP_MN_REGINFO*, you can defer processing by returning *STATUS_PENDING* and completing the request later.

The *QueryRegInfo* Callback

The first system control IRP we'll receive after making our registration call has the minor function code *IRP_MN_REGINFO*. When we pass this IRP to *WmiSystemControl*, it turns around and calls the *QueryRegInfo* function—it finds the function's address in our *WMILIB_CONTEXT* structure. Here's how WMI42.SYS handles this callback:

```
NTSTATUS QueryRegInfo(PDEVICE OBJECT fdo, PULONG flags,
  PUNICODE STRING instname, PUNICODE STRING* regpath,
  PUNICODE STRING resname, PDEVICE OBJECT* pdo)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  *flags = WMIREG FLAG INSTANCE PDO;
  *regpath = &servkey;
  RtlInitUnicodeString(resname, L"MofResource");
  *pdo = pdx->Pdo;
  return STATUS SUCCESS;
  }
```

We set *regpath* to the address of a *UNICODE_STRING* structure that contains the name of the service registry key describing our driver. This key is the one below ...\System\CurrentControlSet\Services. Our *DriverEntry* routine received the name of this key as an argument and saved it in the global variable *servkey*. We set *resname* to the name we chose to give our schema in our resource script. Here's the resource file for WMI42.SYS so that you can see where this name comes from:

```
#include <windows.h>

LANGUAGE LANG ENGLISH, SUBLANG NEUTRAL
MofResource MOFDATA wmi42.bmf
```

WMI42.BMF is where our build script puts the compiled MOF file. You can name this resource anything you want to, but *MofResource* is traditional (in a tradition stretching back to, uh, last Tuesday). All that matters about the name is that you specify the *same* name when you service the *QueryRegInfo* call.

How we set the remaining values depends on how our driver wants to handle instance naming. I'll come back to the subject of instance naming later in this chapter (in "Instance Naming"). The simplest choice, and the one Microsoft strongly recommends, is the one I adopted in WMI42.SYS: have the system automatically generate names that are *static* based on the name the bus driver gave to the physical device object (PDO). When we make this choice of naming method, we do the following tasks in *QueryRegInfo*:

- Set the *WMIREG_FLAG_INSTANCE_PDO* flag in the *flags* value we're returning to WMILIB. Setting the flag here tells WMILIB that at least one of our objects uses PDO naming.
- Set the *pdo* value we're returning to WMILIB. In my sample drivers, my device extension has a field named *Pdo* that I set at *AddDevice* time to make it available at times like this.

Apart from making your life easier, basing your instance names on the PDO allows viewer applications to automatically determine your device's friendly name and other properties without you doing anything more in your driver.

When you return a successful status from *QueryRegInfo*, WMILIB goes on to create a complicated structure called a *WMIREGINFO* that includes your GUID list, your registry key, your resource name, and information about your instance names. It returns to your dispatch function, which then completes the IRP and returns. Figure 10-3 diagrams this process.

IPR2024-00292
KASPERSKY EX 1008 Page 294

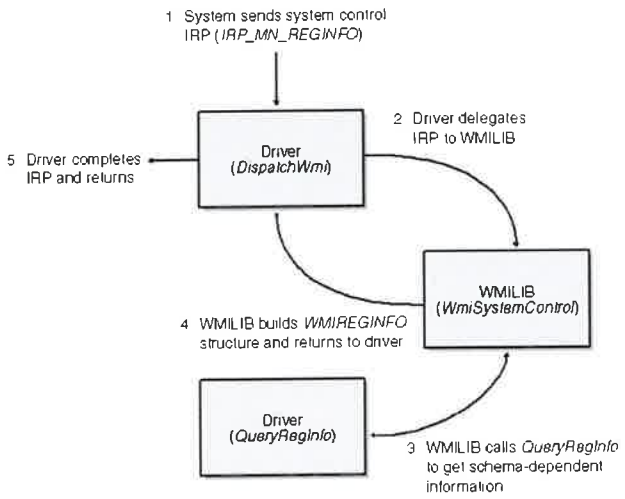


Figure 10-3. *Control flow for IRP_MN_REGINFO.*

The *QueryDataBlock* Callback

The information you provide in your answer to the initial registration query allows the system to route relevant data operations to you. User-mode code can use various COM interfaces to get and set data values at several levels of aggregation. Table 10 - 3 summarizes the four possibilities.

IRP Minor Function	WMILIB Callback	Description
IRP_MN_QUERY_ALL_DATA	QueryDataBlock	Gets all items of all instances
IRP_MN_QUERY_SINGLE_INSTANCE	QueryDataBlock	Gets all items of one instance
IRP_MN_CHANGE_SINGLE_INSTANCE	SetDataBlock	Sets all items of one instance
IRP_MN_CHANGE_SINGLE_ITEM	SetDataItem	Sets one item in one instance

Table 10-3. *Forms of Data Queries*

When someone wants to learn the value or values of the data you're keeping, he or she sends you a system control IRP with one of the minor function codes *IRP_MN_QUERY_ALL_DATA* or *IRP_MN_QUERY_SINGLE_INSTANCE*. If you're using WMILIB, you'll delegate the IRP to *WmiSystemControl*, which will then call your *QueryDataBlock* callback routine. You'll provide the requested data, call another WMILIB routine named *WmiCompleteRequest* to complete the IRP, and then return to WMILIB to unwind the process. In this situation, *WmiSystemControl* will return the *IrpProcessed* disposition code because you've already completed the IRP. Refer to Figure 10-4 for a diagram of the overall control flow.

Your *QueryDataBlock* callback can end up being a relatively complex function if your driver is maintaining multiple instances of a data block that varies in size from one instance to the next. I'll discuss the complications later in "Dealing with Multiple Instances." The WMI42 sample shows how to handle a simpler case in which your driver maintains only one instance of the WMI class:

```
NTSTATUS QueryDataBlock(PDEVICE_OBJECT fdo, PIRP Irp,
  ULONG guidindex, ULONG instindex, ULONG instcount,
  PULONG instlength, ULONG bufsize, PUCHAR buffer)
  {
1
  if (quidindex > arraysize(guidlist))
    return WmiCompleteRequest(fdo, Irp,
      STATUS_WMI_GUID_NOT_FOUND, 0, IO_NO_INCREMENT);
  if (instindex != 0 || instcount != 1)
    return WmiCompleteRequest(fdo, Irp,
      STATUS_WMI_INSTANCE_NOT_FOUND, 0, IO_NO_INCREMENT);
2
  if (!instlength || bufsize < Wmi42_SIZE)
    return WmiCompleteRequest(fdo, Irp, STATUS_BUFFER_TOO_SMALL,
      Wmi42_SIZE, IO_NO_INCREMENT);

  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
```


IPR2024-00292
KASPERSKY EX 1008 Page 295

```
3
    PWmi42 pvalue = (PWmi42) buffer;
    pvalue->TheAnswer = pdx->TheAnswer;
    instlength[0] = Wmi42 SIZE;

4
    return WmiCompleteRequest(fdo, Irp, STATUS SUCCESS,
      Wmi42 SIZE, IO NO INCREMENT);
    }
```

1. The WMI subsystem should already have verified that we're being queried for an instance of a class that we actually support. Thus, *guidindex* should ordinarily be within the bounds of the GUID list, and *instindex* and *instcount* ought not to exceed the number of instances we've said we own. If, however, we've just changed our registration information, we might be servicing a request that was already "in flight," and these tests would be needed to avoid mistakes.

2. We're obliged to make this check to verify that the buffer area is large enough to accommodate the data and data length values we're going to put there. The first part of the test—is there an *instlength* array?—is boilerplate. The second part of the test—is the buffer big enough for a *Wmi42* structure?—is where we verify that all of our data values will fit.

3. The *buffer* parameter points to a memory area where we can put our data. The *instlength* parameter points to an array where we're supposed to place the length of each data instance we're returning. Here we install the single data value our schema calls for—the value of the *TheAnswer* property—and its length. Figuring out what *TheAnswer* actually is numerically is left as an exercise for the reader.

4. The WMILIB specification requires us to complete the IRP by calling the *WmiCompleteRequest* helper routine. The fourth argument indicates how much of the buffer area we used for data values. By now, the other arguments should be self-explanatory.

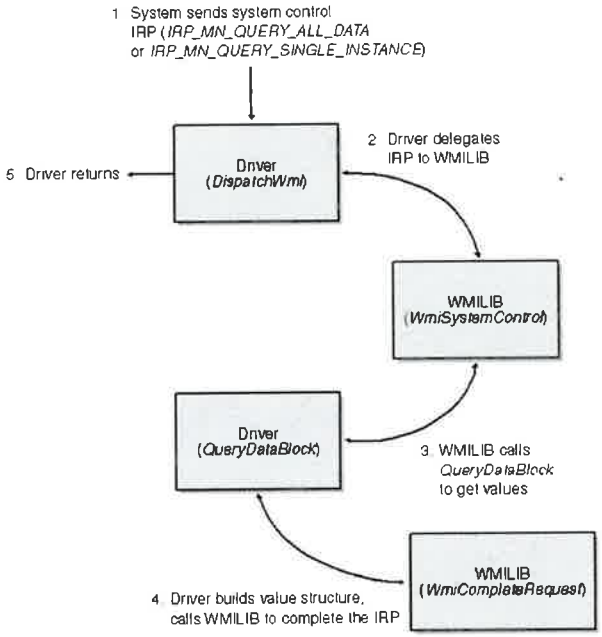


Figure 10-4. *Control flow for data queries.*

The *SetDataBlock* Callback

The system might ask you to change an entire instance of one of your classes by sending you an *IRP_MN_CHANGE_SINGLE_INSTANCE* request. *WmiSystemControl* processes this IRP by calling your *SetDataBlock* callback routine. A simple version of this routine might look like this:

```
NTSTATUS SetDataBlock(PDEVICE OBJECT fdo, PIRP Irp,
   ULONG guidindex, ULONG instindex, ULONG bufsize,
   PUCHAR buffer)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
```


IPR2024-00292
KASPERSKY EX 1008 Page 296

```
  if (quidindex > arraysize(guidlist))
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI GUID NOT FOUND, 0, IO NO INCREMENT);
  if (instindex != 0)
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI INSTANCE NOT FOUND, 0, IO NO INCREMENT);
2
  if (bufsize == Wmi42 SIZE)
    {
    pdx->TheAnswer = ((PWmi42) buffer)->TheAnswer;
    status = STATUS SUCCESS;
    info = Wmi42 SIZE;
    }
  else
    status = STATUS INFO LENGTH MISMATCH, info = 0;
3
  return WmiCompleteRequest(fdo, Irp, status, info, IO NO INCREMENT);
  }
```

1. These are the same sanity checks I showed earlier for the *QueryDataBlock* callback function.
2. The system should already know—based on the MOF declaration—how big an instance of each class is and should give us a buffer that's exactly the right size. If it doesn't, we'll end up causing this IRP to fail. Otherwise, we'll copy a new value for the data block into the place where we keep our copy of that value.
3. We're responsible for completing the IRP by calling *WmiCompleteRequest*.

The *SetDataItem* Callback

Sometimes consumers want to change just one field in one of the WMI objects we support. Each field has an identifying number that appears in the *WmiDataId* property of the field's MOF declaration. (The *Active* and *InstanceName* properties aren't changeable and don't have identifiers. Furthermore, they're implemented by the system and don't even appear in the data blocks we work with.) To change the value of one field, the consumer references the field's ID. We then receive an *IRP_MN_CHANGE_SINGLE_ITEM* request, which *WmiSystemControl* processes by calling our *SetDataItem* callback routine:

```
NTSTATUS SetDataItem(PDEVICE OBJECT fdo, PIRP Irp, ULONG guidindex,
  ULONG instindex, ULONG id, ULONG bufsize, PUCHAR buffer)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status;
  ULONG info;

  if (quidindex > arraysize(guidlist))
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI GUID NOT FOUND, 0, IO NO INCREMENT);
  if (instindex != 0 || instcount != 1)
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI INSTANCE NOT FOUND, 0, IO NO INCREMENT);
  if (id != Wmi42_TheAnswer_ID)
    return WmiCompleteRequest(fdo, Irp,
      STATUS_WMI_ITEMID_NOT_FOUND, 0, IO_NO_INCREMENT);

  if (bufsize == Wmi42 SIZE)
    {
    pdx->TheAnswer = (PWmi42) buffer)->TheAnswer;
    status = STATUS SUCCESS;
    info = Wmi42 SIZE;
    }
  else
    status = STATUS INFO LENGTH MISMATCH, info = 0;

  return WmiCompleteRequest(fdo, Irp, status, info, IO NO INCREMENT);
  }
```

In this example, the only difference between the *SetDataItem* and *SetDataBlock* callbacks is the additional test of the field identifier, shown in boldface.

You can use the "-c" option when you run WMIMOFCK. This option generates a C-language source file with a number of TODO items that you can complete in order to end up with pretty much all the code you need. I don't use this feature myself, because WDMWIZ generates skeleton code that fits better into my own framework and actually requires fewer changes. I do,

IPR2024-00292
KASPERSKY EX 1008 Page 297

however, use the "-h" option of WMIMOFCK as described earlier in this chapter, because there's no good alternative for getting the correct structure mapping.

10.2.2 Advanced Features

The preceding discussion covers much of what you need to know to provide meaningful performance information for metering applications. Use your imagination here: instead of providing just a single statistic (*TheAnswer*), you can accumulate and return any number of performance measures that are relevant to your specific device. You can support some additional WMI features for more specialized purposes. I'll discuss these features now.

Dealing with Multiple Instances

WMI allows you to create multiple instances of a particular class data block for a single device object. You might want to provide multiple instances if your device is a controller or some other device in to which other devices plug; each instance might represent data about one of the child devices. Mechanically, you specify the number of instances of a class in the *WMIGUIDREGINFO* structure for the GUID associated with the class. If WMI42 had three different instances of its standard data block, for example, it would have used the following GUID list in its *WMILIB_CONTEXT* structure:

```
WMIGUIDREGINFO guidlist[] = {
  {&Wmi42 GUID, 3, 0},
  };
```

The only difference between this GUID list and the one I showed you earlier is that the instance count here is 3 instead of 1. This list declares that there will be three instances of the WMI42 data block, each with its own value for the three properties (that is, *InstanceName*, *Active*, and *TheAnswer*) that belong in that block.

If the number of instances changes over time, you can call *IoWmiRegistrationControl* with the action code *WMIREG_ACTION_UPDATE_GUID* to cause the system to send you another registration request, which you'll process using an updated copy of your *WMILIB_CONTEXT* structure. If you're going to be changing your registration information, you should probably allocate the *WMILIB_CONTEXT* structure and GUID list from the free pool rather than use static variables, by the way.

If user-mode code were to enumerate all instances of *GUID_WMI42_SCHEMA*, it would find three instances. This result might present a confusing picture to user-mode code, though. Depending on which WDM platform you're running on, it may be difficult to tell a priori that the three instances disclosed by the enumeration belong to a single device, as opposed to a situation in which three WMI42 devices each expose a single instance of the same class. To allow WMI clients to sort out the difference between the two situations, your schema should include a property (a device name or the like) that can function as a key.

Once you allow for the possibility of multiple instances, several of your WMILIB callbacks will require changes from the simple examples I showed you earlier. In particular:

- *QueryDataBlock* should be able to return the data block for a single instance or for any number of instances beginning at a specific index.
- *SetDataBlock* should interpret its instance number argument to decide which instance to change.
- *SetDataItem* should likewise interpret its instance number argument to locate the instance within which the affected data item will be found.

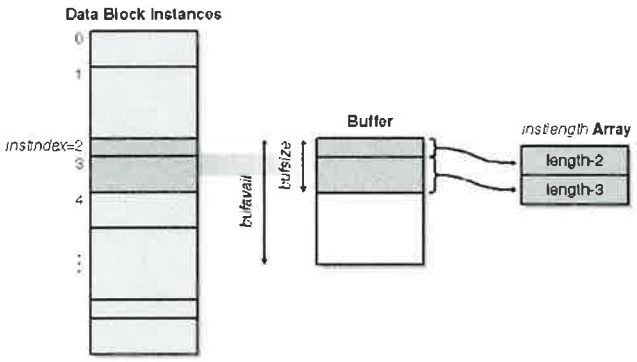


Figure 10-5. *Getting multiple data block instances.*

Figure 10-5 illustrates how your *QueryDataBlock* function uses the output buffer when it's asked to provide more than one instance of a data block. Imagine that you were asked to provide data for two instances beginning at instance number 2. You'll copy the data values, which I've shown as being of different sizes, into the data buffer. You start each instance on an 8-byte

IPR2024-00292
KASPERSKY EX 1008 Page 298

boundary. You indicate the total number of bytes you consume when you complete the query, and you indicate the lengths of each individual instance by filling in the *instlength* array, as shown in the figure.

Instance Naming

Each instance of a WMI class has a unique name. Consumers who know the name of an instance can perform queries and invoke method routines. Consumers who don't know the name or names of the instance or instances you provide can learn them by enumerating the class. In any case, you're responsible for generating the names that consumers use or discover.

I showed you the simplest way—from the driver's perspective, that is—of naming instances of a custom data block, which is to request that WMI automatically generate a static, unique name based on the name of the PDO for your device. If your PDO has the name Root\SAMPLE\0000, for example, a PDO-based name for a single instance of a given data block will be Root\SAMPLE\0000_0.

Basing instance names on the PDO name is obviously convenient because all you need to do in the driver is set the *WMIREG_FLAG_INSTANCE_PDO* flag in the *flags* variable that WMILIB passes to your *QueryRegInfo* callback routine. The author of a consumer application can't know what this name will be, however, because the name will vary depending on how your device was installed. To make the instance names slightly more predictable, you can elect to use a constant *base name* for object instances instead. You indicate this choice by responding in the following way to the registration query:

```
NTSTATUS QueryRegInfo(PDEVICE_OBJECT fdo, PULONG flags,
  PUNICODE_STRING instname, PUNICODE_STRING* regpath,
  PUNICODE_STRING resname, PDEVICE_OBJECT* pdo)
  {
  *flags = WMIREG_FLAG_INSTANCE_BASENAME;
  *regpath = &servkey;
  RtlInitUnicodeString(resname, L"MofResource");
  static WCHAR basename[] = L"WMIEXTRA";
  instname->Buffer = (PWCHAR) ExAllocatePool(PagedPool, sizeof(basename));
  if (!instname->Buffer)
    return STATUS_INSUFFICIENT_RESOURCES;
  instname->MaximumLength = sizeof(basename);
  instname->Length = sizeof(basename) - 2;
  RtlCopyMemory(instname->Buffer, basename, sizeof(basename));
  }
```

The parts of this function that differ from the previous example of *QueryRegInfo* are in boldface. In the WMIEXTRA sample, only one instance of each data block exists, and each receives the instance name *WMIEXTRA* with no additional decoration.

If you elect to use a base name, try to avoid generic names such as Toys because of the confusion that can ensue. The purpose of this feature is to let you use specific names such as TailspinToys.

In some circumstances, static instance names won't suit your needs. If you maintain a population of data blocks that changes frequently, using static names means that you have to request a registration update each time the population changes. The update is relatively expensive, and you should avoid requesting one often. You can assign *dynamic* instance names to the instances of your data blocks instead of static names. The instance names then become part of the queries and replies that you deal with in your driver. Unfortunately, WMILIB doesn't support the use of dynamic instance names. To use this feature, therefore, you'll have to fully implement support for the *IRP_MJ_SYSTEM_CONTROL* requests that WMILIB would otherwise interpret for you. Describing how to handle these IRPs yourself is beyond the scope of this book, but the DDK documentation contains detailed information about how to go about it.

Dealing with Multiple Classes

WMI42 deals with only one class of data block. If you want to support more than one class, you need to have a bigger array of GUID information structures, as WMIEXTRA does:

```
WMIGUIDREGINFO guidlist[] = {
  {&wmiextra_event_GUID, 1, WMIREG_FLAG_EVENT_ONLY_GUID},
  {&wmiextra_expensive_GUID, 1, WMIREG_FLAG_EXPENSIVE },
  {&wmiextra_method_GUID, 1, 0},
  };
```

Before calling one of your callback routines, WMILIB looks up the GUID accompanying the IRP in your list. If the GUID isn't in the list, WMILIB causes the IRP to fail. If it's in the list, WMILIB calls your callback routine with the *guidindex* parameter set equal to the index of the GUID in your list. By inspecting this parameter, you can tell which data block you're being asked to work with.

You can use the special flag *WMIREG_FLAG_REMOVE_GUID* in a GUID information structure. The purpose of this flag is to remove a particular GUID from the list of supported GUIDs during a registration update. Using this flag also prevents

IPR2024-00292
KASPERSKY EX 1008 Page 299

WMILIB from calling you to perform an operation on a GUID that you're trying to remove.

Expensive Statistics

It can sometimes be burdensome to collect all of the statistics that are potentially useful to an end user or administrator. For example, it would be possible for a disk driver (or, more likely, a filter driver sitting in the same stack as a disk driver) to collect histogram data showing how often I/O requests reference a particular sector of the disk. This data would be useful to a disk-defragmenting program because it would allow the most frequently accessed sectors to be placed in the middle of a disk for optimal seek time. You wouldn't want to routinely collect this data, though, because of the amount of memory needed for the collection. That memory would have to be nonpaged too because of the possibility that a particular I/O request would be for page swapping.

WMI allows you to declare a particular data block as being *expensive* so that you don't need to collect it except on demand, as shown in this excerpt from the WMIEXTRA sample program:

```
WMIGUIDREGINFO guidlist[] = {
:
  {&wmiextra expensive GUID, 1,
    WMIREG FLAG EXPENSIVE},
:
};
```

The *WMIREG_FLAG_EXPENSIVE* flag indicates that the data block identified by *wmiextra_expensive_GUID* has this expensive characteristic.

When an application expresses interest in retrieving values from an expensive data block, WMI sends you a system control IRP with the minor function code *IRP_MN_ENABLE_COLLECTION*. When no applications are interested in an expensive data block anymore, WMI sends you another IRP with the minor function code *IRP_MN_DISABLE_COLLECTION*. If you delegate these IRPs to WMILIB, it will turn around and call your *FunctionControl* callback routine to either enable or disable collection of the values in the data block:

```
NTSTATUS FunctionControl(PDEVICE OBJECT fdo, PIRP Irp,
  ULONG guidindex, WMIENABLEDISABLECONTROL fcn, BOOLEAN enable)
  {
:
  return WmiCompleteRequest(fdo, Irp, STATUS SUCCESS, 0, IO NO INCREMENT);
  }
```

In these arguments, *guidindex* is the index of the GUID for the expensive data block in your list of GUIDs, *fcn* will equal the enumeration value *WmiDataBlockControl* to indicate that collection of an expensive statistic is being either enabled or disabled, and *enable* will be *TRUE* or *FALSE* to indicate whether you should or should not collect the statistic, respectively. As shown in this fragment, you call *WmiCompleteRequest* prior to returning from this function.

An application "expresses interest" in a data block, by the way, by retrieving an *IWbemClassObject* interface pointer bound to a particular instance of your data block's WMI class. Notwithstanding the fact that an application has to discover an instance of the class, no instance index appears in the call to your *FunctionControl* callback. The instruction to collect or not collect the expensive statistic therefore applies to all instances of your class.

WMI Events

WMI provides a way for providers to notify consumers of interesting or alarming events. A device driver might use this facility to alert a user to some facet of device operation that requires user intervention. For example, a disk driver might notice that an unusually large number of bad sectors have accumulated on a disk. Logging such an event as described in Chapter 14 is one way to inform the human world of this fact, but an administrator has to actively look at the event log to see the entry. If someone were to write an event monitoring applet, however, and if you were to fire a WMI event when you noticed the degradation, the event could be brought immediately to the user's attention.

NOTE
The network connection tray icon responds to a kernel-mode driver (namely NDIS.SYS) signalling a WMI event.

WMI events are just regular WMI classes used in a special way. In MOF syntax, you must derive the data block from the abstract *WMIEvent* class, as illustrated in this excerpt from WMIEXTRA's MOF file:

```
[Dynamic, Provider("WMIProv"),
 WMI,
 Description("Event Info from WMIExtra"),
 guid("c4b678f6-b6e9-11d2-bb87-00c04fa330a6"),
 locale("MS\\0x409")]
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 300

```
class wmiextra event : WMIEvent
{
    [key, read]
     string InstanceName;

    [read] boolean Active;

    [WmiDataId(1), read] uint32 EventInfo;

};
```

Although events can be normal data blocks, you might not want to allow applications to read and write them separately. If not, use the *EVENT_ONLY* flag in your declaration of the GUID:

```
WMIGUIDREGINFO guidlist[] = {
  {&wmiextra event GUID, 1,
    WMIREG FLAG EVENT ONLY GUID},
  };
```

When an application expresses interest in knowing about a particular event, WMI sends your driver a system control IRP with the minor function code *IRP_MN_ENABLE_EVENTS*. When no application is interested in an event anymore, WMI sends you another IRP with the minor function code *IRP_MN_DISABLE_EVENTS*. If you delegate these IRPs to WMILIB, you'll receive a call in your *FunctionControl* callback to specify the GUID index in your list of GUIDs, the *fcn* code *WmiEventControl*, and a Boolean *enable* flag.

To fire an event, construct an instance of the event class in nonpaged memory and call *WmiFireEvent*. For example:

```
Pwmiextra event junk = (Pwmiextra event)
  ExAllocatePool(NonPagedPool, wmiextra event SIZE);
junk->EventInfo = 42;
WmiFireEvent(fdo, (LPGUID) &wmiextra event GUID, 0,
  sizeof(wmiextra_event), junk);
```

The WMI subsystem will release the memory that's occupied by the event object in due course.

WMI Method Routines

In addition to defining mechanisms for transferring data and signalling events, WMI prescribes a way for consumers to invoke *method routines* implemented by providers. WMIEXTRA defines the following class, which includes a method routine:

```
[Dynamic, Provider("WMIProv"),
 WMI,
 Description("WMIExtra class with method"),
 guid("cd7ec27d-b6e9-11d2-bb87-00c04fa330a6"),
 locale("MS\\0x409")]

class wmiextra method
{
    [key, read]
     string InstanceName;

    [read] boolean Active;

    [Implemented, WmiMethodId(1)] void
     AnswerMethod([in,out] uint32 TheAnswer);

};
```

This declaration indicates that *AnswerMethod* accepts an input/output argument named *TheAnswer* (a 32-bit unsigned integer). Note that all method functions exposed by WDM drivers must be *void* functions because there's no way to indicate a return value. You can still have output arguments or input/output arguments.

When you delegate system control IRPs to WMILIB, a method routine call manifests itself in a call to your *ExecuteMethod* callback routine:

IPR2024-00292
KASPERSKY EX 1008 Page 301

```
NTSTATUS ExecuteMethod(PDEVICE OBJECT fdo, PIRP Irp,
  ULONG guidindex, ULONG instindex, ULONG id,
  ULONG cbInbuf, ULONG cbOutbuf, PUCHAR buffer)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  NTSTATUS status = STATUS SUCCESS;
  ULONG bufused = 0;
  if (guidindex != INDEX WMIEXTRA METHOD)
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI GUID NOT FOUND, 0, IO NO INCREMENT);
  if (instindex != 0)
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI INSTANCE NOT FOUND, 0, IO NO INCREMENT);
  if (id != AnswerMethod)
    return WmiCompleteRequest(fdo, Irp,
      STATUS WMI ITEMID NOT FOUND, 0, IO NO INCREMENT);
  if (cbInbuf < AnswerMethod IN SIZE)
    status = STATUS INVALID PARAMETER;
  else if (cbOutbuf < AnswerMethod OUT SIZE)
    status = STATUS BUFFER TOO SMALL;
  else
    {
    PAnswerMethod IN in = (PAnswerMethod IN) buffer;
    PAnswerMethod OUT out = (PAnswerMethod OUT) buffer;
    out->TheAnswer = in->TheAnswer + 1;
    bufused = AnswerMethod OUT SIZE;
    }
  return WmiCompleteRequest(fdo, Irp, status, bufused, IO NO INCREMENT);
  }
```

WMI method calls use an input class to contain the input arguments and a potentially different output class to contain the returned values. The *buffer* area contains an image of the input class, whose length is *cbInbuf*. Your job is to perform the method and overstore the buffer area with an image of the output class. You complete the request with the byte size (*bufused*) of the output class.

This particular method routine simply adds 1 to its input argument.

Simply enumerating an instance of a class such as *wmiextra_method* triggers a request for the data block. You must cause the data query to succeed even if the class that contains the method routine has no data members. In such a case, you can just complete the query with a 0 data length.



Be very circumspect in the functionality you expose through WMI methods. This advice is exceptionally important because scripts coming from unverified sources can run with the privileges of any logged-in user and can therefore make calls to your methods.

Standard Data Blocks

Microsoft has defined some standardized data blocks for various types of devices. If your device belongs to a class for which standardized data blocks are defined, you should support those blocks in your driver. Consult WMICORE.MOF in the DDK to see the class definitions, and see Table 10-4.

To implement your support for a standard data block, include the corresponding GUID in the list you report back from the registration query. Implement supporting code for getting and putting data, enabling and disabling events, and so on, using the techniques I've already discussed. Don't include definitions of the standard data blocks in your own schema; those class definitions are already in the repository, and you don't want to override them.

You can include the DDK header file WMIDATA.H in your driver to get GUID definitions and class structure layouts.

In many cases, by the way, a Microsoft class driver will be providing the actual WMI support for these standard classes—you might not have any work to do.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 302

Device Type	Standard Class	Description
Keyboard	MSKeyboard_PortInformation	Configuration and performance information
	MSKeyboard_ExtendedID	Extended type and subtype identifiers
	MSKeyboard_ClassInformation	Device identification number
Mouse	MSMouse_PortInformation	Configuration and performance information
	MSMouse_ClassInformation	Device identification number
Disk	MSDiskDriver_Geometry	Format information
	MSDiskDriver_Performance	Performance information and internal device name
	MSDiskDriver_PerformanceData	Performance information alone
Storage	MSStorageDriver_FailurePredictStatus	Determines whether drive is predicting a failure
	MSStorageDriver_FailurePredictData	Failure prediction data
	MSStorageDriver_FailurePredictEvent	Event fired when failure is predicted
	MSStorageDriver_FailurePredictFunction	Methods related to failure prediction
	MSStorageDriver_ATAPISmartData	ATAPI failure prediction data
	MSStorageDriver_FailurePredict-Thresholds	Vendor-specific information
	MSStorageDriver_ScsiInfoExceptions	Flags and options relative to reporting informational - exceptions
Serial	MSSerial_PortName	Name of port
	MSSerial_CommInfo	Communication parameters
	MSSerial_HardwareConfiguration	I/O resource information
	MSSerial_PerformanceInformation	Performance information
	MSSerial_CommProperties	Communication parameters
Parallel	MSParallel_AllocFreeCounts	Counts of allocation and free operations
	MSParallel_DeviceBytesTransferred	Transfer counts
IDE	MSIde_PortDeviceInfo	Pseudo-SCSI identification for an IDE port
Redbook	MSRedbook_DriverInformation	Configuration information about a device used for Redbook audio
	MSRedbook_Performance	Performance info about Redbook audio driver
Tape	MSTapeDriveParam	Feature information about tape drive
	MSTapeMediaCapacity	Information about current media
	MSTapeSymbolicName	Symbolic name (e.g., Tape0) of drive
	MSTapeDriveProblemEvent	Event used to signal a problem
	MSTapeProblemIoError	Statistics about I/O errors
	MSTapeProblemDeviceError	Summary of drive problems
Changer	MSChangerParameters	Feature information about CD changer
	MSChangerProblemEvent	Event used to signal a problem
	MSChangerProblemDeviceError	Summary of device problems
All device types	MSPower_DeviceEnable	Controls whether driver automatically powers device down
	MSPower_DeviceWakeEnable	Enables or disables system wake-up feature
	MSDeviceUI_FirmwareRevision	Revision level of device firmware

Table 10-4. *Standard Data Blocks*

10.3 Windows 98/Me Compatibility Notes

Since a well-crafted driver should support WMI, and since WMILIB isn't available in the original Windows 98, you might need to provide a virtual device driver (VxD) stub for the WMILIB functions so that your driver will load. Consult Appendix A for more information about writing a VxD stub. (The WDMSTUB filter driver discussed in the appendix doesn't include the WMILIB functions, but the appendix describes how you might invent them.)

A number of bugs afflicted the WMI support in the original retail release of Windows 98. The updates to Windows 98 (Second Edition and Service Pack 1) fixed these bugs. Even so, the standard setup procedure doesn't install WMI by default. To install it yourself, open Add/Remove Programs in the Control Panel, select the Windows Setup tab, and request installation of Web-Based Enterprise Mgmt within the Internet Tools category.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 303

Chapter 11

Controller and Multifunction Devices

Two categories of devices don't fit neatly into the PnP framework I described in Chapter 6. These categories are *controller* devices, which manage a collection of child devices, and *multifunction* devices, which have several functions on one card. These kinds of devices are similar in that their correct management entails the creation of multiple device objects with independent I/O resources.

It's very easy in Windows XP to support Peripheral Component Interconnect (PCI), Personal Computer Memory Card International Association (PCMCIA), and USB devices that conform to their respective bus standards for multifunction devices. The PCI bus driver automatically recognizes PCI multifunction cards. For PCMCIA multifunction devices, you can follow the detailed instructions in the DDK for designating MF.SYS as the function driver for your multifunction card; MF.SYS will enumerate the functions on your card and thereby cause the PnP Manager to load individual function drivers. The USB Generic Parent driver will normally load separate function drivers for each interface on a one-configuration device.

Except for USB, the original release of Windows 98 lacks the multifunction support that Windows XP provides. In Windows 98/Me, to deal with controller or multifunction devices, or to deal with nonstandard devices, you'll need to resort to more heroic means. You'll supply a function driver for your main device and supply separate function drivers for the child devices that connect to the main device. The main device's function driver will act like a miniature bus driver by enumerating the child devices and providing default handling for PnP and power requests. Writing a full-fledged bus driver is a large undertaking, and I don't intend to attempt a description of the process here. I will, however, describe the basic mechanisms you use for enumerating child devices. This information will allow you to write drivers for controller or multifunction devices that don't fit the standard molds provided by Microsoft.

11.1 Overall Architecture

In Chapter 2, Figure 2-6 illustrates the topology of device objects when a parent device, such as a bus driver, has children. Controller and multifunction devices use a similar topology. The parent device plugs in to a standard bus. The driver for the standard bus detects the parent, and the PnP Manager configures it just like any ordinary device—up to a point. After it starts the parent device, the PnP Manager sends a Plug and Play request with the minor function code *IRP_MN_QUERY_DEVICE_RELATIONS* to learn the so-called bus relations of the parent device. This query occurs for *all* devices, actually, because the PnP Manager doesn't know yet whether the device has children.

In response to the bus relations query, the parent device's function driver locates or creates additional device objects. Each of these objects becomes the physical device object (PDO) at the bottom of the stack for one of the child devices. The PnP Manager will go on to load the function and filter drivers for the child devices, whereupon you end up with a picture like that in Figure 2-6.

The driver for the parent device has to play two roles. In one role, it's the function device object (FDO) driver for the controller or the multifunction device. In the other role, it's the PDO driver for its child devices. In its FDO role, it handles PnP and power requests in the way function drivers normally handle them. In its PDO role, however, it acts as the driver of last resort for PnP and power requests.

11.1.1 Child Device Objects

The parent device driver for a multifunction device is responsible for creating PDOs for its child devices. There are two basic ways to do this task:

- A driver for a bus or device with hot-plug capability for child devices keeps a list of PDOs that it refreshes each time the PnP Manager sends a bus relations query. Such a driver will perform a hardware enumeration to detect the then-current population of devices, create new PDOs for newly detected devices, and discard PDOs for devices that no longer exist.
- A driver for a device with a fixed number of child functions creates its list of PDOs as soon as possible and reports the same list each time the PnP Manager performs a bus relations query. It destroys the PDOs at the same time it destroys its own FDO.

IPR2024-00292
KASPERSKY EX 1008 Page 305

Device Extension Structures

A minor complication for a multifunction driver is that both the FDO and all the PDOs belong to the same driver object. This means that I/O request packets (IRPs) directed to any of these device objects will come to one set of dispatch routines. The driver needs to handle PnP and power IRPs differently for FDOs and PDOs. Consequently, you need to provide a way for a dispatch function to easily distinguish between an FDO and one of the child PDOs. One way to deal with this complication is to define two device extension structures with a common beginning, as follows:

```
// The FDO extension:

typedef struct  DEVICE EXTENSION {
  ULONG flags;
  :
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

// The PDO extension:

typedef struct  PDO EXTENSION {
  ULONG flags;
  :
  } PDO EXTENSION, *PPDO EXTENSION;

// The common part:

typedef struct  COMMON EXTENSION {
  ULONG flags;
  } COMMON EXTENSION, *PCOMMON EXTENSION;

#define ISPDO 0x00000001
The dispatch routines in the driver then look like this:
NTSTATUS DispatchSomething(PDEVICE OBJECT DeviceObject, PIRP Irp)
  {
  PCOMMON EXTENSION pcx = (PCOMMON EXTENSION) DeviceObject->DeviceExtension;
  if (pcx->flags & ISPDO)
    return DispatchSomethingPdo(DeviceObject, Irp);
  else
    return DispatchSomethingFdo(DeviceObject, Irp);
  }
```

That is, you distinguish between FDO and PDO roles by examining the header that both types of device extension have in common, and then you call an FDO-specific or PDO-specific routine to handle the IRP.

Example of Creating Child Device Objects

MULFUNC, which is available in the companion content, is a very lame multifunction device: it has just two children, and we always know what they are. I just called them A and B. MULFUNC executes the following code—with more error checking than I'm showing you here—at *IRP_MN_START_DEVICE* time to create PDOs for A and B:

```
NTSTATUS StartDevice(PDEVICE OBJECT fdo, ...)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
1
  CreateChild(pdx, CHILDTYPEA, &pdx->ChildA);
  CreateChild(pdx, CHILDTYPEB, &pdx->ChildB);
  return STATUS SUCCESS;
  }

NTSTATUS CreateChild(PDEVICE EXTENSION pdx, ULONG flags, PDEVICE OBJECT* ppdo)
  {
  PDEVICE OBJECT child;
2
  IoCreateDevice(pdx->DriverObject, sizeof(PDO EXTENSION),
    NULL, FILE DEVICE UNKNOWN, FILE AUTOGENERATED DEVICE NAME, FALSE, &child);
  PPDO EXTENSION px = (PPDO EXTENSION) child->DeviceExtension;
  px->flags = ISPDO | flags;
  px->DeviceObject = child;
  px->Fdo = pdx->DeviceObject;
  child->Flags &= ~DO_DEVICE_INITIALIZING;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 306

```
    *ppdo = child;
    return STATUS_SUCCESS;
    }
```

1. *CHILDTYPEA* and *CHILDTYPEB* are additional flag bits for the *flags* member that begins the common device extension. If you were writing a true bus driver, you wouldn't create the child PDOs here—you'd enumerate your actual hardware in response to an *IRP_MN_QUERY_DEVICE_RELATIONS* and create the PDOs then.

2. We're creating a named device object here, but we're asking the system to automatically generate the name by supplying the *FILE_AUTOGENERATED_DEVICE_NAME* flag in the *DeviceCharacteristics* argument slot.

The end result of the creation process is two pointers to device objects (*ChildA* and *ChildB*) in the device extension for the parent device's FDO.

11.2 Handling PnP Requests

A controller or a multifunction parent driver has two subdispatch routines for *IRP_MJ_PNP* requests—one to handle requests it receives while wearing its FDO hat and another to handle requests it receives while wearing its PDO hat. Table 11-1 indicates the actions that the parent driver takes for each type of PnP request in its two roles. I'll explain the "Parent Vote" column later on.

PnP Request	"FDO Hat"	"PDO Hat"	Parent Vote?
IRP_MN_START_DEVICE	Normal	Succeed	N/A
IRP_MN_QUERY_REMOVE_ DEVICE	Normal	Succeed	N/A
IRP_MN_REMOVE_DEVICE	Normal	Succeed	N/A
IRP_MN_CANCEL_REMOVE_ DEVICE	Normal	Succeed	N/A
IRP_MN_STOP_DEVICE	Normal	Succeed	N/A
IRP_MN_QUERY_STOP_ DEVICE	Normal	Succeed	N/A
IRP_MN_CANCEL_STOP_ DEVICE	Normal	Succeed	N/A
IRP_MN_QUERY_DEVICE_ RELATIONS	Special pro- cessing for *BusRelations* query; other- wise normal	Special process- ing for *Target-Relations* query; otherwise ignore	No
IRP_MN_QUERY_INTERFACE	Normal	Special process- ing	No
IRP_MN_QUERY_CAPABILITIES	Normal	Delegate	No
IRP_MN_QUERY_RESOURCES	Normal	Succeed	N/A
IRP_MN_QUERY_RESOURCE_ REQUIREMENTS	Normal	Succeed	N/A
IRP_MN_QUERY_DEVICE_ TEXT	Normal	Succeed	N/A
IRP_MN_FILTER_RESOURCE_ REQUIREMENTS	Normal	Succeed	N/A
IRP_MN_READ_CONFIG	Normal	Delegate	Yes
IRP_MN_WRITE_CONFIG	Normal	Delegate	Yes
IRP_MN_EJECT	Normal	Delegate	Yes
IRP_MN_SET_LOCK	Normal	Delegate	Yes
IRP_MN_QUERY_ID	Normal	Special processing	N/A
IRP_MN_QUERY_PNP_DEVICE _STATE	Normal	Special processing	No
IRP_MN_QUERY_BUS_INFOR- - MATION	Normal	Delegate	Yes
IRP_MN_DEVICE_USAGE_ NOTIFICATION	Normal	Delegate	No
IRP_MN_SURPRISE_REMOVAL	Normal	Succeed	N/A
IRP_MN_QUERY_LEGACY_ BUS_INFORMATION	Normal	Delegate	Yes
Any other	Normal	Ignore	Yes

Table 11-1. *Parent Driver Handling of PnP Requests*

I used a shorthand notation in this table to indicate the action, as follows:

- *Normal* means "normal processing for a function driver." In other words, when wearing its FDO hat, the parent driver

IPR2024-00292
KASPERSKY EX 1008 Page 307

handles nearly every PnP request the same way a function driver would. Chapter 6 discusses a function driver's responsibilities in detail. For example, you pass all requests down the parent device stack except for a query you are causing to fail. You configure your device in response to *IRP_MN_START_DEVICE*. And so on.

- *Succeed* means to complete the IRP with *STATUS_SUCCESS*.
- *Ignore* means to complete the IRP with whatever status is already in the IRP's *IoStatus* field.
- *Delegate* means to repeat the IRP on the parent device's FDO stack and return the same results back on the PDO stack.

I'll discuss the mechanics of these actions in the next few sections of this chapter.

11.2.1 Telling the PnP Manager About Our Children

The PnP Manager inquires about the children of every device by sending an *IRP_MN_QUERY_DEVICE_RELATIONS* request with the type code *BusRelations*. Wearing its FDO hat, the parent driver responds to this request with code like the following:

```
NTSTATUS HandleQueryRelations(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = ...;
  PIO STACK LOCATION stack = ...;
1
  if (stack->Parameters.QueryDeviceRelations.Type != BusRelations)
    return DefaultPnpHandler(fdo, Irp);
2
  PDEVICE RELATIONS newrel = (PDEVICE RELATIONS)
    ExAllocatePool(PagedPool, sizeof(DEVICE RELATIONS) + sizeof(PDEVICE OBJECT));
  newrel->Count = 2;
  newrel->Objects[0] = pdx->ChildA;
  newrel->Objects[1] = pdx->ChildB;
3
  ObReferenceObject(pdx->ChildA);
  ObReferenceObject(pdx->ChildB);
4
  Irp->IoStatus.Information = (ULONG PTR) newrel;
  Irp->IoStatus.Status = STATUS SUCCESS;
  return DefaultPnpHandler(fdo, Irp);
  }
```

1. This IRP can concern several types of relations besides the bus relations we're interested in here. We simply delegate these other queries to the bus driver for the underlying hardware bus.
2. Here we allocate a structure that will contain two device object pointers. The *DEVICE_RELATIONS* structure ends in an array with a dimension of 1, so we need only add on the size of an additional pointer when we calculate the amount of memory to allocate.
3. We call *ObReferenceObject* to increment the reference counts associated with each of the device objects we put into the *DEVICE_RELATIONS* array. The PnP Manager will dereference the objects at an appropriate time.
4. We need to pass this request down to the real bus driver in case it or a lower filter knows additional facts that we didn't know. This IRP uses an unusual protocol for pass down and completion. You set the *IoStatus* as shown here if you actually handle the IRP; otherwise, you leave the *IoStatus* alone. Note the use of the *Information* field to contain a pointer to the *DEVICE_RELATIONS* structure. In other situations we've encountered in this book, the *Information* field has always held a number.

I glossed over an additional complication in the preceding code fragment that you'll notice in the code sample. An upper filter might have already installed a list of device objects in the *IoStatus.Information* field of the IRP. We must not lose that list. Rather, we must extend it by adding our own two device object pointers.

The PnP Manager automatically sends a query for bus relations at start time. You can force the query to be sent by calling this service function:

```
IoInvalidateDeviceRelations(pdx->Pdo, BusRelations);
```

A bus driver with hot-plug capability uses this function when it detects the insertion or removal of a child device. A controller or a multifunction driver with a fixed population of child devices wouldn't need to make this call.

11.2.2 PDO Handling of PnP Requests

In this section, I'll illustrate the mechanics of the "PDO Hat" column of Table 11-1. You already know from Chapter 6 and

IPR2024-00292
KASPERSKY EX 1008 Page 308

from the preceding section how to handle IRPs in the FDO role.

The Succeed Action

The parent should simply have many PnP IRPs *succeed* without doing any particular processing:

```
NTSTATUS SucceedRequest(PDEVICE OBJECT pdo, PIRP Irp)
  {
  Irp->IoStatus.Status = STATUS SUCCESS;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return STATUS SUCCESS;
  }
```

The only remarkable feature of this short subroutine is that it doesn't change the *IoStatus.Information* field of the IRP. The PnP Manager always initializes this field in some way before launching an IRP. In some cases, the field might be altered by a filter driver or the function driver to point to some data structure or another. It would be incorrect for the PDO driver to alter the field.

The Ignore Action

The parent driver can *ignore* certain IRPs. Ignoring an IRP is similar to causing it to fail with an error code except that the driver doesn't change the IRP's status fields:

```
NTSTATUS IgnoreRequest(PDEVICE OBJECT pdo, PIRP Irp)
  {
  NTSTATUS status = Irp->IoStatus.Status;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return status;
  }
```

Evidently, you're completing the IRP with whatever values already happen to be in *IoStatus.Status* and *IoStatus.Information*. The reason this strategy makes sense is that whoever originates a PnP request initializes these values to *STATUS_NOT_SUPPORTED* and 0, respectively. The PnP Manager gleans information from the fact that an IRP completes with these same values still in place, namely, that none of the drivers in the stack actually did anything with the IRP. The DDK instructions for function and filter drivers indicate that a driver that processes certain types of IRP is supposed to change the status to *STATUS_SUCCESS* before passing the IRP down the stack. Those instructions dovetail with "ignore" handling in some of Microsoft's bus drivers and in a controller or a multifunction driver built according to the pattern I'm describing in this chapter.

The Delegate Action

The parent driver can simply *delegate* some PnP requests to the real bus driver that lies underneath the parent device's FDO. Delegation in this case is not quite as simple as just calling *IoCallDriver* because by the time we receive an IRP as a PDO driver, the I/O stack is generally exhausted. We must therefore create what I call a *repeater IRP* that we can send to the driver stack we occupy as FDO driver:

```
NTSTATUS RepeatRequest(PDEVICE OBJECT pdo, PIRP Irp)
  {
  PPDO EXTENSION pdx = (PPDO EXTENSION) pdo->DeviceExtension;
  PDEVICE OBJECT fdo = pdx->Fdo;
  PDEVICE EXTENSION pfx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

1
  PDEVICE OBJECT tdo = IoGetAttachedDeviceReference(fdo);
2
  PIRP subirp = IoAllocateIrp(tdo->StackSize + 1, FALSE);

  PIO STACK LOCATION substack = IoGetNextIrpStackLocation(subirp);
  substack->DeviceObject = tdo;
  substack->Parameters.Others.Argument1 = (PVOID) Irp;
3
  IoSetNextIrpStackLocation(subirp);
  substack = IoGetNextIrpStackLocation(subirp);
  RtlCopyMemory(substack, stack,
    FIELD OFFSET(IO STACK LOCATION, CompletionRoutine));
  substack->Control = 0;
4
```


IPR2024-00292
KASPERSKY EX 1008 Page 309

```
    BOOLEAN needsvote = <I'll explain later>;
    IoSetCompletionRoutine(subirp, OnRepeaterComplete,
      (PVOID) needsvote, TRUE, TRUE, TRUE);
5
    subirp->IoStatus.Status = STATUS NOT SUPPORTED;
    IoMarkIrpPending(Irp);
    IoCallDriver(tdo, subirp);
    return STATUS PENDING
    }

NTSTATUS OnRepeaterComplete(PDEVICE OBJECT tdo, PIRP subirp, PVOID needsvote)
    {
6
    ObDereferenceObject(tdo);
    PIO STACK LOCATION substack = IoGetCurrentIrpStackLocation(subirp);
7
    PIRP Irp = (PIRP) substack->Parameters.Others.Argument1;
8
    if (subirp->IoStatus.Status == STATUS NOT SUPPORTED)
      {
      if (needsvote)
        Irp->IoStatus.Status = STATUS UNSUCCESSFUL;
      }
    else
      Irp->IoStatus = subirp->IoStatus;
9
    IoFreeIrp(subirp);
10
    IoCompleteRequest(Irp, IO NO INCREMENT);
11
    return STATUS MORE PROCESSING REQUIRED;
    }
```

1. We're going to send the repeater IRP to the topmost filter driver in the stack to which our FDO belongs. This service routine returns the address of the topmost device object, and it also adds a reference to the object to prevent the Object Manager from deleting the object for the time being.

2. When we allocate the IRP, we create an extra stack location in which we can record some context information for the completion routine we're going to install. The *DeviceObject* pointer we place in this extra location becomes the first argument to the completion routine.

3. Here we initialize the first real stack location, which is the one that the topmost driver in the FDO stack will receive. Then we install our completion routine. This is an instance in which we can't use the standard *IoCopyCurrentIrpStackLocationToNext* macro to copy a stack location: we're dealing with two separate I/O stacks.

4. We need to plan ahead for how we're going to deal with the possibility that the parent device stack doesn't actually handle this repeater IRP. Our later treatment will depend on exactly which minor function of IRP we're repeating in a way I'll describe later on. Mechanically, what we do is calculate a Boolean value—I called it *needsvote*—and pass it as the context argument to our completion routine.

5. You always initialize the status field of a new PnP IRP to hold the special value *STATUS_NOT_SUPPORTED*. The Driver Verifier will bugcheck if you don't.

6. This statement is how we release our reference to the topmost device object in the FDO stack.

7. We saved the address of the original IRP here.

8. This short section sets the completion status for the original IRP. Refer to the following main text for an explanation of what's going on here.

9. We allocated the repeater IRP, so we need to delete it.

10. We can complete the original IRP now that the FDO driver stack has serviced its clone.

11. We must return *STATUS_MORE_PROCESSING_REQUIRED* because the IRP whose completion we dealt with—the repeater IRP—has now been deleted.

The preceding code deals with a rather complex problem that afflicts the various PnP IRPs that a parent driver repeats on its FDO stack. The PnP Manager initializes PnP IRPs to contain *STATUS_NOT_SUPPORTED*. It can tell whether any driver actually handled one of these IRPs by examining the ending status. If the IRP completes with *STATUS_NOT_SUPPORTED*, the PnP Manager can deduce that no driver did anything with the IRP. If the IRP completes with any other status, the PnP

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 310

Manager knows that some driver deliberately caused the IRP either to fail or to succeed but didn't simply ignore it.

A driver that creates a PnP IRP must follow the same convention by initializing *IoStatus.Status* to *STATUS_NOT_SUPPORTED*. As I remarked, the Driver Verifier will bugcheck if you forget to do this. But this initialization gives rise to the following problem: suppose one of the devices in the child stack (that is, above the PDO for the child device) changes *IoStatus.Status* to another value before passing a particular IRP down to us in our role as PDO driver. We will create a repeater IRP, preinitialized with *STATUS_NOT_SUPPORTED*, and pass it down the parent stack (that is, the stack to which we belong in our role as FDO driver). If the repeater IRP completes with *STATUS_NOT_SUPPORTED*, what status should we use in completing the original IRP? It shouldn't be *STATUS_NOT_SUPPORTED* because that would imply that none of the child-stack drivers processed the IRP (but one did, and it changed the main IRP's status). That's where the *needsvote* flag comes in.

For some of the IRPs we repeat, we don't care whether a parent driver actually processes the IRP. We say (actually, the Microsoft developers say) that the parent drivers don't need to "vote" on the IRP. If you look carefully at *OnRepeaterComplete*, you'll see that we don't change the main IRP's ending status in this case. For other of the IRPs we repeat, we can't provide a real answer if the parent-stack drivers ignore the IRP. For these IRPs, on which the parent must "vote," we have the main IRP fail with *STATUS_UNSUCCESSFUL*. To see which IRPs belong to the "needs vote" class and which IRPs don't, take a look at the last column in Table 11-1. The minor functions for which the table indicates *N/A* are ones that the parent driver never repeats on the parent stack in the first place, by the way.

If one of the parent drivers actually *does* process the repeater IRP, however, we copy the entire *IoStatus* field, which includes *both* the *Status* and *Information* values, into the main IRP. The *Information* field might contain the answer to a query, and this copy step is how we pass the answer upward.

I did one other slightly subtle thing in *RepeatRequest*—I marked the IRP pending and returned *STATUS_PENDING*. Most PnP IRPs complete synchronously so that the call to *IoCallDriver* will most likely cause immediate completion of the IRP. So why mark the IRP pending and cause the I/O Manager unnecessary pain in the form of needing to schedule an asynchronous procedure call (APC) as part of completing the main IRP? The reason is that if we don't return *STATUS_PENDING* from our dispatch function—recall that *RepeatRequest* is running as a subroutine below the dispatch function for *IRP_MJ_PNP*—we must return exactly the same value that we use when we complete the IRP. Only our completion routine knows which value this will actually be after checking for *STATUS_NOT_SUPPORTED* and checking the *needsvote* flag, and there's no good way for our completion routine to communicate its decision back to the dispatch routine.

11.2.3 Handling Device Removal

The PnP Manager is aware of the parent-child relationship between a parent's FDO and its child PDOs. Consequently, when the user removes the parent device, the PnP Manager automatically removes all the children. Oddly enough, though, the parent driver should *not* normally delete a child PDO when it receives an *IRP_MN_REMOVE_DEVICE*. The PnP Manager expects PDOs to persist until the underlying hardware is gone. A multifunction driver will therefore not delete the children PDOs until it's told to delete the parent FDO. A bus driver for a hot-pluggable device, however, will delete a child PDO when it receives *IRP_MN_REMOVE_DEVICE* after failing to report the device during an enumeration.

11.2.4 Handling *IRP_MN_QUERY_ID*

The most important of the PnP requests that a parent driver handles is *IRP_MN_QUERY_ID*. The PnP Manager issues this request in several forms to determine which device identifiers it will use to locate the INF file for a child device. You respond by returning (in *IoStatus.Information*) a *MULTI_SZ* value containing the requisite device identifiers. The MULFUNC device has two children with the (bogus) device identifiers *WCO1104 and *WCO1105. It handles the query in the following way:

```
NTSTATUS HandleQueryId(PDEVICE OBJECT pdo, PIRP Irp)
  {
  PPDO EXTENSION pdx = (PPDO EXTENSION) pdo->DeviceExtension;
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PWCHAR idstring;
  switch (stack->Parameters.QueryId.IdType)
    {
1
  case BusQueryInstanceID:
    idstring = L"0000";
    break;
2
  case BusQueryDeviceID:
    if (pdx->flags & CHILDTYPEA)
      idstring = LDRIVERNAME L"\\*WCO1104";
    else
      idstring = LDRIVERNAME L"\\*WCO1105";
    break;
3
  case BusQueryHardwareIDs:
```


IPR2024-00292
KASPERSKY EX 1008 Page 311

```
    if (pdx->flags & CHILDTYPEA)
      idstring = L"*WCO1104";
    else
      idstring = L"*WCO1105";
    break;
  default:
    return CompleteRequest(Irp, STATUS NOT SUPPORTED, 0);
    }
  ULONG nchars = wcslen(idstring);
  ULONG size = (nchars + 2) * sizeof(WCHAR);
  PWCHAR id = (PWCHAR) ExAllocatePool(PagedPool, size);
  wcscpy(id, idstring);
  id[nchars + 1] = 0;
  return CompleteRequest(Irp, STATUS SUCCESS, (ULONG PTR) id);
  }
```

1. The *instance* identifier is a single string value that uniquely identifies a device of a particular type on a bus. Using a constant such as "0000" won't work if more than one device of the parent type can appear in the computer.
2. The *device* identifier is a single string of the form "enumerator\type" and basically supplies two components in the name of the hardware registry key. Our ChildA device's hardware key will be in ...\Enum\Mulfunc*WCO1104\0000, for example.
3. The *hardware* identifiers are strings that uniquely identify a type of device. In this case, I just made up the pseudo-EISA (Extended Industry Standard Architecture) identifiers *WCO1104 and *WCO1105.

NOTE
Be sure to use your own name in place of MULFUNC if you construct a device identifier in the manner I showed you here. To emphasize that you shouldn't just copy my sample program's name in a hard-coded constant, I wrote the code to use the manifest constant *LDRIVERNAME*, which is defined in the DRIVER.H file in the MULFUNC project.

The Windows 98/Me PnP Manager will tolerate your supplying the same string for a device identifier that you supply for a hardware identifier, but the Windows XP PnP Manager won't. I learned the hard way to supply a made-up enumerator name in the device ID. Calling *IoGetDeviceProperty* to get the PDO's enumerator name leads to a bug check because the PnP Manager ends up working with a *NULL* string pointer. Using the parent's enumerator name—ROOT in the case of the MULFUNC sample—leads to the bizarre result that the PnP Manager brings the child devices back after you delete the parent!

11.2.5 Handling *IRP_MN_QUERY_DEVICE_RELATIONS*

The last PnP request to consider is *IRP_MN_QUERY_DEVICE_RELATIONS*. Recall that the FDO driver answers this request by providing a list of child PDOs for a bus relations query. Wearing its PDO hat, however, the parent driver need only answer a request for the so-called target device relation by providing the address of the PDO:

```
NTSTATUS HandleQueryRelations(PDEVICE OBJECT pdo, PIRP Irp)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  NTSTATUS status = Irp->IoStatus.Status;
  if (stack->Parameters.QueryDeviceRelations.Type == TargetDeviceRelation)
    {
    PDEVICE RELATIONS newrel = (PDEVICE RELATIONS)
      ExAllocatePool(PagedPool, sizeof(DEVICE RELATIONS));
    newrel->Count = 1;
    newrel->Objects[0] = pdo;
    ObReferenceObject(pdo);
    status = STATUS SUCCESS;
    Irp->IoStatus.Information = (ULONG PTR) newrel;
    }
  Irp->IoStatus.Status = status;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return
```

11.2.6 Handling *IRP_MN_QUERY_INTERFACE*

IRP_MN_QUERY_INTERFACE allows any driver in a PnP device stack to locate a direct-call interface in any lower driver in the same stack. A direct-call interface allows a driver to call directly one or more functions in another driver without first constructing an IRP. The basic concepts involved in the direct call interface mechanism are these:

IPR2024-00292
KASPERSKY EX 1008 Page 312

- A GUID identifies each unique direct-call interface. The interface itself is embodied in a structure that contains pointers to the functions that implement the interface's methods.
- A driver that wants to use a particular interface originates a *QUERY_INTERFACE* request that includes the identifying GUID and an instance of the interface structure. Thereafter, such a driver can directly call functions pointed to by members of the interface structure. When the driver is done using the direct-call interface, it calls the *InterfaceDereference* function to release its reference to the exporting driver.
- The driver that exports a particular interface looks for *QUERY_INTERFACE* requests that specify that interface's GUID. To handle such a request, the exporting driver fills in fields in an interface structure provided by the caller with pointers to functions inside that driver. That driver also undertakes not to unload from memory until the caller releases its reference to the interface.

I'll explain these concepts in more detail now. Refer to the MULFUNC sample in the companion content for a fully worked out example that uses these concepts in a real driver.

Identifying an Interface

You identify a direct-call interface by creating and publishing a GUID and a structure. Conventionally, the symbolic name of the GUID will be of the form *GUID_XXX_STANDARD*, to match the pattern established by the DDK header WDMGUID.H. For example, MULFUNC exports a direct-call interface with the following GUID:

```
DEFINE GUID(GUID RESOURCE SUBALLOCATE STANDARD, 0xaa04540,
  0x6fd1, 0x11d3, 0x81, 0xb5, 0x0, 0xc0, 0x4f, 0xa3,
  0x30, 0xa6);
```

The purpose of the *RESOURCE_SUBALLOCATE* interface is to permit child devices to divvy up I/O resources that technically belong to the parent device; I'll discuss how this works at the end of this chapter.

The structure associated with the *RESOURCE_SUBALLOCATE* interface is as follows (note that *INTERFACE* is declared in a DDK header because it's the base class of every direct-call interface):

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

struct  RESOURCE_SUBALLOCATE_STANDARD : public INTERFACE {
  PGETRESOURCES GetResources;
  };
typedef struct  RESOURCE_SUBALLOCATE_STANDARD
  RESOURCE_SUBALLOCATE_STANDARD,
  *PRESOURCE_SUBALLOCATE_STANDARD;
```

In other words, the *RESOURCE_SUBALLOCATE* interface includes a *GetResources* function as well as the *InterfaceReference* and *InterfaceDereference* functions from the base class.

Locating and Using a Direct-Call Interface

A driver that wants to use a direct-call interface exported by a driver below it in the PnP stack constructs and sends a *QUERY_INTERFACE* request. Table 11-2 indicates the parameters in *Parameters.QueryInterface* for this request.

Parameter	Meaning
InterfaceType	Pointer to GUID that identifies the interface
Size	Size of the interface structure pointed to by the Interface parameter
Version	Version of the interface structure
Interface	Address of interface structure to be filled in by the exporting driver
InterfaceSpecificData	Additional data expected by the driver that exports the interface—depends on the interface

Table 11-2. *Parameters for IRP_MN_QUERY_INTERFACE*

Here's an example of one way to issue the *QUERY_INTERFACE* request:

```
RESOURCE_SUBALLOCATE_STANDARD suballoc; // <== the eventual result
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 313

```
KEVENT event;
KeInitializeEvent(&event, NotificationEvent, FALSE);
IO STATUS BLOCK iosb;
PIRP Irp = IoBuildSynchronousFsdRequest(IRP MJ PNP,
  pdx->LowerDeviceObject, NULL, 0, NULL, &event, &iosb);
PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->MinorFunction = IRP MN QUERY INTERFACE;
stack->Parameters.QueryInterface.InterfaceType =
  &GUID RESOURCE SUBALLOCATE STANDARD;
stack->Parameters.QueryInterface.Size = sizeof(suballoc);
stack->Parameters.QueryInterface.Version =
  RESOURCE SUBALLOCATE STANDARD VERSION;
stack->Parameters.QueryInterface.Interface = &suballoc;
stack->Parameters.QueryInterface.InterfaceSpecificData = NULL;
NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
if (status == STATUS PENDING)
  {
  KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
  status = Irp->IoStatus.Status;
  }
```

In this example, we use a synchronous IRP to communicate down the stack. We expect somebody underneath to fill in the *suballoc* structure and complete the IRP with a success status.

If the interface query succeeds, we can then directly call the functions to which members of the interface structure point. Ordinarily, each interface function requires a context argument taken from the returned interface structure, as in this example (see the *SUBALLOC* filter that's part of the MULFUNC sample):

```
PCM RESOURCE LIST raw, translated;
status = suballoc.GetResources(suballoc.Context, pdx->Pdo, &raw, &translated);
```

The other arguments to an interface function, and the meaning of the return value, are matters to be decided by the designer of the interface.

When you're done using a direct-call interface, make the following call:

```
suballoc.InterfaceDereference(suballoc.Context);
Exporting a Direct-Call Interface
```

To export a direct-call interface, you need to handle the *IRP_MN_QUERY_INTERFACE* request. The first step will be to examine the interface GUID in the stack parameters to see whether the caller is trying to locate an interface you support. For example:

```
if (*stack->Parameters.QueryInterface.InterfaceType !=
  GUID RESOURCE SUBALLOCATE STANDARD)
  <default handling>
```

NOTE
The DDK headers contain *operator* statements to define C++ comparison operators for GUIDs. If you're writing your driver exclusively in C, use the *IsEqualGuid* function instead.

The DDK implies that a bus driver should fail a query for an unknown interface that it receives in its PDO role: "[A] driver that handles this IRP should avoid passing the IRP to another device stack to get the requested interface. Such a design would create dependencies between the device stacks that are difficult to manage. For example, the device represented by the second device stack cannot be removed until the appropriate driver in the first stack dereferences the interface." I'll have to disagree and advise you to do the contrary in a controller or a multifunction parent driver. There is no other way for a child device driver to get access to functionality exported by the real bus. For the record, the parent device stack can't be removed until all child device stacks have been removed anyway, and child drivers should be clever enough to release their references to direct-call interfaces as part of their shutdown processing.

If the interface has evolved through more than one version, the next step in handling the interface query will be to decide which version of the interface to provide. A convenient convention is to begin numbering interface versions with 1 and increment the version number by 1 each time something important in the interface changes. Provide a manifest constant for the current version in the same header file that defines the interface GUID and structure. Callers will specify their desired version number in the IRP parameters by using this manifest constant, which effectively pinpoints the version of the structure with which they were compiled. You can then negotiate down to the oldest of the requested version and the newest version supported by your driver. For example:

IPR2024-00292
KASPERSKY EX 1008 Page 314

```
USHORT version = RESOURCE SUBALLOCATE STANDARD VERSION;
if (version > stack->Parameters.QueryInterface.Version)
  version = stack->Parameters.QueryInterface.Version;
if (version == 0)
  return CompleteRequest(Irp, Irp->IoStatus.Status);
```

If you start your version numbering with 1, the version number 0 can occur only if the caller has asked for version number 0. The correct response in that case is to complete the IRP with whatever initial status is in the IRP—this value will usually be *STATUS_NOT_SUPPORTED*.

The third step is to initialize the interface structure provided by the caller. For example:

```
if (stack->Parameters.QueryInterface.Size <
  sizeof(RESOURCE SUBALLOCATE STANDARD))
  return CompleteRequest(Irp, STATUS INVALID PARAMETER);
PRESOURCE SUBALLOCATE STANDARD ifp = (PRESOURCE SUBALLOCATE STANDARD)
  stack->Parameters.QueryInterface.Interface;
ifp->Size = sizeof(RESOURCE SUBALLOCATE STANDARD);
ifp->Version = 1;
ifp->Context = (PVOID) fdx;
ifp->InterfaceReference = (PINTERFACE REFERENCE)
  SuballocInterfaceReference;
ifp->InterfaceDereference = (PINTERFACE DEREFERENCE)
  SuballocInterfaceDereference;
ifp->GetResources = (PGETRESOURCES) GetChildResources;
```

Finally you should reference the interface in such a way that your driver will stay loaded until the caller calls the *InterfaceDereference* function.

11.2.7 Handling *IRP_MN_QUERY_PNP_DEVICE_STATE*

In some situations, you'll want to suppress the Device Manager display of some or all child devices. To suppress the display, add the flag *PNP_DEVICE_DONT_DISPLAY_IN_UI* to the device flags reported in response to *IRP_MN_QUERY_PNP_DEVICE_STATE*. Apart from this optional step, you should delegate the IRP to the parent stack, as described earlier.

11.3 Handling Power Requests

Wearing its FDO hat, a controller or a multifunction driver handles *IRP_MJ_POWER* requests exactly as described in Chapter 8, with one small exception that I'll discuss presently in connection with *IRP_MN_WAIT_WAKE*. Wearing its PDO hat, the controller or the driver unconditionally causes to succeed power requests, other than *IRP_MN_WAIT_WAKE*, for which special processing is required. Table 11-3 summarizes these actions.

PnP Request	"FDO Hat"	"PDO Hat"
IRP_MN_POWER_SEQUENCE	Normal	Complete
IRP_MN_QUERY_POWER	Normal	Succeed
IRP_MN_SET_POWER	Normal	Special handling
IRP_MN_WAIT_WAKE	Complete child IRPs; otherwise normal	Special handling
Other	Normal	Complete

In the remainder of this section, I'll describe the mechanics of handling power requests, insofar as they're different from standard function driver handling.

The Complete Action

☑ Wearing the PDO hat, the parent driver completes any power IRP it doesn't understand with whatever *Status* and *Information* value are already in the IRP. Part of the Driver Verifier's initialization for I/O Verification is to make sure you do this.

```
NTSTATUS DefaultPowerHandler(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PoStartNextPowerIrp(Irp);
  NTSTATUS status = Irp->IoStatus.Information;
  IoCompleteRequest(Irp, IO NO INCREMENT);
  return status;
  }
```


IPR2024-00292
KASPERSKY EX 1008 Page 315

The Succeed Action

When the parent driver causes an *IRP_MJ_POWER* request for a child PDO to succeed, it calls *PoStartNextPowerIrp* and then completes the IRP, as in this fragment:

```
NTSTATUS HandleQueryPower(PDEVICE OBJECT pdo, PIRP Irp)
  {
  PoStartNextPowerIrp(Irp);
  return CompleteRequest(Irp, STATUS SUCCESS, 0);
  }
```

The way this fragment differs from what a function driver would do is that the parent driver, being the end of the line, has no one to pass the IRP down to and must, therefore, complete the request.

You should not repeat an *IRP_MN_QUERY_POWER* on the parent device stack. Doing so in Windows 98/Me causes the Configuration Manager to query the child devices recursively in an infinite loop that eventually exhausts the execution stack. In either system, the Power Manager is already aware of the parent-child relationships between devices and orchestrates the necessary queries on its own.

Handling Device *IRP_MN_SET_POWER*

An *IRP_MN_SET_POWER* for a child PDO requires extra work if it specifies a device power state. If it's possible for the parent device to independently control the power state of a child device, the parent driver needs to make that happen. Regardless of whether the child device has a power state independent of the parent, the parent driver should call *PoSetPowerState* to notify the Power Manager of the new power state. Then it should call *PoStartNextPowerIrp* and complete the IRP.

```
NTSTATUS HandleSetPower(IN PDEVICE OBJECT pdo, IN PIRP Irp)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  if (stack->Parameters.Power.Type == DevicePowerState)
    {
    <set child power level>
    PoSetPowerState(pdo, DevicePowerState,
      stack->Parameters.Power.State);
    }
  PoStartNextPowerIrp(pdo);
  CompleteRequest(Irp, STATUS SUCCESS, 0);
  <possibly change parent power level>
  return STATUS SUCCESS;
  }
```

In addition, the parent driver might want to change the power level of the parent device to the lowest level consistent with the power levels of all the child devices. For example, if all child devices are in the D3 state, it's likely that the parent device should be in D3 as well. If any child device is in the D0 state, it's likely that the parent device should be in D0 as well. Note that MULFUNC doesn't illustrate this behavior.

Handling *IRP_MN_WAIT_WAKE*

A child device's system wake-up feature will likely depend on the parent device having wake-up capability too. For this reason, a controller or a multifunction driver has additional responsibilities for handling *IRP_MN_WAIT_WAKE* that an ordinary function driver wouldn't have.

When a parent driver receives an *IRP_MN_WAIT_WAKE* for a child PDO, it should take the following actions:

- If the device capabilities indicate that the parent device is incapable of waking the system under any circumstances, the parent should cause the request to fail with *STATUS_NOT_SUPPORTED*. (This is what MULFUNC does.)
- If an *IRP_MN_WAIT_WAKE* is already outstanding for the same child device, the parent should have the new request fail with *STATUS_DEVICE_BUSY*.
- If the child device is already in so low a power state that it couldn't wake the system, or if the device can't wake the system from the power state specified in the IRP, the parent driver should cause the IRP to fail with *STATUS_INVALID_DEVICE_STATE*. Strictly speaking, the function driver shouldn't originate a *WAIT_WAKE* request if either of these predicates is true, but it's still up to the parent driver to enforce these rules.
- Otherwise, the parent driver should mark the IRP pending, cache the IRP in a cancel-safe way, and return STATUS_PENDING. You can, for example, adapt the I/O control (IOCTL) caching scheme discussed in Chapter 9, or you can use a cancel-safe queue (see Chapter 5) as a parking place. This case will be the normal one for a device that

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 316

actually supports wake-up.

In the last case, the parent driver should also originate its own *WAIT_WAKE* request on the parent device stack if it hasn't already done so. In this respect, the parent driver is simply doing what any wake-up-capable function driver would do except that it treats the pendency of a child *WAIT_WAKE* as an additional trigger for sending its own IRP.

If the parent driver's *WAIT_WAKE* subsequently completes with a success status, the parent should complete one or more of the outstanding child *WAIT_WAKE* requests. If the parent can determine that a particular child woke up, the parent should complete just that child's IRP. Otherwise, it should complete the IRPs belonging to all children. This step ensures that affected child function drivers handle the wake-up signal appropriately.

If the parent's *WAIT_WAKE* subsequently completes with an error, the parent driver should ordinarily complete all child *WAIT_WAKE* requests with the same code. This advice sounds like a more all-encompassing rule than it really is. Only two failure codes should occur in practice. *STATUS_CANCELLED* means that the parent driver itself has decided to cancel its outstanding *WAIT_WAKE* in preparation for shutting down or because the end user has disabled the parent device's wake-up feature. *STATUS_INVALID_DEVICE_STATE* indicates that the system or parent device power state is too low to support wake-up. In either case, the parent should alert all children that their own wake-up features are being disabled by causing their *WAIT_WAKE* requests to fail.

Case Study in Multifunction Wake-Up

I once wrote a driver for the standard USB SmartCard reader (CCID) class. The CCID specification allows for multiple card slots, and the natural way to implement that functionality is by means of a multifunction driver that creates multiple, identical, child SmartCard reader devices. Many reader manufacturers decided to add system wake-up to their feature set, such that inserting or removing a card would bring the computer out of standby.

I deemed it unlikely that an end user would want to exercise independent control over each of the slots in a multislot reader. Accordingly, I decided to hide the child devices in the Device Manager using the technique described earlier in this chapter. Consequently, the Device Manager property page for the parent device would be the only way for the end user to control whether the wake-up feature was enabled or disabled.

In this driver, then, the primary responsibility for wake-up rested with the parent driver in its FDO role. The child function driver (which was actually embedded in the same executable file as the parent driver) went through the motions of creating and handling *WAIT_WAKE* requests, but the parent driver essentially ignored these requests beyond simply caching and completing them.

The only quirk in this driver related to how the child driver picked a power state. Since the standard Device Manager hookup for enabling the children's wake-up feature was missing, the child driver had to find out through a back door that the parent was enabled for wake-up and select a compatible device power state. Because I decided to package the child function driver in the same executable with the parent driver, this was easy to arrange.

11.4 Handling Child Device Resources

If your device is a controller type, the child devices that plug in to it presumably claim their own I/O resources. If you have an automated way to discover the devices' resource requirements, you can return a list of them in response to an *IRP_MN_QUERY_RESOURCE_REQUIREMENTS* request. If there is no automated way to discover the resource requirements, the child device's INF file should have a LogConfig section to establish them.

If you're dealing with a multifunction device, chances are that the parent device claims all the I/O resources that the child functions use. If the child functions have separate WDM drivers, you have to devise a way to separate the resources by function and let each function driver know which ones belong to it. This task is not simple. The PnP Manager normally tells a function driver about its resource assignments in an *IRP_MN_START_DEVICE* request. (See the detailed discussion in Chapter 7.) There's no normal way for you to force the PnP Manager to use some of *your* resources instead of the ones it assigns, though. Note that responding to a requirements query or a filter request doesn't help because those requests deal with requirements that the PnP Manager will then go on to satisfy using new resources.

Microsoft's MF.SYS driver deals with resource subdivision by using some internal interfaces with the system's resource arbitrators that aren't accessible to us as third-party developers. There are two different ways of subdividing resources: one that works in Windows XP and another one that works in Windows 98/Me. Since we can't do what MF.SYS does, we need to find some other way to suballocate resources owned by the parent device.

If you can control all of the child device function drivers, your parent driver can export a direct-call interface. In this case, child drivers obtain a pointer to the interface descriptor by sending an *IRP_MN_QUERY_INTERFACE* request to the parent driver. They call functions in the parent driver at start device and stop device time to obtain and release resources that the parent actually owns.

If you can't modify the function drivers for your child devices, you can solve the resource subdivision problem by installing a tiny upper filter—see Chapter 16—above each of the child device's FDOs. The only purpose of the filter is to plug a list of assigned resources in to each *IRP_MN_START_DEVICE*. The filter can communicate via a direct-call interface with the parent

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 317

driver. MULFUNC actually works this way, and you can study it to learn more about the mechanics.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 318

Chapter 12

The Universal Serial Bus

End user convenience is the keynote of the universal serial bus (USB). The Plug and Play (PnP) concept has simplified the process of installing certain types of hardware on existing PCs. However, configuration issues continue to plague end users with respect to legacy devices such as serial and parallel ports, keyboards, and mice. Port availability is one of the factors that have historically limited proliferation of peripherals, including modems, answering machines, scanners, and personal digital assistants. USB helps solve these problems by providing a uniform method of connecting a potentially large number of self-identifying devices through a single PC port.

Although this book concerns software, a discussion of some of the electrical and mechanical aspects of USB is in order because they're important to software developers. From the end user's point of view, USB's main feature is the use by every device of an identical 4-conductor wire with a standardized plug that fits into a socket on the back of the PC or on a hub device plugged in to the PC. Furthermore, you can attach or remove USB devices at will without explicitly opening or closing the applications that use them and without worrying about electrical damage.

This chapter covers two broad topics. In the first part of the chapter, I'll describe the programming architecture of USB. This architecture encompasses several ideas, including a hierarchical method for attaching devices to a computer, a generic scheme for power management, and a standard for self-identification that relies on a hierarchy of descriptors on board the hardware. The USB architecture also employs a scheme for subdividing fixed-duration *frames* and *microframes* into *packets* that convey data to and from devices. Finally, USB allows for four different ways of transporting data between the host computer and *endpoints* on devices. One method, named *isochronous*, permits a fixed amount of data to be moved without error correction at regular intervals—possibly as frequently as three times in every microframe. The other methods, named *control*, *bulk*, and *interrupt*, allow data to be transmitted with automatic error correction.

In the second part of this chapter, I'll describe the additional features of a Windows Driver Model driver for a USB device over and above the features you already know about. Rather than communicate directly with hardware by using hardware abstraction layer (HAL) function calls, a USB driver relies heavily on the bus driver and a kernel-mode library named USBD.SYS. To send a request to its device, the driver creates a *USB request block* (URB), which it submits to the bus driver. Configuring a USB device, for example, requires the driver to submit several URBs for reading descriptors and sending commands. The bus driver in turn schedules requests onto the bus according to demand and available bandwidth.

The ultimate source for information about USB is the official specification, which was at revision level 2.0 when this book went to press. The specification and various other documents produced by the USB committee and its working groups were available on line at http://www.usb.org/developers/.

Note on Sample Programs
Anchor Chips, Incorporated, kindly provided me one of their EZ-USB development kits that I used to develop the sample drivers for the first edition. Anchor Chips was subsequently acquired by Cypress Semiconductor (www.cypress.com). The Cypress Semiconductor USB chip set revolves around a modified 8051 microprocessor and additional core logic to perform some of the low-level protocol functions mandated by the USB specification. The development board also contains additional external memory, a UART and serial connector, a set of push buttons, and an LED readout to facilitate development and debugging of 8051 firmware using Cypress Semiconductor's software framework. One of the key features of the Cypress Semiconductor chip set is that you can download firmware over the USB connection easily. For a programmer like me with a phobia for hardware in general and EEPROM programming in particular, that feature is a godsend.

The USB sample drivers in the companion content illustrate the simplest possible USB devices and stand alone as examples of how to perform various tasks. If you happen to have a Cypress Semiconductor development kit, however, you can also try out these samples with real firmware. Each sample contains a WDM driver in a SYS subdirectory, a MicrosoftWin32 test program in a TEST subdirectory, and a firmware program in an EZUSB directory. You can follow the directions in the HTM files included with each sample to build these components or simply to install the prebuilt versions that are in the companion content.

12.1 Programming Architecture

The authors of the USB specification anticipated that programmers would need to understand how to write host and device software without necessarily needing or wanting to understand the electrical characteristics of the bus. Chapter 5, "USB Data Flow Model," and Chapter 9, "USB Device Framework," of the specification describe the features most useful to driver authors. In this section, I'll summarize those chapters.

IPR2024-00292
KASPERSKY EX 1008 Page 319

12.1.1 Device Hierarchy

Figure 12-1 illustrates the topology of a simple USB setup. A host controller unit connects to the system bus in the same way other I/O devices might. The operating system communicates with the host controller by means of I/O ports or memory registers, and it receives event notifications from the host controller through an ordinary interrupt signal. The host controller in turn connects to a tree of USB devices. One kind of device, called a *hub*, serves as a connection point for other devices. The host controller includes a *root hub*. Hubs can be daisy chained together to a maximum depth defined by the USB specification. Currently up to five hubs can be chained from the root hub, for an overall tree depth of seven. Other kinds of devices, such as cameras, mice, keyboards, and so on, plug in to hubs. For the sake of precision, USB uses the term *function* to describe a device that isn't a hub. The specification currently allows for up to 127 functions and hubs to be attached to the bus.

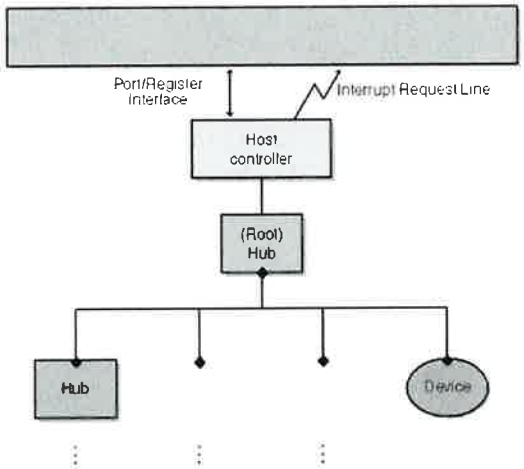


Figure 12-1. *Hierarchy of USB devices.*

High-Speed, Full-Speed, and Low-Speed Devices

The USB specification categorizes devices by communication speed. A USB 2.0 controller drives the bus at 480 megabits per second. Devices (both hubs and functions) can operate at high speed (480 megabits per second), full speed (12 megabits per second), or low speed (1.5 megabits per second). In USB 2.0, hubs are responsible for communicating with full-speed and low-speed devices using a scheme that interferes as little as possible with the high-speed signaling employed by the bus and high-speed devices.

The previous version of the USB specification (1.1) provided just for full-speed and low-speed devices. Communication normally occurs on a 1.1 bus at full speed, and hubs normally don't send data to low-speed devices. The operating system prefaces any message destined for a low-speed device with a special *preamble* packet that causes the hubs to temporarily enable the low-speed devices.

Power

The USB cable carries power as well as data signals. Each hub can supply electrical power to the devices attached to it and, in the case of subsidiary hubs, to downstream devices as well. USB imposes limits on how much power a bus-powered device can consume. These limits vary depending on whether the device is plugged in to a powered hub, how far the device is from the nearest powered hub, and so on. In addition, USB allows devices to operate in a suspended state and consume very little power—just enough to support wake-up and configuration signalling. Instead of relying on bus power, you can build independently powered hubs and devices. (In fact, the Windows Hardware Quality Lab (WHQL) accepts bus-powered hub devices for testing only when they're part of a composite device rather than being real hubs with ports for plugging devices in to.)

USB devices are able to awaken the system from a low-power state. When the system goes to low power, the operating system places the USB in the low-power state as well. A device possessing an enabled remote wake-up feature can later signal upstream to awaken upstream hubs, the USB host controller, and eventually the entire system.

USB device designers should be aware of some limitations on wake-up signalling. First, remote system wake-up works only on a computer with an Advanced Configuration and Power Interface (ACPI)-enabled BIOS. Older systems support either Advanced Power Management (APM) or no power-management standard at all. I've also found tremendous variability among PCs in their ability to support USB wake-up. On a trip to a computer superstore in mid-2002, I found only one notebook computer that would respond to a USB device's wake-up signal. I know of no principled way except by experimentation to find out when a given computer and operating system combination will work in this regard.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 320

USB hubs can support a power-management feature called *selective suspend*. This feature allows a hub to suspend ports individually and has the overall purpose of facilitating power management in battery-operated devices. Windows XP supports this feature by means of a special I/O control (IOCTL) that I'll discuss later in this chapter.

12.1.2 What's in a Device?

In general, each USB device can have one or more *configurations* that govern how it behaves. See Figure 12-2. Configurations of a single device can differ in their power consumption, their ability to remotely wake the computer, and in their populations of interfaces. Microsoft drivers invariably work with just the first configuration of a device. The Microsoft support for composite devices won't engage if the device has multiple configurations. Consequently, multiconfiguration devices seem to be rare in practice, and Microsoft discourages people from designing new ones. I've heard of just these few scenarios in which multiple configurations would make sense:

- An Integrated Services Digital Network (ISDN) communications device that presents either two 56-Kb channels or one 128-Kb channel
- A device that presents a simple configuration for use by the BIOS and a more complex one for use by Windows drivers
- A trackball that can be configured as either a mouse or a joystick

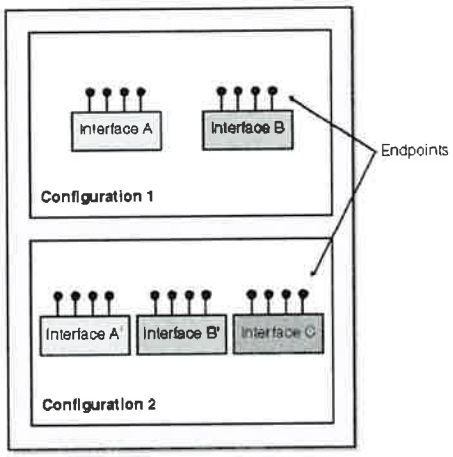


Figure 12-2. *Device configurations, interfaces, and endpoints.*

Each configuration of a device embodies one or more *interfaces* that prescribe how software should access the hardware. This concept of an interface is similar to the concept I discussed in Chapter 2 in connection with naming devices. That is, devices that support the same interface are essentially interchangeable in terms of software because they respond to the same commands in the same specified way. Also, interfaces frequently have *alternate settings* that correspond to different bandwidth requirements.

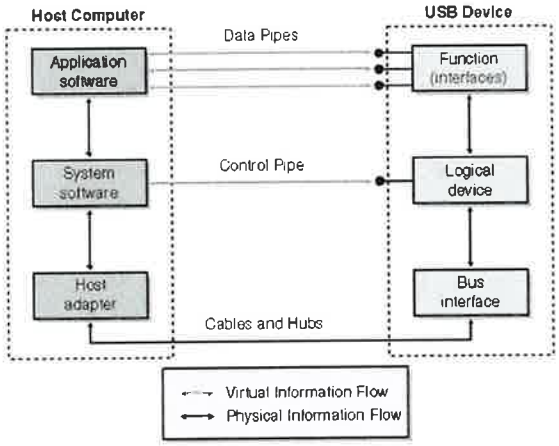


Figure 12-3. *Layered model for USB communication.*

IPR2024-00292
KASPERSKY EX 1008 Page 321

A device interface exposes one or more *endpoints*, each of which serves as a terminus for a communications pipe. Figure 12-3 diagrams a layered communication model that illustrates the role of a pipe and an endpoint. At the lowest level, the USB wire connects the host bus controller to the bus interface on a device. At the second level, a *control pipe* connects system software to a logical device. At the third and highest level, a bundle of pipes connects client software with the collection of interfaces that constitutes the device's function. Information actually flows vertically up and down both sides of the diagram, but it's useful to think of the pipes as carrying information horizontally between the corresponding layers.

A set of drivers provided by Microsoft occupies the lower edge of the system software box in the figure. These drivers include a host controller driver (USBOHCI.SYS, USBUHCD.SYS, or USBEHCI.SYS), a hub driver (USBHUB.SYS), and a library used by all the system and client drivers (USBD.SYS). For convenience, I'll lump all of these drivers together under the name *parent driver*. Collectively, these drivers manage the hardware connection and the mechanics of communicating over the various pipes. WDM drivers, such as the ones you and I might write, occupy the upper edge of the system software box. Broadly speaking, the job of a WDM driver is to translate requests from client software into transactions that the parent driver can carry out. Client software deals with the actual functionality of the device. For example, an image-rendering application might occupy the client software slot opposite a still-image function such as that of a digital camera.

12.1.3 Information Flow

USB defines four methods of transferring data, as summarized in Table 12-1. The methods differ in the amount of data that can be moved in a single transaction—see the next section for an explanation of the term *transaction*—in whether any particular periodicity or latency can be guaranteed, and in whether errors will be automatically corrected. Each method corresponds to a particular type of endpoint. In fact, endpoints of a given type (that is, control, bulk, interrupt, or isochronous) always communicate with the host by using the corresponding transfer type.

Transfer Type	Description	Lossless?	Latency Guarantee?
Control	Used to send and receive structured information of a control nature	Yes	Best effort
Bulk	Used to send or receive blocks of unstructured data	Yes	No
Interrupt	Like a bulk pipe but includes a maximum latency	Yes	Polled at guaranteed minimum rate
Isochronous	Used to send or receive blocks of unstructured data with guaranteed periodicity	No	Read or written at regular intervals

Table 12-1. *Data Transfer Types*

Endpoints have several attributes in addition to their type. One endpoint attribute is the maximum amount of data that the endpoint can provide or consume in a single transaction. Table 12-2 indicates the maximum values for each endpoint type for each speed of device. In general, any single transfer can involve less than the maximum amount of data that the endpoint is capable of handling. Another attribute of an endpoint is its direction, described as either *input* (information moves from the device to the host) or *output* (information moves from the host to the device). Finally, each endpoint has a number that functions along with the input/output direction indicator as the address of the endpoint.

Transfer Type	High Speed	Full Speed	Low Speed
Control	64	8, 16, 32, or 64	8
Bulk	≤ 512	8, 16, 32, or 64	NA (Low-speed devices can't have bulk endpoints.)
Interrupt	≤ 1024	≤ 64	≤ 8
Isochronous	≤ 3072	≤ 1023	NA (Low-speed devices can't have isochronous endpoints.)

Table 12-2. *Allowable Endpoint Maximum Packet Sizes*

USB uses a *polling* protocol in which the host requests the device to carry out some function on a more or less regular basis. When a device needs to send data to the host, the host must somehow note this and issue a request to the device to send the data. In particular, USB devices don't interrupt the host computer in the traditional sense. In place of an asynchronous interrupt, USB provides interrupt endpoints that the host polls periodically. The host polls interrupt and isochronous endpoints at a frequency specified by an option in the endpoint descriptor, as follows:

- For a USB 2.0 device operating at high speed, the polling interval, *bInterval*, must be in the range 1 through 16, inclusive, and specifies polling every $2^{bInterval-1}$ microframes.
- For a USB 2.0 device operating at full speed, the polling interval must be in the range 1 through 16, inclusive, and specifies polling every $2^{bInterval-1}$ frames.
- For a USB 1.1 device operating at full speed, an isochronous endpoint must specify a polling interval of 1 and an interrupt endpoint can specify a polling interval of 1 through 255, inclusive. Note that a USB 2.0 host driver need not distinguish between 2.0 and 1.1 devices when interpreting the polling interval for an isochronous device, since $2^{1-1} == 1$.
- For a USB 1.1 device operating at low speed, an interrupt endpoint must specify a polling interval of 10 through 255. There are no isochronous endpoints in a low-speed device.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 322

Information Packaging

When a client program sends or receives data over a USB pipe, it first calls a Win32 API that ultimately causes the function driver (that's us) to receive an I/O request packet (IRP). The driver's job is to direct the client request into a pipe ending at the appropriate endpoint on the device. It submits the requests to the bus driver, which breaks the requests into *transactions*. The bus driver schedules the transactions for presentation to the hardware. Information flows on a USB 2.0 bus in *microframes* that occur once every 125 microseconds and on a USB 1.1 bus in *frames* that occur once every millisecond. The bus driver must correlate the duration of all outstanding transactions so as to fit them into frames and microframes. Figure 12-4 illustrates the result of this process.

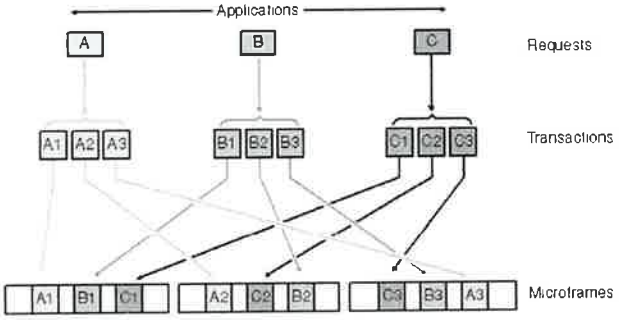


Figure 12-4. *Transaction and frame model for information flow.*

When a host communicates over a USB 2.0 bus to a full-speed or low-speed device, the *transaction translation* feature of a hub provides intermediate buffering to allow the upstream bus (that is, the bus on the host side of the hub) to continue running at high speed. In effect, a USB 2.0 hub operates full-speed and low-speed downstream ports (that is, ports on the side of the hub away from the host) as a USB 1.1 bus, with frames scheduled every millisecond.

When a host communicates over a USB 1.1 bus to a low-speed device, it introduces a special *preamble* packet to switch the bus signalling to low speed for the duration of a single transaction. Except at these times, low-speed devices are out of the signalling circuit.

In USB, a transaction has one or more *phases*. A phase is a token, data, or handshake packet. Depending on the type, a transaction consists of a *token phase*, an optional *data phase*, and an optional *handshake phase*, as shown in Figure 12-5. During the token phase, the host transmits a packet of data to all currently configured devices. The token packet includes a device address and (often) an endpoint number. Only the addressed device will process the transaction; devices neither read nor write data on the bus for the duration of transactions addressed to other devices. During the data phase, data is placed on the bus. For output transactions, the host puts data on the bus and the addressed device consumes it. For input transactions, the roles are reversed and the device places data on the bus for consumption by the host. During the handshake phase, either the device or the host places a packet on the bus that provides status information. When a device provides the handshake packet, it can send an *ACK* packet to indicate successful receipt of information, a *NAK* packet to indicate that it's busy and didn't attempt to receive information, or a *STALL* packet to indicate that the transaction was correctly received but logically invalid in some way. When the host provides the handshake, it can send only an *ACK* packet.

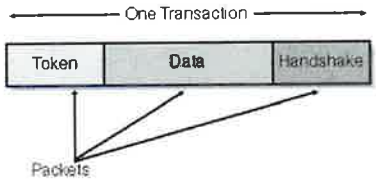


Figure 12-5. *Phases of a bus transaction.*

USB 2.0 uses an additional handshaking packet for output operations to bulk endpoints, called *NYET*. This is either the Russian word for *no* or a contraction of *not yet*. *NYET* is part of a flow-control scheme called *PING*, and it means that the endpoint cannot accept another full packet. The host is expected to defer sending additional output for a period of time governed by the endpoint's *bInterval* attribute (which was not used for bulk endpoints in USB 1.1). The purpose of the PING protocol and the NYET handshake is to avoid tying up the bus with data packets that are going to be NAK'ed by a busy device.

You'll notice that there's no handshake packet that means, "I found a transmission error in this transaction." Whoever is waiting for an acknowledgment is expected to realize that lack of acknowledgment implies an error and to retry the transaction. The USB designers believe that errors will be infrequent, by the way, which means that any occasional delay because of retries won't have a big effect on throughput.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 323

More About Device Addressing
The previous text says that *all* configured devices receive the electrical signals associated with every transaction. This is almost true, but a true renaissance programmer should know another detail. When a USB device first comes on line, it responds to a default address (which happens to be numerically 0, but you don't need to know that). Certain electrical signalling occurs to alert the host bus driver that a new device has arrived on the scene, whereupon the bus driver assigns a device address and sends a control transaction to tell "device number 0" what its real address is. From then on, the device answers only to the real address.

States of an Endpoint

In general, an endpoint can be in any of the states illustrated in Figure 12-6. In the Idle state, the endpoint is ready to process a new transaction initiated by the host. In the Busy state, the endpoint is busy processing a transaction and can't handle a new one. If the host tries to initiate a transaction to a busy endpoint (other than a control endpoint, as described in the next section), the device will respond with a *NAK* handshake packet to cause the host to retry later. Errors that the device detects in its own functionality (not including transmission errors) cause the device to send a *STALL* handshake packet for its current transaction and to enter the Stalled state. Control endpoints automatically unstall when they get a new transaction, but the host must send a clear feature control request to any other kind of endpoint before addressing another request to a stalled endpoint.

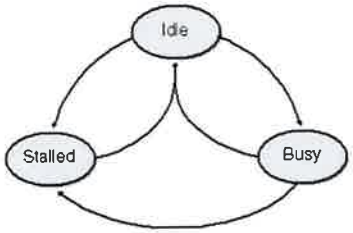


Figure 12-6. *States of an endpoint.*

Control Transfers

A *control transfer* conveys control information to or from a control endpoint on a device. For example, one part of the overall process by which the operating system configures a USB device is performing input control transfers to read various *descriptor* structures kept on board the device. Another part of the configuration process involves an output control transfer to establish one of the many possible configurations as current and to enable one or more interfaces. Control transfers are lossless in that the bus driver retries erroneous transfers up to three times before giving up and reporting an error status to upstream software. As indicated in Table 12-2, control endpoints must specify a maximum data transfer length of 8, 16, 32, or 64 bytes. An individual transaction can involve less data than the indicated maximum but not more.

Control transactions are a high priority in USB. A device isn't allowed to claim business as an excuse to avoid handling a control transaction. Moreover, the bus driver reserves up to 10 percent of each frame time (20 percent of each microframe for a high-speed device) for control transactions.

Every device has at least one control endpoint numbered 0 that responds to input and output control transactions. Strictly speaking, endpoints belong to interfaces configurations, but endpoint 0 is an exception in that it terminates the default control pipe for a device. Endpoint 0 is active even before the device receives its configuration and no matter which other endpoints (if any) are available. A device need not have additional control endpoints besides endpoint 0 (although the USB specification allows for the possibility) because endpoint 0 can service most control requests perfectly well. If you define a vendor-specific request that can't complete within the frame, however, you should create an additional control endpoint to forestall having your on-board handler preempted by a new transaction.

Each control transfer includes a setup stage, which can be followed by an optional data stage in which additional data moves to or from the device, and a status stage, in which the device either responds with an *ACK* packet or a *STALL* packet or doesn't respond at all. Figure 12-7 diagrams the setup stage, which includes a *SETUP* token, a data phase (not to be confused with the data stage of the transfer), and a handshake phase. The data and status stages of a control transfer follow the same protocol rules as a bulk transfer, as shown in the next subsection. Devices are required to accept control transfers at all times and can therefore not respond with *NAK* to indicate a busy endpoint. Sending an invalid request to a control endpoint elicits a *STALL* response, but the device automatically clears the stall condition when it receives the next *SETUP* packet. This special case of stalling is called *protocol stall* in the USB specification—see Section 8.5.3.4.

The *SETUP* token that prefaces a control transfer consists of 8 data bytes, as illustrated in Figure 12-8. In this and other data layout figures, I'm showing data bytes in the order in which they're transmitted over the USB wire, but I'm showing bits within individual bytes starting with the high-order bit. Bits are transmitted over the wire starting with the least-significant bit, but host software and device firmware typically work with data after the bits have been reversed. Intel computers and the USB bus protocols employ the little-endian data representation, in which the least-significant byte of a multibyte data item occupies the lowest address. The 8051 microprocessor used in several USB chip sets, including the Cypress Semiconductor chip set, is

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 324

actually a big-endian computer. Firmware must therefore take care to reverse data bytes appropriately.

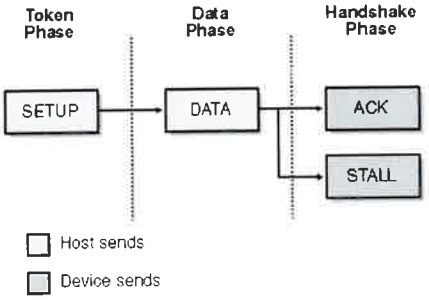


Figure 12-7. *Phases of the setup stage of a control transfer.*

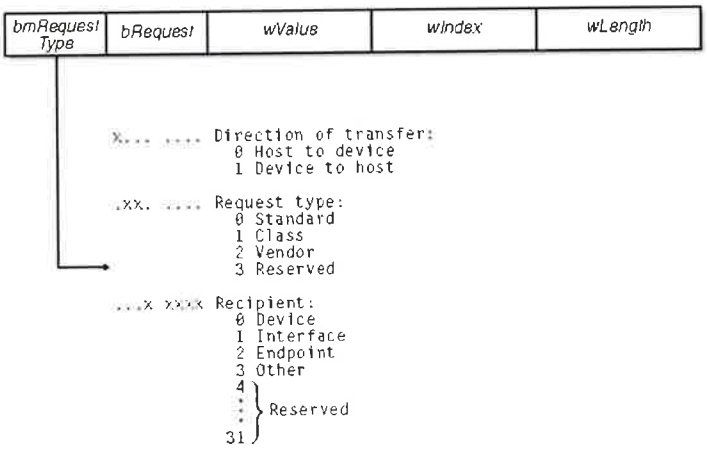


Figure 12-8. *Contents of a SETUP token.*

Request Code	Symbolic Name	Description	Possible Recipients
0	GET_STATUS	Gets status information	Any
1	CLEAR_FEATURE	Clears a two-state feature	Any
2		(Reserved)	
3	SET_FEATURE	Sets a two-state feature	Any
4		(Reserved)	
5	SET_ADDRESS	Sets device address	Device
6	GET_DESCRIPTOR	Gets device, configuration, or string descriptor	Device
7	SET_DESCRIPTOR	Sets a descriptor (optional)	Device
8	GET_CONFIGURATION	Gets current configuration index	Device
9	SET_CONFIGURATION	Sets new current configuration	Device
10	GET_INTERFACE	Gets current alternate setting index	Interface
11	SET_INTERFACE	Enables alternate setting	Interface
12	SYNCH_FRAME	Reports synchronization frame number	(Isochronous) Endpoint

Table 12-3. *Standard Device Requests*

Notice in Figure 12-8 that the first byte of a *SETUP* token indicates the direction of information flow, a request type, and the type of entity that is the target of the control transfer. The request types are *standard* (defined as part of the USB specification), *class* (defined by the USB working group responsible for a given class of device), and *vendor* (defined by the maker of the device). Control requests can be addressed to the device as a whole, to a specified interface, to a specified endpoint, or to some other vendor-specific entity on the device. The second byte of the *SETUP* token indicates which request of the type indicated in the first byte is being made. Table 12-3 lists the standard requests that are currently defined. For information about class-specific requests, consult the appropriate device class specification. (See the first URL I gave you at the beginning of this chapter for information on how to find these specifications.) Device manufacturers are free to define their own vendor-specific

IPR2024-00292
KASPERSKY EX 1008 Page 325

request codes. For example, Cypress Semiconductor uses the request code A0h to download firmware from the host.

NOTE
Remember that control requests that affect the state of a particular endpoint are sent to a control endpoint and not to the endpoint whose state is affected.

The remainder of the *SETUP* packet contains a *value* code whose meaning depends on which request is being made, an *index* value with similarly mutable meaning, and a *length* field that indicates how many bytes of data are to be transferred during the data stage of the control transaction. The index field contains the endpoint or interface number when a control request addresses an endpoint or an interface. A 0 value for the data length implies that this particular transaction has no data phase.

I'm not going to exhaustively describe all of the details of the various standard control requests; you should consult Section 9.4 of the USB specification for full information. I do want to briefly discuss the concept of a device *feature*, however. USB envisages that any of the addressable entities belonging to a device can have features that can be represented by the state of a single bit. Two such features are standardized for all devices, and one additional feature is standardized for controllers, hubs, and high-speed-capable functions.

The *DEVICE_REMOTE_WAKEUP* feature—a feature belonging to the device as a whole—indicates whether the device should use its ability (if any) to remotely wake up the computer when external events occur. Host software (specifically, the bus driver) enables or disables this feature by addressing a *SET_FEATURE* or *CLEAR_FEATURE* command to the device and specifying the value code 1 to designate the wake-up feature. The DDK uses the symbolic name *USB_FEATURE_REMOTE_WAKEUP* for this feature code.

CAUTION
Be sure your device really will signal a wake-up before turning on the *DEVICE_REMOTE_WAKEUP* bit in the configuration descriptor. The WHQL tests for USB devices specifically verify that the feature works if it's advertised.

The *ENDPOINT_HALT* feature—a feature belonging to an endpoint—indicates whether the endpoint is in the *functional stall* state. Host software can force an endpoint to stall by sending the endpoint a *SET_FEATURE* command with the value code 0 to designate *ENDPOINT_HALT*. The firmware that manages the endpoint might independently decide to stall too. Host software (once again, the bus driver) clears the stall condition by sending a *CLEAR_FEATURE* command with the value code 0. The DDK uses the symbolic name *USB_FEATURE_ENDPOINT_STALL* for this feature code.

Setting the *TEST_MODE* feature—a feature belonging to the device—places the device in a special test mode to facilitate compliance testing. Apart from hubs and controllers, only devices that can operate at high speed support this feature. You don't clear the *TEST_MODE* feature to exit from test mode; you power cycle the device instead.

The USB specification doesn't prescribe ranges of device or endpoint feature codes for vendor use. To avoid possible standardization issues later, you should avoid defining device-level or endpoint-level features. Instead, define your own vendor-type control transactions. Notwithstanding this advice, later in this chapter I'll show you a sample driver (FEATURE) that controls the 7-segment LED display on the Cypress Semiconductor development board. For purposes of that sample, I defined an interface-level feature numbered 42. (USB currently defines a few interface-level features for power management, so you wouldn't want to emulate my example except for learning about how features work.)

Bulk Transfers

A *bulk transfer* conveys up to 512 bytes of data to or from a bulk endpoint on a high-speed device or up to 64 bytes of data to or from a bulk endpoint on a full-speed device. Like control transfers, bulk transfers are lossless. Unlike control transfers, bulk transfers don't have any particular guaranteed latency. If the host has room left over in a frame or microframe after accommodating other bandwidth reservations, it will schedule pending bulk transfers.

Figure 12-9 illustrates the phases that make up a bulk transfer. The transfer begins with either an *IN* or an *OUT* token that addresses the device and the endpoint. In the case of an output transaction, a data phase follows in which data moves from the host to the device and then a handshake phase in which the device provides status feedback. If the endpoint is busy and unable to accept new data, it generates a *NAK* packet during the handshake phase—the host will retry the output transaction later. If the endpoint is stalled, it generates a *STALL* packet during the handshake phase—the host must later clear the halt condition before retrying the transmission. If the endpoint receives and processes the data correctly, it generates an *ACK* packet in the handshake phase. The only remaining case is the one in which the endpoint doesn't correctly receive the data for some reason and simply doesn't generate a handshake—the host will detect the absence of any acknowledgment and automatically retry up to three times.

Following the *IN* token that introduces an input bulk transfer, the device performs one of two operations. If it can, it sends data to the host, whereupon the host either generates an *ACK* handshake packet to indicate error-free receipt of the data or stays mute to indicate some sort of error. If the host detects an error, the absence of an *ACK* to the device causes the data to remain available—the host will retry the input operation later on. If the endpoint is busy or halted, however, the device generates a *NAK* or *STALL* handshake instead of sending data. The *NAK* indicates that the host should retry the input operation later, and the *STALL* requires the host to eventually send a clear feature command to reset the halt condition.

High-speed bulk output endpoints use a flow control protocol designed to minimize the amount of bus time wasted in fruitless

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 326

attempts to send data that the endpoint can't accept. In USB 1.1, an endpoint sends a *NAK* regarding a transfer it can't accept. In USB 2.0, however, a high-speed bulk endpoint descriptor's *bInterval* value indicates the *NAK rate*. If this value is 0, the endpoint never sends a *NAK*. Otherwise, it specifies the frequency (in microframes) with which the endpoint can send a *NAK* regarding an output transaction. The host can use a special *PING* packet to determine whether the endpoint is ready for output, and the endpoint can respond with either an *ACK* or a *NAK*. After a *NAK*, the host can then choose to wait *bInterval* packets before sending a *PING* again. After an *ACK*, the host sends an output packet. If the endpoint receives this normally, it replies with an *ACK* if it can accept another packet or with a *NYET* if it cannot.

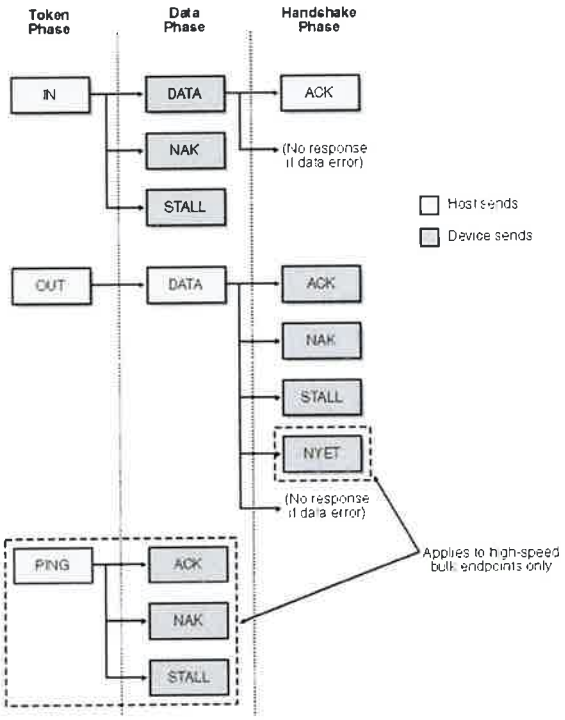


Figure 12-9. *Phases of a bulk or an interrupt transfer.*

Interrupt Transfers

An *interrupt transfer* is similar to a bulk transfer insofar as the operation of the bus and the device is concerned. It moves up to 1024 bytes (high speed), 64 bytes (full speed), or 8 bytes (low speed) of data losslessly to or from an interrupt endpoint. The main difference between interrupt and bulk transfers has to do with latency. An interrupt endpoint specifies a polling interval as described earlier in this chapter. The host reserves sufficient bandwidth to make sure of performing an *IN* or *OUT* transaction directed toward the endpoint at least as frequently as the polling interval.

NOTE
USB devices don't generate asynchronous interrupts: they always respond to a poll. You might need to know that the Microsoft host controller drivers effectively round the polling interval specified in an interrupt endpoint descriptor down to a power of 2 no greater than 32. For example, an endpoint that specifies a polling interval of 31 milliseconds will actually be polled every 16 milliseconds. A specified polling interval between 32 and 255 milliseconds results in an actual polling interval of 32 milliseconds.

Isochronous Transfers

An *isochronous transfer* moves up to 3072 data bytes to or from a high-speed endpoint during each microframe or up to 1023 data bytes to or from a full-speed endpoint during every bus frame. Because of the guaranteed periodicity of isochronous transfers, they're ideal for time-sensitive data such as audio signals. The guarantee of periodicity comes at a price, however: isochronous transfers that fail because of data corruption don't get retried automatically—in fact, with an isochronous transfer, "late" is just as bad as "wrong," so there's no point in doing in a retry.

An isochronous transaction consists of an *IN* or *OUT* token followed by a data phase in which data moves to or from the host. No handshake phase occurs because no errors are retried. See Figure 12-10.

IPR2024-00292
KASPERSKY EX 1008 Page 327

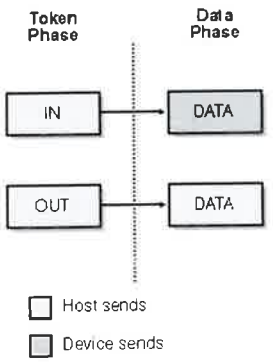


Figure 12-10. *Phases of an isochronous transfer.*

The host reserves up to 80 percent of the bus bandwidth (90 percent in the case of a USB 1.1 bus) for isochronous and interrupt transfers. In fact, system software needs to reserve bandwidth in advance to make sure that all active devices can be accommodated. In USB 1.1, the available bandwidth translates to approximately 1500 bytes per 1-millisecond frame, or roughly 1.5 maximum-size endpoints. In USB 2.0, the available bandwidth translates to about 7400 bytes per 125-microsecond microframe, or roughly 2.5 maximum-size endpoints. Taking account of the higher data rate, USB 2.0 has almost 20 times the isochronous capacity of USB 1.1.

12.1.4 Descriptors

USB devices maintain on-board data structures known as *descriptors* to allow for self-identification to host software. Table 12-4 lists the different descriptor types. Each descriptor begins with a 2-byte header containing the byte count of the entire descriptor (including the header) and a type code. As a matter of fact, if you ignore the special case of a string descriptor—concerning which, see "String Descriptors" a bit further on—the length of a descriptor is implied by its type because all descriptors of a given type have the same length. The explicit length is nonetheless present in the header to provide for future extensibility. Additional, type-specific data follows the fixed header.

In the remainder of this section, I'll describe the layout of each type of descriptor by using the data structures defined in the DDK (specifically, in USB100.H). The official rendition of this information is in Section 9.6 of the USB specification.

Descriptor Type	*Description*
Device	Describes an entire device
Device Qualifier	Device configuration information for the other speed of operation
Configuration	Describes one of the configurations of a device
Other Speed Configuration	Configuration descriptor for the other speed of operation
Interface	Describes one of the interfaces that's part of a configuration
Endpoint	Describes one of the endpoints belonging to an interface
String	Contains a human-readable Unicode string describing the device, a configuration, an interface, or an endpoint

Table 12-4. *Descriptor Types*

Device Descriptors

Each device has a single device descriptor that identifies the device to host software. The host uses a *GET_DESCRIPTOR* control transaction directed to endpoint 0 to read this descriptor. The device descriptor has the following definition in the DDK:

```
typedef struct  USB DEVICE DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  USHORT bcdUSB;
  UCHAR bDeviceClass;
  UCHAR bDeviceSubClass;
  UCHAR bDeviceProtocol;
  UCHAR bMaxPacketSize0;
  USHORT idVendor;
  USHORT idProduct;
  USHORT bcdDevice;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 328

```
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

The *bLength* field in a device descriptor will equal 18, and the *bDescriptorType* field will equal 1 to indicate that it's a device descriptor. The *bcdUSB* field contains a version code (in binary-coded decimal) indicating the version of the USB specification to which this descriptor conforms. New devices use the value 0x0200 here to indicate conformance with the 2.0 specification, whether they support high-speed, full-speed, or low-speed operation.

The values *bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* identify the type of device. Possible device class codes are defined by the USB specification and at the time of this writing include the codes listed in Table 12-5. Individual device class working groups within the USB committee define subclass and protocol codes for each device class. For example, the audio class has subclass codes for control, streaming, and MIDI streaming interfaces. And the mass storage class defines protocol codes for various methods of using endpoints for data transfer.

You can specify a class for an entire device or at the interface level, but in practice the device class, subclass, and protocol codes are often in an interface descriptor rather than in the device descriptor. (The device descriptor contains 0 for these codes in such cases.) USB also provides an escape valve for unusual types of devices in the form of the device class code 255. A vendor can use this type code to designate a nonstandard device for which the subclass and protocol codes provide the vendor-specific description. For example, a device built around the Cypress Semiconductor chip set comes on line with a device descriptor having class, subclass, and protocol codes all equal to 255. (The device has an extensive collection of endpoints and is also capable of accepting a vendor-specific control request to download firmware that will change the personality of the device to something else having its own [new] set of descriptors.)

The *bMaxPacketSize0* field of the device descriptor gives the maximum size of a data packet for a control transfer over endpoint 0. There isn't a separate endpoint descriptor for this endpoint (which every device has to implement), so this field is the only place where the number can be presented. Since this field is at offset 7 within the descriptor, the host can always read enough of the descriptor to retrieve this value, even if endpoint 0 is capable only of the minimum size transfer (8 bytes). Once the host knows how big endpoint 0 transfers can be, it can structure subsequent requests appropriately.

The *idVendor* and *idProduct* fields specify a vendor code and a vendor-specific product identifier for the device. *bcdDevice* specifies a release number (such as 0x0100 for version 1.0) for the device. These three fields determine which driver the host software will load when it detects the device. The USB organization assigns vendor codes, and each vendor assigns its own product codes.

Symbolic Name in DDK Header	Class Code	Description
USB_DEVICE_CLASS_RESERVED	0	Indicates that class codes are in the interface descriptors
USB_DEVICE_CLASS_AUDIO	1	Devices used to manipulate analog or digital audio, voice, and other sound-related data (but not including transport mechanisms)
USB_DEVICE_CLASS_COMMUNICATIONS	2	Telecommunications devices, such as modems, telephones, and answering machines
USB_DEVICE_CLASS_HUMAN_INTERFACE	3	Human interface devices (HID devices), such as keyboards and mice
USB_DEVICE_CLASS_MONITOR	4	Display monitors
USB_DEVICE_CLASS_PHYSICAL_INTERFACE	5	HID devices involving real-time physical feedback, such as force-feedback joysticks
USB_DEVICE_CLASS_POWER	6	HID devices that perform power management, such as batteries, chargers, and so on
USB_DEVICE_CLASS_PRINTER	7	Printers
USB_DEVICE_CLASS_STORAGE	8	Mass storage devices, such as disk and CD-ROM
USB_DEVICE_CLASS_HUB	9	USB hubs
	10	Communications data
	11	SmartCard reader
	12	Content security
	220	Diagnostic device
	224	Wireless controller (e.g., Bluetooth)
	254	Application-specific (firmware update, Infrared Data Association [IrDA] bridge)
USB_DEVICE_CLASS_VENDOR_SPECIFIC	255	Vendor-defined device class

Table 12-5. *USB Device Class Codes*

IPR2024-00292
KASPERSKY EX 1008 Page 329

Device Version Numbering
Microsoft strongly encourages vendors to increment the device version number for each revision of hardware or firmware to facilitate downstream software updates. Often a vendor releases a new version of hardware along with a revised driver. Also, hardware updates sometimes invalidate software patches or filter drivers that were present so as to address earlier hardware bugs. An automatic update mechanism might therefore have trouble updating a system if it can't determine which revision of the hardware it's working with.

The *iManufacturer*, *iProduct*, and *iSerialNumber* fields identify string descriptors that provide a human-readable description of the manufacturer, the product, and the unit serial number. These strings are optional, and a 0 value in one of these fields indicates the absence of the descriptor. If you put a serial number on a device, Microsoft recommends that you make it unique for each physical device. If you do so, and if your driver is digitally signed, the end user will be able to move the device around to different ports on the same computer and have it recognized as being the same device.

Lastly, the *bNumConfigurations* field indicates how many configurations the device is capable of. Microsoft drivers work only with the first configuration of a device. I'll explain later, in "Configuration," what you might do for a device that has multiple configurations.

Device Qualifier Descriptor

High-speed-capable USB 2.0 devices can operate at either high speed or full speed. They deliver a device descriptor corresponding to the speed at which they are actually operating. They also deliver a Device Qualifier Descriptor that describes the device-level information that might be different if the device were operating at the other speed:

```
typedef struct  USB_DEVICE_QUALIFIER_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  USHORT bcdUSB;
  UCHAR bDeviceClass;
  UCHAR bDeviceSubClass;
  UCHAR bDeviceProtocol;
  UCHAR bMaxPacketSize0;
  UCHAR bNumConfigurations;
  UCHAR bReserved;
  } USB_DEVICE_QUALIFIER_DESCRIPTOR, *PUSB_DEVICE_QUALIFIER_DESCRIPTOR;
```

Apart from *bLength* (10) and *bDescriptorType* (6), all of these fields have exactly the same meaning as in a device descriptor. This descriptor doesn't repeat the vendor, product, device, manufacturer, product, and serial number fields of the device descriptor, which is constant for a device for all supported speeds.

Configuration Descriptors

Each device has one or more configuration descriptors that describe the various configurations of which the device is capable. System software reads a configuration descriptor by performing a *GET_DESCRIPTOR* control transaction addressed to endpoint 0. The DDK defines the configuration descriptor structure as follows:

```
typedef struct  USB_CONFIGURATION_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  USHORT wTotalLength;
  UCHAR bNumInterfaces;
  UCHAR bConfigurationValue;
  UCHAR iConfiguration;
  UCHAR bmAttributes;
  UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

The *bLength* and *bDescriptorType* fields will be 9 and 2, respectively, to indicate a configuration descriptor 9 bytes in length. The *wTotalLength* field contains the total length of this configuration descriptor plus the interface and endpoint descriptors that are part of the configuration. In general, the host performs one *GET_DESCRIPTOR* request to retrieve the 9-byte configuration descriptor proper and then another *GET_DESCRIPTOR* request specifying this total length. The second request, therefore, transfers the *grand unified* descriptor. (It's impossible to retrieve interface and endpoint descriptors except as part of a configuration descriptor.)

The *bNumInterfaces* field indicates how many interfaces are part of the configuration. The count includes just the interfaces themselves, not each alternate setting of an interface. The purpose of this field is to allow for multifunction devices such as keyboards that have embedded locator (mouse and the like) functionality.

The *bConfigurationValue* field is an index that identifies the configuration. You use this value in a *SET_CONFIGURATION*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 330

control request to select the configuration. The first configuration descriptor for a device has a nonzero value here. (Selecting configuration 0 puts the device in an unconfigured state in which only endpoint 0 is active.)

The *iConfiguration* field is an optional string descriptor index pointing to a Unicode description of the configuration. The value 0 indicates the absence of a string description.

The *bmAttributes* byte contains a bit mask describing power and perhaps other characteristics of this configuration. See Table 12-6. The unmentioned bits are reserved for future standardization. A configuration supporting remote wake-up will have the remote wake-up attribute set. The high-order 2 bits interact with the *MaxPower* field of the configuration descriptor to describe the power characteristics of the configuration. Basically, every configuration sets the high-order bit (which used to mean the device was powered from the bus) and also sets *MaxPower* to the maximum number of two milliamp power units that it will draw from the bus. A configuration that uses some local power will also set the self-powered attribute bit.

Bit Mask	Symbolic Name	Description
80h	USB_CONFIG_BUS_POWERED	Obsolete—should always be set to 1
40h	USB_CONFIG_SELF_POWERED	Configuration is self-powered
20h	USB_CONFIG_REMOTE_WAKEUP	Configuration has a remote wake-up feature

Table 12-6. *Configuration Attribute Bits*

Other Speed Configuration Descriptors

An Other Speed Configuration Descriptor is identical to the corresponding Configuration Descriptor except that it uses a different type code—7. A high-speed-capable device supplies this descriptor to describe a configuration as it would appear if the device were operating at the other speed of which it is capable.

Interface Descriptors

Each configuration has one or more interface descriptors that describe the interface or interfaces that provide device functionality. System software can fetch an interface descriptor only as part of a *GET_DESCRIPTOR* control request that retrieves the entire configuration descriptor of which the interface descriptor is a part. The DDK defines the interface descriptor structure as follows:

```
typedef struct  USB_INTERFACE_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bInterfaceNumber;
  UCHAR bAlternateSetting;
  UCHAR bNumEndpoints;
  UCHAR bInterfaceClass;
  UCHAR bInterfaceSubClass;
  UCHAR bInterfaceProtocol;
  UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

The *bLength* and *bDescriptorType* fields will be 9 and 4, respectively, to indicate an interface descriptor 9 bytes in length. *bInterfaceNumber* and *bAlternateSetting* are index values that can be used in a *SET_INTERFACE* control transaction to specify activation of the interface. You should number the interfaces within a configuration, and the alternate settings within an interface, starting with 0 because system software and firmware will often treat an interface number as an index into an array.

The *bNumEndpoints* field indicates how many endpoints—other than 0, which is assumed to always be present—are part of the interface.

The *bInterfaceClass*, *bInterfaceSubClass*, and *bInterfaceProtocol* fields describe the functionality provided by the interface. A nonzero class code should be one of the device class codes I discussed earlier, in which case the subclass and protocol codes will have the same meaning as well. Zero values in these fields are not allowed at the present time—0 is reserved for future standardization.

Finally, *iInterface* is the index of a string descriptor containing a Unicode description of the interface. The value 0 indicates that no string is present. You'll want to supply interface string descriptors for a composite device since the parent driver will use those strings when enumerating the interfaces as child devices.

Endpoint Descriptors

Each interface has zero or more endpoint descriptors that describe the endpoint or endpoints that handle transactions with the host. System software can fetch an endpoint descriptor only as part of a *GET_DESCRIPTOR* control request that retrieves the entire configuration descriptor of which the endpoint descriptor is a part. The DDK defines the endpoint descriptor structure as follows:

IPR2024-00292
KASPERSKY EX 1008 Page 331

```
typedef struct  USB_ENDPOINT_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  UCHAR bEndpointAddress;
  UCHAR bmAttributes;
  USHORT wMaxPacketSize;
  UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

The *bLength* and *bDescriptorType* fields will be 7 and 5, respectively, to indicate an endpoint descriptor of length 7 bytes. *bEndpointAddress* encodes the directionality and number of the endpoint, as illustrated in Figure 12-11. For example, the address value 0x82 denotes an IN endpoint numbered 2, and the address 0x02 denotes an OUT endpoint that's also numbered 2. Except for endpoint 0, the USB specification allows you to have two different endpoints that share the same number but perform transfers in the opposite direction. Many USB chip sets don't support this overloading of endpoint number, though.

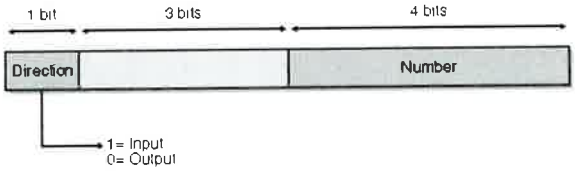


Figure 12-11. *Bit assignments within an endpoint descriptor's address field.*

Figure 12-12 illustrates the layout of bits within an endpoint descriptor's *bmAttributes* field. Bits 0 through 1 define the endpoint type, corresponding to the data transfer types listed in Table 12-1. Bits 2 through 5 define additional attributes for isochronous endpoints.

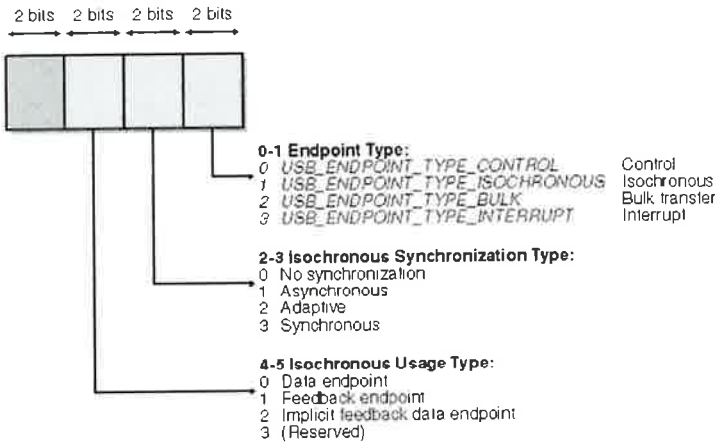


Figure 12-12. *Bit assignments within an endpoint descriptor's attributes field.*

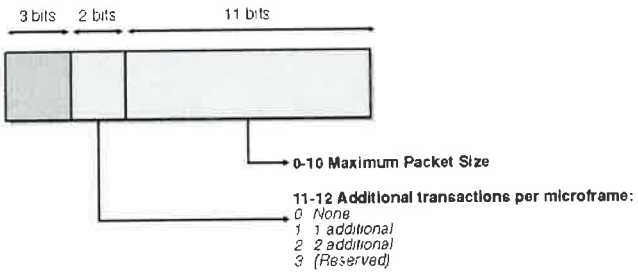


Figure 12-13. *Bit assignments within an endpoint descriptor's packet size field.*

The *wMaxPacketSize* value indicates the largest amount of data the endpoint can transfer during one transaction. Consult Figure 12-13 for a bit layout of this field. Table 12-2 lists the possible values for this field for each type of endpoint. For example, a control or bulk endpoint on a full-speed device specifies the value 8, 16, 32, or 64. High-speed interrupt and isochronous endpoints can perform 1, 2, or 3 transactions in a microframe, as indicated by the coding in bits 11 through 12 of this field. The additional transactions allow additional data to be transferred in a microframe. See Table 12-7.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 332

Number of Additional Transactions	Allowable wMaxPacketSize Values (Bits 0-10)	Total Amount of Data per Microframe
0	1-1024	≤1024
1	513-1024	≤ 2048
2	683-1024	≤ 3072

Table 12-7. *Effect of Additional Transactions on Packet and Transfer Sizes*

Interrupt and isochronous endpoint descriptors specify a polling interval in the *bInterval* field. As discussed earlier in this chapter, this number indicates how often the host should poll the endpoint for a possible data transfer. A high-speed bulk endpoint descriptor specifies its *NAK* rate in the *bInterval* field.

String Descriptors

A device, configuration, or endpoint descriptor contains optional string indexes that identify human-readable strings. The strings themselves are stored on the device in Unicode in the form of USB string descriptors. System software can read a string descriptor by addressing a *GET_DESCRIPTOR* control request to endpoint 0. The DDK declares the string descriptor structure as follows:

```
typedef struct  USB_STRING_DESCRIPTOR {
  UCHAR bLength;
  UCHAR bDescriptorType;
  WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;
```

The *bLength* value is variable, depending on how long the string data is. The *bDescriptorType* field will be 3 to indicate that this is a string descriptor. The *bString* data contains the string data itself without a null terminator.

USB devices can support strings in multiple languages. String number 0 is an array of supported language identifiers rather than a character string. (The string index 0 used in another descriptor denotes the absence of a string reference. Thus, index number 0 is available for this special use.) The language identifiers are of the same LANGID type that Win32 programs use. For example, 0x0409 is the code for American English. The USB specification prescribes that a device should return an error when asked for a string descriptor for a language that the device doesn't advertise supporting, so you should read the string-zero array before issuing requests for string descriptors. Consult Section 9.6.7 of the USB specification for more information about language identifiers.

Other Descriptors

USB is an evolving specification, and I can present only a snapshot of its evolution at the time of writing. Many USB class specifications define one or more class-specific descriptors that appear within the block of data returned by a request to read a configuration descriptor. Discussing these class-specific descriptors is beyond the scope of this work, except to mention that class-specific descriptors will follow the interface descriptor to which they apply and precede the endpoint descriptors for that interface.

12.2 Working with the Bus Driver

In contrast with drivers for devices that attach to traditional PC buses such as Peripheral Component Interconnect (PCI), a USB device driver never talks directly to its hardware. Instead, it creates an instance of the data structure known as the USB request block, which it then submits to the parent driver.

You submit USB request blocks (URBs) to the parent driver using an IRP with the major function code *IRP_MJ_INTERNAL_DEVICE_CONTROL*. In some situations, you can directly call a function using the parent driver's direct-call interface. The parent driver in turn schedules bus time in some frame or another to carry out the operation encoded in the URB.

In this section, I'll describe the mechanics of working with the parent driver to carry out the typical operations a USB function driver performs. I'll first describe how to build and submit a URB. Then I'll discuss the mechanics of configuring and reconfiguring your device. Finally I'll outline how your driver can manage each of the four types of communication pipes.

12.2.1 Initiating Requests

To create a URB, you allocate memory for the *URB* structure and invoke an initialization routine to fill in the appropriate fields for the type of request you're about to send. Suppose, for example, that you were beginning to configure your device in response to an *IRP_MN_START_DEVICE* request. One of your first tasks might be to read the device descriptor. You might use the following snippet of code to accomplish this task:

```
USB_DEVICE_DESCRIPTOR dd;
```


IPR2024-00292
KASPERSKY EX 1008 Page 333

```
URB urb;
UsbBuildGetDescriptorRequest(&urb,
  sizeof( URB CONTROL DESCRIPTOR REQUEST),
  USB DEVICE DESCRIPTOR TYPE, 0, 0, &dd, NULL,
  sizeof(dd), NULL);
```

We first declare a local variable (named *urb*) to hold a *URB* data structure. The *URB* is declared (in USBDI.H) as a union of several substructures, one for each of the requests you might want to make of a USB device. We're going to be using the *UrbControlDescriptorRequest* substructure of the *URB* union, which is declared as an instance of *struct _URB_CONTROL_DESCRIPTOR_REQUEST*. Using an automatic variable like this is fine if you know the stack has enough room to hold the largest possible *URB* and if you'll await completion of the *URB* before allowing the variable to pass out of scope.

You can, of course, dynamically allocate the memory for a *URB* from the heap if you want:

```
PURB urb = (PURB) ExAllocatePool(NonPagedPool,
  sizeof( URB CONTROL DESCRIPTOR REQUEST));
if (!urb)
  return STATUS INSUFFICIENT RESOURCES;
UsbBuildGetDescriptorRequest(urb, ...);
:
ExFreePool(urb);
```

UsbBuildGetDescriptorRequest is documented like a normal service routine, but it's actually a macro (declared in USBDLIB.H) that generates inline statements to initialize the fields of the get descriptor request substructure. The DDK headers define one of these macros for most types of URBs you might want to build. See Table 12-8. As is true of preprocessor macros in general, you should avoid using expressions that have side effects in the arguments to this macro.

Helper Macro	Type of Transaction
UsbBuildInterruptOrBulkTransferRequest	Input or output to an interrupt or bulk endpoint
UsbBuildGetDescriptorRequest	*GET_DESCRIPTOR* control request for endpoint 0
UsbBuildGetStatusRequest	*GET_STATUS* request for a device, an interface, or an endpoint
UsbBuildFeatureRequest	*SET_FEATURE or CLEAR_FEATURE* request for a device, an interface, or an endpoint
UsbBuildSelectConfigurationRequest	*SET_CONFIGURATION*
UsbBuildSelectInterfaceRequest	*SET_INTERFACE*
UsbBuildVendorRequest	Any vendor-defined control request

Table 12-8. *Helper Macros for Building URBs*

In the preceding code fragment, we specify that we want to retrieve the device descriptor information in a local variable (*dd*) whose address and length we supply. URBs that involve data transfer allow you to specify a nonpaged data buffer in either of two ways. You can specify the virtual address and length of the buffer, as I did in the fragment. Alternatively, you can supply a memory descriptor list (MDL) for which you've already done the probe-and-lock step by calling *MmProbeAndLockPages*.

> **More About URBs**
> Internally, the bus driver always uses an MDL to describe data buffers. If you specify a buffer address, the parent driver creates the MDL itself. If you happen to already have an MDL, it would be counterproductive to call *MmGetSystemAddressForMdlSafe* and pass the resulting virtual address to the parent driver: the parent driver will turn around and create *another* MDL to describe the same buffer!

The URB also has a chaining field named *Urblink* that the parent driver uses internally to submit a series of URBs all at once to the host controller driver. The various macro functions for initializing URBs also have an argument in which you can theoretically supply a value for this linking field. You and I should *always* supply *NULL* because the concept of linked URBs hasn't been fully implemented—trying to link data transfer URBs will lead to system crashes, in fact.

Sending a URB

Having created a URB, you need to create and send an internal IOCTL request to the parent driver, which is sitting somewhere lower in the driver hierarchy for your device. In many cases, you'll want to wait for the device's answer, and you'll use a helper routine like this one:

```
NTSTATUS SendAwaitUrb(PDEVICE OBJECT fdo, PURB urb)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fdo->DeviceExtension;
  KEVENT event;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 334

```
KeInitializeEvent(&event, NotificationEvent, FALSE);
IO STATUS BLOCK iostatus;
PIRP Irp = IoBuildDeviceIoControlRequest
  (IOCTL INTERNAL USB SUBMIT URB, pdx->LowerDeviceObject,
  NULL, 0, NULL, 0, TRUE, &event, &iostatus);
PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
stack->Parameters.Others.Argument1 = (PVOID) urb;
NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
if (status  == STATUS PENDING)
  {
  KeWaitForSingleObject(&event, Executive, KernelMode,
    FALSE, NULL);
  status = iostatus.Status;
  }
return status;
}
```

This is simply an example of creating and sending a synchronous IRP to another driver (scenario 6 from Chapter 5). The only wrinkle is the precise way the URB "letter" is stuffed into the *INTERNAL_DEVICE_CONTROL* "envelope": by setting the stack *Parameters.Others.Argument1* field to point to the URB.

NOTE
It bears emphasizing that drivers package URBs into normal IRPs with the major function code *IRP_MJ_INTERNAL_DEVICE_CONTROL*. To provide for an upper filter driver to send its own URBs, every driver for a USB device should have a dispatch function that passes this IRP down to the next layer.

Status Returns from URBs

When you submit a URB to the USB bus driver, you eventually receive back an *NTSTATUS* code that describes the result of the operation. Internally, the bus driver uses another set of status codes with the *typedef* name *USBD_STATUS*. These codes are not *NTSTATUS* codes.

When the parent driver completes a URB, it sets the URB's *UrbHeader.Status* field to one of these *USBD_STATUS* values. You can examine this value in your driver to glean more information about how your URB fared. The *URB_STATUS* macro in the DDK simplifies accessing:

```
NTSTATUS status = SendAwaitUrb(fdo, &urb);
USBD STATUS ustatus = URB STATUS(&urb);
:
```

There's no particular protocol for preserving this status and passing it back to an application, however. You're pretty much free to do what you will with it.

12.2.2 Configuration

The USB bus driver automatically detects attachment of a new USB device. It then reads the device descriptor structure to determine what sort of device has suddenly appeared. The vendor and product identifier fields of the descriptor, together with other descriptors, determine which driver needs to be loaded.

The Configuration Manager calls the driver's *AddDevice* function in the normal way. *AddDevice* does all the tasks you've already heard about: it creates a device object, links the device object into the driver hierarchy, and so on. The Configuration Manager eventually sends the driver an *IRP_MN_START_DEVICE* Plug and Play request. Back in Chapter 6, I showed you how to handle that request by calling a helper function named *StartDevice* with arguments describing the translated and untranslated resource assignments for the device. One piece of good news is that you needn't worry about I/O resources at all in a USB driver because you have none. So you can write a *StartDevice* helper function with the following skeletal form:

```
NTSTATUS StartDevice(PDEVICE OBJECT fdo)
  {
  PDEVICE EXTENSION pdx =
    (PDEVICE EXTENSION) fdo->DeviceExtension;
  <configure device>
  return STATUS SUCCESS;
  }
```

I glibly said *configure device* where you'll write rather a lot of code to configure the hardware. But as I said, you needn't concern yourself with I/O ports, interrupts, direct memory access (DMA) adapter objects, or any of the other resource-oriented elements I described in Chapter 7.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 335

Where's the Driver?
I'll discuss the mechanics of installing WDM drivers in Chapter 15. It will help to understand some of those details right now, however. Let's suppose that your device has the vendor ID 0x0547 and the product ID 0x102A. I've borrowed the vendor ID belonging to Cypress Semiconductor (with their permission) for purposes of this illustration. I'm using the product ID for the USB42 sample (the Answer Device) that you'll find in the companion content.

USB describes many methods for the operating system to locate a device driver (or set of drivers) based on the device, configuration, and interface descriptors on a device. See Universal Serial Bus Common Class Specification (Rev. 1.0, December 16, 1997), Section 3.10. My samples all rely on the second-highest priority method, whereby the vendor and product identifiers alone determine the driver.

Confronted with a device having the vendor and product identifiers I just mentioned, the PnP Manager will look for a registry entry that contains information about a device named *USB\VID_0547&PID_102A*. If no such entry exists in the registry, the PnP Manager will trigger the new hardware wizard to locate an INF file describing such a device. The wizard might prompt the end user for a disk, or it might find the INF file already present on the computer. The wizard will then install the driver and populate the registry. Once the PnP Manager locates the registry entries, it can dynamically load the driver. That's where we come in.

The executive overview of what you need to accomplish in *StartDevice* is as follows. First you'll select a configuration for the device. If your device is like most devices, it has just one configuration. Once you select the configuration, you choose one or more of the interfaces that are part of that configuration. It's not uncommon for a device to support multiple interfaces, by the way. Having chosen a configuration and a set of interfaces, you send a select configuration URB to the bus driver. The bus driver in turn issues commands to the device to enable the configuration and interfaces. The bus driver creates *pipes* that allow you to communicate with the endpoints in the selected interfaces and provides handles by which you can access the pipes. It also creates handles for the configuration and the interfaces. You extract the handles from the completed URB and save them for future use. That accomplished, you're done with the configuration process.

Composite Devices
If your device has one configuration and multiple interfaces, the Microsoft *generic parent* driver will handle it automatically as a *composite*, or multifunction, device. You supply function drivers for each of the interfaces on the device by using INF files that specify the subfunction index along with a vendor and product ID. The generic parent driver creates a physical device object (PDO) for each interface, whereupon the PnP Manager loads the separate function drivers you've provided. When one of these function drivers reads a configuration descriptor, the generic parent driver provides an edited version of the descriptor that describes just one interface.

Refer to Chapter 15 for more information about the possible forms of device identifier in an INF file.

Reading a Configuration Descriptor

It's best to think of a fixed-size configuration descriptor as the header for a variable-length structure that describes a configuration, all its interfaces, and all the interfaces' endpoints. See Figure 12-14.

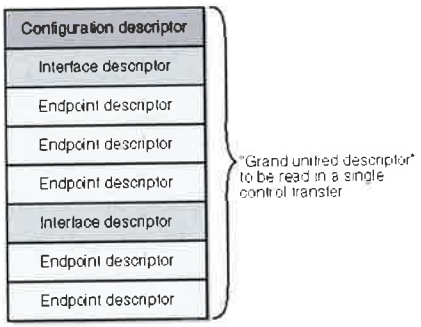


Figure 12-14. *Structure of a configuration descriptor.*

You must read the entire variable-length structure into a contiguous area of memory because the hardware won't allow you to directly access the interface and endpoint descriptors. Unfortunately, you don't initially know how long the combined structure is. The following fragment of code shows how you can use two URBs to read a configuration descriptor:

```
ULONG iconfig = 0;
URB urb;
USB_CONFIGURATION_DESCRIPTOR tcd;
UsbBuildGetDescriptorRequest(&urb,
  sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 336

```
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    iconfig, 0, &tcd, NULL, sizeof(tcd), NULL);
  SendAwaitUrb(fdo, &urb);
  ULONG size = tcd.wTotalLength;
  PUSB_CONFIGURATION_DESCRIPTOR pcd =
    (PUSB_CONFIGURATION_DESCRIPTOR) ExAllocatePool(
    NonPagedPool, size);
  UsbBuildGetDescriptorRequest(&urb,
    sizeof( URB_CONTROL_DESCRIPTOR_REQUEST),
    USB_CONFIGURATION_DESCRIPTOR_TYPE,
    iconfig, 0, pcd, NULL, size, NULL);
  SendAwaitUrb(fdo, &urb);
  :
  ExFreePool(pcd);
```

In this fragment, we issue one URB to read a configuration descriptor—I specified an index of 0 to get the first one—into a temporary descriptor area named *tcd*. This descriptor contains the length (*wTotalLength*) of the combined structure, which includes configuration, interface, and endpoint descriptors. We allocate that much memory and issue a second URB to read the entire descriptor. At the end of the process, the *pcd* variable points to the whole shebang. (Don't leave out the error checking as I just did—see the code samples in the companion content for examples of how to handle the many errors that might arise in this short sequence.)

TIP
You read configuration descriptors using a zero-based index. When the bus driver eventually issues a control transaction to enable that configuration, it uses the *bConfigurationValue* from the configuration descriptor. Usually, there's just one configuration descriptor numbered *1* that you read using index *0*. Are there any readers who aren't dizzy yet?

If your device has a single configuration, go ahead to the next step using the descriptor set you've just read. Otherwise, you'll need to enumerate the configurations (that is, step the *iconfig* variable from 0 to 1 less than the *bNumConfigurations* value in the device descriptor) and apply some sort of algorithm to pick from among them.

Selecting the Configuration

You eventually have to select a configuration by sending a series of control commands to the device to set the configuration and enable the desired interfaces. We'll be using a function named *USBD_CreateConfigurationRequestEx* to create the URB for this series of commands. One of its arguments is an array of pointers to descriptors for the interfaces you intend to enable. Your next step in configuration after settling on the configuration you want to use, therefore, is to prepare this array.

Reading a String Descriptor
For reporting or other purposes, you might want to retrieve some of the string descriptors that your device might provide. In the USB42 sample, for example, the device contains English-language descriptors for the vendor, product, and serial number as well as for the single configuration and interface supported by the device. I wrote the following helper function for reading string descriptors:

```
NTSTATUS GetStringDescriptor(PDEVICE_OBJECT fdo, UCHAR istring,
  PUNICODE_STRING s)
  {
  NTSTATUS status;
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
  URB urb;

  UCHAR data[256];

  if (!pdx->langid)
    {
    UsbBuildGetDescriptorRequest(&urb,
      sizeof( URB_CONTROL_DESCRIPTOR_REQUEST),
      USB_STRING_DESCRIPTOR_TYPE,
      0, 0, data, NULL, sizeof(data), NULL);
    status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status))
      return status;
    pdx->langid = *(LANGID*)(data + 2);
    }
```


IPR2024-00292
KASPERSKY EX 1008 Page 337

```
UsbBuildGetDescriptorRequest(&urb,
  sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST),
  USB_STRING_DESCRIPTOR_TYPE,
  istring, pdx->langid, data, NULL, sizeof(data), NULL);
status = SendAwaitUrb(fdo, &urb);
if (!NT_SUCCESS(status))
  return status;

ULONG nchars = (data[0] - sizeof(WCHAR)) / sizeof(WCHAR);
if (nchars > 257)
  nchars = 257;
PWSTR p = (PWSTR) ExAllocatePool(PagedPool,
  (nchars + 1) * sizeof(WCHAR));
if (!p)
  return STATUS_INSUFFICIENT_RESOURCES;
RtlCopyMemory(p, data + 2, nchars * sizeof(WCHAR));
p[nchars] = 0;
s->Length = (USHORT) (sizeof(WCHAR) * nchars);
s->MaximumLength = (USHORT) ((sizeof(WCHAR) * nchars)
  + sizeof(WCHAR));
s->Buffer = p;

return STATUS_SUCCESS;
}
```

The new and interesting part of this function—given that you already know a lot about kernel-mode programming if you've been reading this book sequentially—is the initialization of the URB to fetch a string descriptor. In addition to supplying the index of the string we want to get, we also supply a standard *LANGID* language identifier. This is the same kind of language identifier that you use in a Win32 application. As I mentioned earlier, devices can provide strings in multiple languages, and string descriptor 0 contains a list of the supported language identifiers. To be sure to always ask for a supported language, I read string 0 the first time this routine executes and arbitrarily choose the first language as the one to ask for. In the actual sample drivers, the identifier will always be 0x0409, which identifies American English. The parent driver passes this language identifier along with the string index as a parameter for the get descriptor request it sends to the device. The device itself is responsible for deciding which string to return.

The output from my *GetStringDescriptor* function is a *UNICODE_STRING* that you use in the normal way. You would eventually call *RtlFreeUnicodeString* to release the string buffer.

I used *GetStringDescriptor* in the USB42 sample to generate extra debugging output about the device. For example, *StartDevice* contains code similar to this fragment:

```
UNICODE_STRING sd;
if (pcd->iConfiguration
  && NT_SUCCESS(GetStringDescriptor(fdo,
  pcd->iConfiguration, &sd)))
  {
  KdPrint(("USB42 - Selecting configuration named %ws\n",
    sd.Buffer));
  RtlFreeUnicodeString(&sd);
  }
```

I actually used a macro so that I wouldn't have to type this same code a bunch of times, but you get the idea.

Recall that when we read the configuration descriptor, we also read all of its interface descriptors into adjacent memory. This memory therefore contains a series of descriptors: a configuration descriptor, an interface descriptor followed by all of its endpoints, another interface descriptor followed by all of *its* endpoints, and so on. One way of choosing interfaces is to parse through this collection of descriptors and remember the addresses of the interface descriptors you're interested in. The bus driver provides a routine named *USBD_ParseConfigurationDescriptorEx* to simplify that task:

```
PUSB_INTERFACE_DESCRIPTOR pid;
pid = USBD_ParseConfigurationDescriptorEx(pcd, StartPosition,
  InterfaceNumber, AlternateSetting, InterfaceClass,
  InterfaceSubclass, InterfaceProtocol);
```

In this function, *pcd* is the address of the grand unified configuration descriptor. *StartPosition* is either the address of the configuration descriptor (the first time you make this call) or the address of a descriptor at which you want to begin searching. The remaining parameters specify criteria for a descriptor search. The value -1 indicates that you don't want the corresponding criterion to be employed in the search. You can look for the next interface descriptor that has zero or more of these attributes:

- The given InterfaceNumber

IPR2024-00292
KASPERSKY EX 1008 Page 338

- The given AlternateSetting index
- The given InterfaceClass index
- The given InterfaceSubclass index
- The given InterfaceProtocol index

When *USBD_ParseConfigurationDescriptorEx* returns an interface descriptor to you, you save it as the *InterfaceDescriptor* member of an element in an array of *USBD_INTERFACE_LIST_ENTRY* structures, and then you advance past the interface descriptor so that you can parse the next one. The array of interface list entries will be one of the parameters to the eventual call to *USBD_CreateConfigurationRequestEx*, so I need to say a little more about it. Each entry in the array is an instance of the following structure:

```
typedef struct  USBD INTERFACE LIST ENTRY {
    PUSB INTERFACE DESCRIPTOR InterfaceDescriptor;
    PUSBD INTERFACE INFORMATION Interface;
} USBD_INTERFACE_LIST_ENTRY, *PUSBD_INTERFACE_LIST_ENTRY;
```

When you initialize an entry in the array, you set the *InterfaceDescriptor* member equal to the address of an interface descriptor that you want to enable and you set the *Interface* member to *NULL*. You define one entry for each interface, and then you add another entry whose *InterfaceDescriptor* is *NULL* to mark the end. For example, in my USB42 sample, I know in advance that only one interface exists, so I use the following code to create the interface list:

```
PUSB INTERFACE DESCRIPTOR pid =
  USBD ParseConfigurationDescriptorEx(pcd, pcd, -1, -1,
  -1, -1, -1);
USBD INTERFACE LIST ENTRY interfaces[2] = {
  {pid, NULL},
  {NULL, NULL},
  };
```

That is, I parse the configuration descriptor to locate the first (and only) interface descriptor. Then I define a two-element array to describe that one interface.

If you need to enable more than one interface because you're providing your own composite device support, you'll repeat the parsing call in a loop. For example:

```
ULONG size = (pcd->bNumInterfaces + 1) *
    sizeof(USBD INTERFACE LIST ENTRY);
PUSBD INTERFACE LIST ENTRY interfaces =
    (PUSBD INTERFACE LIST ENTRY) ExAllocatePool(NonPagedPool,
    size);
RtlZeroMemory(interfaces, size);
ULONG i = 0;
PUSB INTERFACE DESCRIPTOR pid =
    (PUSB INTERFACE DESCRIPTOR) pcd;
while ((pid = USBD ParseConfigurationDescriptorEx(pcd,
    pid, ...)))
    interfaces[i++].InterfaceDescriptor = pid++;
```

1. We first allocate memory to hold as many interface list entries as there are interfaces in this configuration, plus one. We zero the entire array. Wherever we leave off in filling the array during the subsequent loop, the next entry will be *NULL* to mark the end of the array.
2. The parsing call includes whatever criteria are relevant to your device. In the first iteration of the loop, *pid* points to the configuration descriptor. In later iterations, it points just past the interface descriptor returned by the preceding call.
3. Here we initialize the pointer to an interface descriptor. The postincrement of *i* causes the next iteration to initialize the next element in the array. The postincrement of *pid* advances past the current interface descriptor so that the next iteration parses the next interface. (If you call *USBD_ParseConfigurationDescriptorEx* with the second argument pointing to an interface descriptor that meets your criteria, you'll get back a pointer to that same descriptor. If you don't advance past that descriptor before making the next call, you're doomed to repeat the loop forever.)

The next step in the configuration process is to create a URB that we'll submit—soon, I promise—to configure the device:

IPR2024-00292
KASPERSKY EX 1008 Page 339

```
PURB selurb = USBD_CreateConfigurationRequestEx(pcd, interfaces);
```

In addition to creating a URB (to which *selurb* points at this moment), *USBD_CreateConfigurationRequestEx* also initializes the *Interface* members of your *USBD_INTERFACE_LIST* entries to point to *USBD_INTERFACE_INFORMATION* structures. These information structures are physically located in the same memory block as the URB and will, therefore, be released back to the heap when you eventually call *ExFreePool* to return the URB. An interface information structure has the following declaration:

```
typedef struct  USBD INTERFACE INFORMATION {
  USHORT Length;
  UCHAR InterfaceNumber;
  UCHAR AlternateSetting;
  UCHAR Class;
  UCHAR SubClass;
  UCHAR Protocol;
  UCHAR Reserved;
  USBD INTERFACE HANDLE InterfaceHandle;
  ULONG NumberOfPipes;
  USBD PIPE INFORMATION Pipes[1];
  } USBD_INTERFACE_INFORMATION, *PUSBD_INTERFACE_INFORMATION;
```

The array of pipe information structures is what we're really interested in at this point since the other fields of the structure will be filled in by the parent driver when we submit this URB. Each of them looks like this:

```
typedef struct  USBD PIPE INFORMATION {
  USHORT MaximumPacketSize;
  UCHAR EndpointAddress;
  UCHAR Interval;
  USBD PIPE TYPE PipeType;
  USBD PIPE HANDLE PipeHandle;
  ULONG MaximumTransferSize;
  ULONG PipeFlags;
  } USBD_PIPE_INFORMATION, *PUSBD_PIPE_INFORMATION;
```

So we have an array of *USBD_INTERFACE_LIST* entries, each of which points to a *USBD_INTERFACE_INFORMATION* structure that contains an array of *USBD_PIPE_INFORMATION* structures. Our immediate task is to fill in the *MaximumTransferSize* member of each of those pipe information structures if we don't want to accept the default value chosen by the parent driver. The default value is *USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE*, which was equal to *PAGE_SIZE* in the DDK I was using at the time I wrote this book. The value we specify isn't directly related either to the maximum packet size for the endpoint (which governs how many bytes can be moved in a single bus transaction) or to the amount of data the endpoint can absorb in a series of transactions (which is determined by the amount of memory available on the device). Instead, it represents the largest amount of data we'll attempt to move with a single URB. This can be less than the largest amount of data that an application might send to the device or receive from the device, in which case our driver must be prepared to break application requests into pieces no bigger than this maximum size. I'll discuss how that task can be accomplished later in "Managing Bulk Transfer Pipes."

The reason that we have to supply a maximum transfer size is rooted in the scheduling algorithm that the host controller drivers use to divide URB requests into transactions within bus frames. If we send a large amount of data, it's possible for our data to hog a frame to the exclusion of other devices. We therefore want to moderate our demands on the bus by specifying a reasonable maximum size for the URBs that we'll send at once.

The code needed to initialize the pipe information structures is something like this:

```
for (ULONG ii = 0; ii < <number of interfaces>; ++ii)
  {
  PUSBD INTERFACE INFORMATION pii = interfaces[ii].Interface;
  for (ULONG ip = 0; ip < pii->NumberOfPipes; ++ip)
    pii->Pipes[ip].MaximumTransferSize = <some constant>;
  }
```

NOTE
The *USBD_CreateConfigurationRequestEx* function initializes the *MaximumTransferSize* member of each pipe information structure to *USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE* and the *PipeFlags* member to 0. Bear this in mind when you look at older driver samples and when you write your own driver.

Once you've initialized the pipe information structures, you're finally ready to submit the configuration URB:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 340

```
SendAwaitUrb(fdo, selurb);
```

Finding the Handles

Successful completion of the select configuration URB leaves behind various handle values that you should record for later use:

- The *UrbSelectConfiguration.ConfigurationHandle* member of the URB is a handle for the configuration.
- The *InterfaceHandle* member of each *USBD_INTERFACE_INFORMATION* structure contains a handle for the interface.
- Each of the *USBD_PIPE_INFORMATION* structures has a *PipeHandle* for the pipe ending in the corresponding endpoint.

For example, the USB42 sample records two handle values (in the device extension):

```
typedef struct  DEVICE EXTENSION {
:
  USBD CONFIGURATION HANDLE hconfig;
  USBD PIPE HANDLE hpipe;
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

pdx->hconfig = selurb->UrbSelectConfiguration.ConfigurationHandle;
pdx->hpipe = interfaces[0].Interface->Pipes[0].PipeHandle;
ExFreePool(selurb);
```

At this point in the program, the select configuration URB is no longer needed and can be discarded.

Shutting Down the Device

When your driver receives an *IRP_MN_STOP_DEVICE* request, you should place the device in its unconfigured state by creating and submitting a select configuration request with a *NULL* configuration pointer:

```
URB urb;
UsbBuildSelectConfigurationRequest(&urb,
  sizeof( URB SELECT CONFIGURATION), NULL);
SendAwaitUrb(fdo, &urb);
```

12.2.3 Managing Bulk Transfer Pipes

The companion content has two sample programs that illustrate bulk transfers. The first and simplest is named USB42. It has an input bulk endpoint that delivers back the constant value 42 each time you read it. (I call this the Answer device because the number 42 is Douglas Adams's answer to the Ultimate Question of Life, the Universe and Everything in The Hitchhiker's Guide to the Galaxy.) The code to do the reading is as follows:

```
URB urb;
UsbBuildInterruptOrBulkTransferRequest(&urb,
  sizeof( URB BULK OR INTERRUPT TRANSFER),
  pdx->hpipe, Irp->AssociatedIrp.SystemBuffer, NULL, cbout,
  USBD TRANSFER DIRECTION IN | USBD SHORT TRANSFER OK, NULL);
status = SendAwaitUrb(fdo, &urb);
```

This code runs in the context of the handler for a *DeviceIoControl* call that uses the buffered method for data access, so the *SystemBuffer* field of the IRP points to the place to which data should be delivered. The *cbout* variable is the size of the data buffer we're trying to fill.

There's not much to explain about this request. You indicate with a flag whether you're reading (*USBD_TRANSFER_DIRECTION_IN*) or writing (*USBD_TRANSFER_DIRECTION_OUT*) the endpoint. You can optionally indicate with another flag bit (*USBD_SHORT_TRANSFER_OK*) whether you're willing to tolerate having the device provide less data than the maximum for the endpoint. The pipe handle is something you capture at *IRP_MN_START_DEVICE* time in the manner already illustrated.

Design of the LOOPBACK Sample

The LOOPBACK sample is considerably more complicated than USB42. The device it manages has two bulk transfer endpoints, one for input and another for output. You can feed up to 4096 bytes into the output pipe, and you can retrieve what you put in from the input pipe. The driver itself uses standard *IRP_MJ_READ* and *IRP_MJ_WRITE* requests for data

IPR2024-00292
KASPERSKY EX 1008 Page 341

movement.

LOOPBACK allows the application to read or write more than the pipe *MaximumTransferSize* during a single operation. This fact imposes constraints on how the driver works:

- Each read or write IRP might require several stages to accomplish. In keeping with the USB specification (specifically, section 5.8.3), each stage except the last must be a multiple of the endpoint's maximum packet size.
- It would obviously not do for the stages of different read or write requests to be intermixed at the device level. Therefore, LOOPBACK queues read and write requests to serialize access to the endpoint.
- To avoid hassles related to IRP cancellation, LOOPBACK piggybacks its read and write URBs on the same *IRP_MJ_READ* or *IRP_MJ_WRITE* it receives from above.

LOOPBACK's *StartIo* Routine

The interesting routines in LOOPBACK are the *StartIo* routine that handles a single read or write request and the I/O completion routine for URB requests sent down to the bus driver. Both read and write IRPs feed into the same *StartIo* routine. The major reason why a single *StartIo* routine is appropriate in this driver is that we want to be doing either a read or a write, but not both simultaneously. Using a single routine is practical because there's very little difference in the way we handle reads and writes:

```
VOID StartIo(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx =
    (PDEVICE EXTENSION) fdo->DeviceExtension;
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  BOOLEAN read = stack->MajorFunctionCode == IRP MJ READ;
  USBD PIPE HANDLE hpipe = read ? pdx->hinpipe : pdx->houtpipe;
  ULONG urbflags =  USBD SHORT TRANSFER OK
    | (read ? USBD TRANSFER DIRECTION IN
    : USBD TRANSFER DIRECTION OUT);
:
  }
```

LOOPBACK sets the *DO_DIRECT_IO* flag in its device object. Consequently, the data buffer is described by an MDL whose address is at *Irp->MdlAddress*. We can determine the length of the requested transfer in two ways. We can fetch *stack->Parameters.Read.Length* or *stack->Parameters.Write.Length*. (Both *Read* and *Write* are identical substructures of the *IO_STACK_LOCATION*, by the way.) Alternatively, we can rely on the MDL:

```
ULONG length = Irp->MdlAddress ?
  MmGetMdlByteCount(Irp->MdlAddress) : 0;
```

Personally, I hate it when there are two ways of doing the same thing because then I worry that one of them will stop working in some future release of the operating system. I've developed the habit of picking the most popular of alternatives in the belief that it's least likely to break over time. Most drivers I've seen that use MDLs get the buffer length from the MDL. Therefore, I do too.

The next logical step in *StartIo* is to calculate the length of the first segment of a potentially multisegment transfer:

```
ULONG seglen = length;
if (seglen > pdx->maxtransfer)
  seglen = pdx->maxtransfer;
```

(LOOPBACK's *StartDevice* function sets *maxtransfer* to the *MaximumTransferSize* for the input and output pipes. I made that equal to 1024 in this driver in order to exercise the multisegment transfer logic. The device firmware itself has a limit of 4096 bytes for a single logical transfer.)

Our call to *UsbBuildInterruptOrBulkTransferRequest* will be a bit more complicated than in the USB42 example because we're using *DO_DIRECT_IO* and because the transfer may require several stages. In preparation for that call, LOOPBACK creates a partial MDL, which describes just a portion of the entire buffer:

```
ULONG PTR va =
  (ULONG PTR) MmGetMdlVirtualAddress(Irp->MdlAddress);
PMDL mdl = IoAllocateMdl((PVOID) (PAGE SIZE - 1), seglen, FALSE,
  FALSE, NULL);
IoBuildPartialMdl(Irp->MdlAddress, mdl, (PVOID) va, seglen);
```

This is the point in coding *StartIo* where you face a major decision. How will you create and send the one or more URBs that

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 342

are required to perform this operation? One alternative, which I don't think is the best, is to create a series of *IRP_MJ_INTERNAL_DEVICE_CONTROL* requests, each with its own URB, and send them down the PnP stack to the USB bus driver. The reason I don't like this choice is that it requires a bunch of extra bookkeeping to control when you complete the main IRP and how you deal with cancellation of the main IRP. I actually followed this plan in the USBISO sample discussed later in this chapter, but there wasn't any sensible alternative in that case because of timing requirements.

The easier choice for a bulk transfer operation is to simply use and reuse the main IRP—the one passed into *StartIo*, in other words—as an envelope within which to stuff the URBs for successive stages. All we need to do is to initialize the next stack location by hand instead of by calling *IoCopyCurrentIrpStackLocationToNext*. Our *StartIo* routine then installs a completion routine and sends the main IRP down to the bus driver. Our completion routine recycles the IRP and the URB to perform the next stage of the transfer. When the last stage completes, our completion routine releases the memory occupied by the USB and arranges to set *IoStatus.Information* equal to the number of bytes actually transferred, as required by the specifications for *IRP_MJ_READ* and *IRP_MJ_WRITE*.

Our completion routine actually needs a bit more information than just the URB address, however. I define the following context structure in LOOPBACK:

```
struct  RWCONTEXT : public  URB
  {
  ULONG PTR va;          // virtual address for next
                         // segment of transfer
  ULONG length;          // length remaining to transfer
  PMDL mdl;              // partial MDL
  ULONG numxfer;         // cumulative transfer count
  :
  };
typedef struct _RWCONTEXT RWCONTEXT, *PRWCONTEXT;
```

(This declaration relies on the fact that my drivers use C++ syntax, so I can derive one structure from another.) The initialization of the context structure is along these lines:

```
PRWCONTEXT ctx = (PRWCONTEXT) ExAllocatePool(NonPagedPool,
  sizeof(RWCONTEXT));
UsbBuildInterruptOrBulkTransferRequest(ctx,
  sizeof( URB BULK OR INTERRUPT TRANSFER),
  hpipe, NULL, mdl, seglen, urbflags, NULL);

ctx->va = va + seglen;
ctx->length = length - seglen;
ctx->mdl = mdl;
ctx->numxfer = 0;
```

Notice that no cast operator is needed for *ctx* because it's derived from the URB structure. The MDL pointer for the URB is the *partial* MDL we created earlier, and the length is the chosen segment length.

After all of this initialization, we can finally prepare and send the IRP down to the bus driver:

```
stack = IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP MJ INTERNAL DEVICE CONTROL;
stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
stack->Parameters.DeviceIoControl.IoControlCode =
  IOCTL INTERNAL USB SUBMIT URB;

IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE)
  OnReadWriteComplete, (PVOID) ctx, TRUE, TRUE, TRUE);

IoCallDriver(pdx->LowerDeviceObject, Irp);
```

It's useful to know that the USB bus driver will accept read/write URBs at *DISPATCH_LEVEL*. This is just as well, considering that *StartIo* will be running at *DISPATCH_LEVEL*.

LOOPBACK's Read/Write Completion Routine

Here's the essential part of the completion routine:

```
NTSTATUS OnReadWriteComplete(PDEVICE OBJECT fdo,
  PIRP Irp, PRWCONTEXT ctx)
  {
  PDEVICE_EXTENSION pdx =
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 343

```
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
  BOOLEAN read = (ctx->UrbBulkOrInterruptTransfer.TransferFlags &
    USBD_TRANSFER_DIRECTION_IN) != 0;
1
  ctx->numxfer += ctx->UrbBulkOrInterruptTransfer.TransferBufferLength;

  NTSTATUS status = Irp->IoStatus.Status;
2
  if (NT_SUCCESS(status) && ctx->length && !Irp->Cancel)
    {
3
    ULONG seglen = ctx->length;
    if (seglen > pdx->maxtransfer)
      seglen = pdx->maxtransfer;
    PMDL mdl = ctx->mdl;
4
    MmPrepareMdlForReuse(mdl);
    IoBuildPartialMdl(Irp->MdlAddress, mdl,
      (PVOID) ctx->va, seglen);

5
    ctx->UrbBulkOrInterruptTransfer.TransferBufferLength =
      seglen;

6
    PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
    stack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;
    stack->Parameters.Others.Argument1 = (PVOID) (PURB) ctx;
    stack->Parameters.DeviceIoControl.IoControlCode =
      IOCTL_INTERNAL_USB_SUBMIT_URB;
    IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
      OnReadWriteComplete, (PVOID) ctx, TRUE, TRUE, TRUE);

7
    ctx->va += seglen;
    ctx->length -= seglen;

8
    IoCallDriver(pdx->LowerDeviceObject, Irp);
    return STATUS_MORE_PROCESSING_REQUIRED
    }

  if (NT_SUCCESS(status))
9
    Irp->IoStatus.Information = ctx->numxfer;
  else
    <recover from error>
10
  IoFreeMdl(ctx->mdl);
  ExFreePool(ctx);
  StartNextPacket(&pdx->dqReadWrite, fdo);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);

  return STATUS_SUCCESS;
  }
```

1. We'll eventually need to set *IoStatus.Information* to the total number of bytes transferred. This statement is where we cumulate the total at the end of a stage.

2. Here we test to see whether there's another stage to do: Did the previous stage complete OK? Is the residual length nonzero? Did someone try to cancel the previous stage?

3. As was true for the first stage, each stage is limited by the maximum transfer size for the pipe. Furthermore, each stage but the last must be a multiple of the endpoint packet size. I didn't mention it earlier, but the maximum transfer size should be chosen to be such a multiple (as I have done here).

4. We're going to reuse the partial MDL for the next stage. *MmPrepareMdlForReuse* resets flag bits and pointers. *IoBuildPartialMdl* initializes the fields in the MDL structure to describe the data to be transferred to or from the main buffer in the next stage. Note that the virtual address (*va* field of the context structure) isn't being used as an address but rather as an index into the buffer described by the main MDL.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 344

5. The URB hasn't been altered. The only thing that's different is the length.

6. The I/O Manager *has*, however, zeroed most of the next stack location to prevent us from relying on its contents. We therefore need to completely reinitialize the next stack location.

7. Here's where we update the virtual address and the residual length for the next time this completion routine gets called.

8. We call *IoCallDriver* to recycle the IRP. What we do next doesn't depend on the status returned by the bus driver.

9. One of the rules for *IRP_MJ_READ* and *IRP_MJ_WRITE* is that, on successful completion, *IoStatus.Information* should equal the number of bytes actually transferred. We've been keeping track of this in *numxfer*, so this is where we finally obey the rule.

10. The remainder of the completion routine is just straightforward cleanup after *StartIo*.

Just to see if you've been paying *really* close attention to *everything* I've been saying in this book, here are three more silly contest questions about the completion routine:

1. Why doesn't the completion routine call *IoReuseIrp* before recycling the IRP?

2. Why does the completion routine always return *STATUS_MORE_PROCESSING_REQUIRED* after sending the recycled IRP down the stack?

3. What deduction can you make from the fact that the author *did* read absolutely everything in the book (several times, in fact), returned *STATUS_SUCCESS* from the completion routine, and yet omitted the boilerplate call to *IoMarkIrpPending*?

Answers to Silly Contest Questions

1. *IoReuseIrp* completely reinitializes an IRP and is appropriate when the originator of an IRP wants to use it again. We want only to reinitialize the next stack location. The only thing about this IRP that would actually require resetting would be the *Cancel* flag. If we found that set, it would imply that someone called *IoCancelIrp* on the main IRP. In that case, we don't try to perform the next stage.

2. If the bus driver pended the stage IRP, it's clearly the right thing to do to return *STATUS_MORE_PROCESSING_REQUIRED*. There will be another call to *IoCompleteRequest* from the bus driver later on, and the system will call this completion routine again then. If the bus driver completed the stage transfer synchronously, this completion routine has *already* been called recursively. We don't want this invocation of *IoCompleteRequest* to do any more work on this IRP in either case.

3. You could deduce that the author is a sanctimonious twit. Either that or the dispatch routine marked the IRP pending and returned *STATUS_PENDING* as part of the normal protocol for queuing the IRP. Or both—these choices are not mutually exclusive.

Error Recovery in LOOPBACK

When you send or receive data to or from a bulk transfer endpoint, the bus and bus driver take care of retrying garbled transmissions. Consequently, if your URB appears to complete successfully, you can be confident that the data you intended to transfer has in fact been transferred correctly. When an error occurs, however, your driver needs to attempt some sort of recovery. There is a well-defined protocol for recovering from an error, illustrated by additional code in LOOPBACK (a subroutine named *RecoverFromError*) that I didn't show you earlier:

First issue an *IOCTL_INTERNAL_USB_GET_PORT_STATUS* request to determine the status of the hub port to which your device is connected.

If the status flags indicate that the port is not enabled but is still connected (that is, you're not dealing with a surprise removal), perform a *URB_FUNCTION_ABORT_PIPE* operation on the failed endpoint to flush all pending I/O, and then reset the port by issuing an *IOCTL_INTERNAL_USB_RESET_PORT*.

In any case, issue a *URB_FUNCTION_RESET_PIPE* to reset the endpoint. Among other things, this clears an endpoint stall condition.

Retry or allow to fail the request that previously failed, depending on the semantics of your device.

An annoyance about these steps is that many of them have to be done at *PASSIVE_LEVEL*, yet you discover the need for them in a completion routine running (perhaps) at *DISPATCH_LEVEL*. To deal with the restrictions, you need to schedule a work item as shown here (see Chapter 14 for an explanation of the mechanics):

```
struct  RWCONTEXT : public  URB
  {
  PIO_WORKITEM rcitem;  // work item created for recovery
```


IPR2024-00292
KASPERSKY EX 1008 Page 345

```
  PIRP Irp;             // the main IRP that we're going to fail
  };

NTSTATUS OnReadWriteComplete(...)
  {
  .
  .
  if (NT_SUCCESS(status))
    Irp->IoStatus.Information = ctx->numxfer;
  else if (status != STATUS_CANCELLED)
    {
    ctx->rcitem = IoAllocateWorkItem(fdo);
    ctx->Irp = Irp;
    IoQueueWorkItem(ctx->rcitem,
      (PIO_WORKITEM_ROUTINE) RecoverFromError,
      CriticalWorkQueue, (PVOID) ctx);
    return STATUS_MORE_PROCESSING_REQUIRED;
    }
  .
  .
  }
```

The actual error recovery routine is as follows:

```
VOID RecoverFromError(PDEVICE_OBJECT fdo, PRWCONTEXT ctx)
  {
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fdo->DeviceExtension;
  BOOLEAN read = (ctx->UrbBulkOrInterruptTransfer.TransferFlags
    & USBD_TRANSFER_DIRECTION_IN) != 0;
  ULONG portstatus = GetStatus(fdo);
  USBD_PIPE_HANDLE hpipe = read ? pdx->hinpipe : pdx->houtpipe;
  if (!(portstatus & USBD_PORT_ENABLED) &&
    (portstatus & USBD_PORT_CONNECTED))
    {
    AbortPipe(fdo, hpipe);
    ResetDevice(fdo);
    }
  ResetPipe(fdo, hpipe);
  IoFreeWorkItem(ctx->rcitem);
  PIRP Irp = ctx->Irp;
  IoFreeMdl(ctx->mdl);
  ExFreePool(ctx);
  StartNextPacket(&pdx->dqReadWrite, fdo);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  }
```

AbortPipe, *ResetDevice*, and *ResetPipe* are helper routines that issue the internal control operations and URBs that I described earlier. Note that when *RecoverFromError* calls *IoCompleteRequest*, our own completion routine isn't called. Therefore, all the cleanup that would normally be done by the completion routine has to be repeated.

It crossed my mind to be tricky in the way I called *RecoverFromError*. If the completion routine happens to be running at *PASSIVE_LEVEL*, it looks as if you can just bypass queuing a work item. This would be a mistake, though, unless you use *IoSetCompletionRoutineEx* to install the completion routine. The problem that can occur otherwise is that whoever sent you the IRP can remove its guard against you being unloaded as soon as *RecoverFromError* calls *IoCompleteRequest*. That leaves a few instructions in *RecoverFromError* and *OnReadWriteComplete* to execute at a time when the driver has been unloaded. Using *IoSetCompletionRoutineEx* prevents the driver from unloading until the completion routine returns. It's much more costly to call *IoSetCompletionRoutineEx* every time than it is to queue a work item in the unlikely case of an I/O error that needs recovery, so I elected to use the work item approach.

The LOOPBACK firmware exhibits a real-world problem that I didn't attempt to solve in the driver. If a failure occurs in a read or write operation, the write and read-back operations might get out of synchronization. You might see this happen if you turn on the Driver Verifier's Low Resources Simulation option for LOOPBACK because pool allocations will start randomly failing. Subsequent invocations of the test program will ordinarily fail because the device firmware is returning the wrong data from its ring buffer.

To solve a problem like the one I just described, somebody—either the driver or an application—has to be aware of the way the device works and has to issue some sort of command to resynchronize the firmware. LOOPBACK is already complicated enough, and the solution is so peculiar to this one device and its firmware that I didn't want to burden the sample with code to deal with it.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 346

12.2.4 Managing Interrupt Pipes

From the device side of the bus, an interrupt pipe is practically identical to a bulk transfer pipe. The only important difference from that perspective is that the host will be polling an interrupt endpoint with a guaranteed frequency. The device will respond with *NAK* except at instants when it will present an interrupt to the host. To report an interrupt event, the device sends an *ACK* to the host after providing whatever morsel of data is supposed to accompany the interrupt.

From the driver's perspective, managing an interrupt pipe is quite a bit more complicated than managing a bulk pipe. When the driver needs to read or write data to a bulk pipe, it just creates an appropriate URB and sends it to the bus driver. But for an interrupt pipe to serve its intended purpose of notifying the host of interesting hardware events, the driver basically needs to keep a read request outstanding at all times. A way to keep a read request active is to use the same idea I showed you in LOOPBACK, wherein we have a completion routine that keeps recycling a URB.

The USBINT sample illustrates how to manage an interrupt pipe with a URB that's always active. Rather than discuss the sample point by point, I just want to go briefly over a couple of key areas:

- You mustn't have a read active when you stop or power down the device. Therefore, USBINT goes to some trouble to knock down its interrupt read when removing power and to restart it when restoring power. Since these steps have to be done asynchronously to avoid violating the rule against blocking during power transitions, the driver uses the arcane *SaveDeviceContext* and *RestoreDeviceContext* callbacks from GENERIC.SYS.
- The completion routine for the interrupt read is, in effect, the interrupt service routine for the driver. You can expect it to run at *DISPATCH_LEVEL* because it's an I/O completion routine. One of its jobs is to reinitialize and reissue the interrupt read so that one is always outstanding.
- As usual, there can be a race condition between the driver cancelling the interrupt read at *StopDevice* or power-down time and the bus driver completing that IRP. Avoiding these races should be old hat to my readers by this point in the book.

12.2.5 Control Requests

If you refer back to Table 12-3, you'll notice that there are 11 standard types of control requests. You and I will never explicitly issue *SET_ADDRESS* requests. The bus driver does that when a new device initially comes on line; by the time we ever get control in a WDM driver, the bus driver has assigned an address to the device and read the device descriptor to learn that *we're* the device driver. I've already discussed how to create the URBs that cause the bus driver to send control requests for getting descriptors or for setting a configuration or an interface in the "Initiating Requests" and "Configuration" sections. In this section, I'll fill in the blanks related to the remaining kinds of control transactions.

Controlling Features

If we want to set or clear a feature of a device, an interface, or an endpoint, we submit a feature URB. For example, the following code (which appears in the FEATURE sample driver in the companion content) sets a vendor-defined interface feature:

```
URB urb;
UsbBuildFeatureRequest(&urb,
  URB_FUNCTION_SET_FEATURE_TO_INTERFACE,
  FEATURE_LED_DISPLAY, 1, NULL);
status = SendAwaitUrb(fdo, &urb);
```

The second argument to *UsbBuildFeatureRequest* indicates whether we want to set or clear a feature belonging to the device, an interface, an endpoint, or another vendor-specific entity on the device. This parameter takes eight possible values, and you can guess without me telling you that they're formed according to the following formula:

```
URB_FUNCTION_ [SET | CLEAR]_FEATURE_TO_
  [DEVICE | INTERFACE | ENDPOINT | OTHER]
```

The third argument to *UsbBuildFeatureRequest* identifies the feature in question. In the FEATURE sample, I invented a feature named *FEATURE_LED_DISPLAY*. The fourth argument identifies a particular entity of whatever type is being addressed. In this example, I wanted to address interface 1, so I coded 1.

USB defines two standard features that you might be tempted to control using a feature URB: the remote wake-up feature and the endpoint stall feature. You don't, however, need to set or clear these features yourself because the bus driver does so automatically. When you issue an *IRP_MN_WAIT_WAKE* request—see Chapter 8—the bus driver ensures that the device's configuration allows for remote wake-up, and it also automatically enables the remote wake-up feature for the device. The bus driver issues a clear feature request to unstall a device when you issue a *RESET_PIPE* URB.

IPR2024-00292
KASPERSKY EX 1008 Page 347

About the FEATURE Sample
The FEATURE sample in the companion content illustrates how to set or clear a feature. The device firmware (in the EZUSB subdirectory) defines a device with no endpoints. The device supports an interface-level feature numbered 42, which is the *FEATURE_LED_DISPLAY* referenced symbolically in the driver. When the feature is set, the Cypress Semiconductor development board's seven-segment LED display becomes illuminated and shows how many times the feature has been set since the device was attached (modulo 10). When the feature is clear, the LED display shows only the decimal point.

The FEATURE device driver (in the SYS subdirectory) contains code to set and clear the feature and to exercise a few other control commands in response to IOCTL requests. Refer to CONTROL.CPP to see this code, which isn't much more complicated than the code fragments displayed in the text.

The test program (in the TEST subdirectory) is a Win32 console application that performs a *DeviceIoControl* to set the custom feature; issues additional *DeviceIoControl* calls to obtain status masks, the configuration number, and the alternate setting for the single interface; waits five seconds; and then performs another *DeviceIoControl* to clear the feature. Each time you run the test, you should see the development board's display light up for five seconds, showing successively larger decimal integers.

Determining Status

If you want to obtain the current status of the device, an interface, or an endpoint, you formulate a get status URB. For example:

```
URB urb;
USHORT epstatus;
UsbBuildGetStatusRequest(&urb,
  URB FUNCTION GET STATUS FROM ENDPOINT,
  <index>, &epstatus, NULL, NULL);
SendAwaitUrb(fdo, &urb);
```

You can use four different URB functions in a get status request, and they allow you to retrieve the current status mask for the device as a whole, for a specified interface, for a specified endpoint, or for a vendor-specific entity. See Table 12-9.

The status mask for a device indicates whether the device is self-powered and whether its remote wake-up feature is enabled. See Figure 12-15. The mask for an endpoint indicates whether the endpoint is currently stalled. See Figure 12-16. USB previously defined interface-level status bits related to power management in the *Interface Power Management* specification that was withdrawn while this book was at press. USB should never prescribe vendor-specific status bits since they're by definition up to vendors to specify.

Operation Code	Retrieve Status From...
URB_FUNCTION_GET_STATUS_FROM_DEVICE	Device as a whole
URB_FUNCTION_GET_STATUS_FROM_INTERFACE	Specified interface
URB_FUNCTION_GET_STATUS_FROM_ENDPOINT	Specified endpoint
URB_FUNCTION_GET_STATUS_FROM_OTHER	Vendor-specific object

Table 12-9. *URB Function Codes Used for Getting Status*

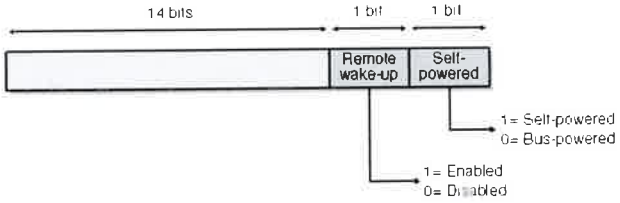


Figure 12-15. *Bits in device status.*

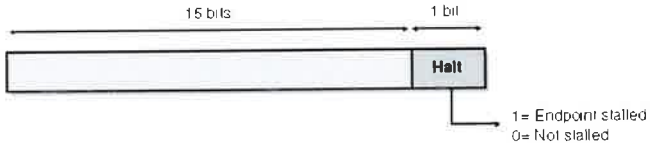


Figure 12-16. *Bits in endpoint status.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 348

12.2.6 Managing Isochronous Pipes

The purpose of an isochronous pipe is to allow the host and the device to exchange time-critical data with guaranteed regularity. The bus driver will devote up to 80 percent of the bus bandwidth to isochronous and interrupt transfers. What this means is that every 125-ms microframe will include reserved time slots long enough to accommodate maximum-size transfers to or from each of the isochronous and interrupt endpoints that are currently active. Figure 12-17 illustrates this concept for three different devices. Devices A and B each have an isochronous endpoint, for which a fixed and relatively large amount of time is reserved in every microframe. Device C has an interrupt endpoint whose polling frequency is once every two microframes; it has a reservation for a small portion of every second microframe. During microframes that don't include a poll of Device C's interrupt endpoint, additional bandwidth is available, perhaps for bulk transfers or other purposes.

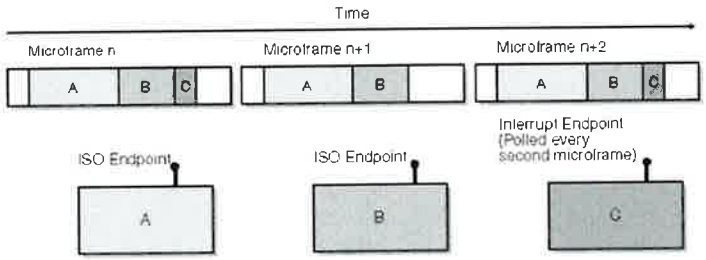


Figure 12-17. *Allocation of bandwidth to isochronous and interrupt endpoints.*

Reserving Bandwidth

The bus driver reserves bandwidth for you when you enable an interface by examining the endpoint descriptors that are part of the interface. Reserving bandwidth is just like buying a theater ticket, though: you don't get a refund if you don't use the space. Consequently, it's important to enable an interface that contains an isochronous endpoint only when you'll be using the bandwidth you thereby reserve, and it's important that the endpoint's declared maximum transfer size be approximately the amount you intend to use. Normally, a device with isochronous capability has a default interface that doesn't have any isochronous or interrupt endpoints. When you know you're about to access that capability, you enable an *alternate setting* of the same interface that *does* have the isochronous or interrupt endpoints.

An example will clarify the mechanics of reserving bandwidth. The USBISO sample in the companion content has an interface with a default and an alternate setting. The default setting has no endpoints. The alternate setting has an isochronous endpoint with a maximum transfer size of 256 bytes. See Figure 12-18.

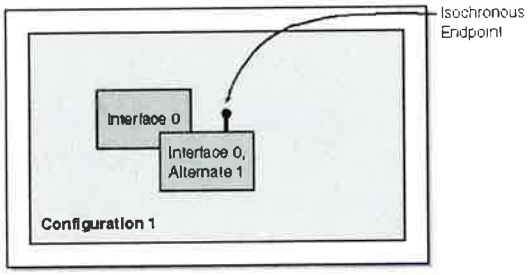


Figure 12-18. *Descriptor structure for the USBISO device.*

At *StartDevice* time, we select a configuration based on the default interface. Since the default interface doesn't have an isochronous or interrupt endpoint in it, we don't reserve any bandwidth just yet. When someone opens a handle to the device, however, we invoke the following *SelectAlternateInterface* helper function to switch to the alternate setting for our interface. (Again, I've omitted the error checking.)

```
NTSTATUS SelectAlternateInterface(PDEVICE_OBJECT fdo)
  {
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
1
  PUSB_INTERFACE_DESCRIPTOR pid =
    USBD_ParseConfigurationDescriptorEx(pdx->pcd, pdx->pcd, 0, 1, -1, -1, -1);
  ULONG npipes = pid->bNumEndpoints;
2
  ULONG size = GET_SELECT_INTERFACE_REQUEST_SIZE(npipes);
  PURB urb = (PURB) ExAllocatePool(NonPagedPool, size);
```


IPR2024-00292
KASPERSKY EX 1008 Page 349

```
RtlZeroMemory(urb, size);
3
4
UsbBuildSelectInterfaceRequest(urb, size, pdx->hconfig, 0, 1);
urb->UrbSelectInterface.Interface.Length = GET USBD INTERFACE SIZE(npipes);
urb->UrbSelectInterface.Interface.Pipes[0].MaximumTransferSize = PAGE SIZE;
5
NTSTATUS status = SendAwaitUrb(fdo, &urb);
6
7
if (NT SUCCESS(status))
  {
  pdx->hinpipe =
    urb.UrbSelectInterface.Interface.Pipes[0].PipeHandle;
  status = STATUS SUCCESS;
  }
ExFreePool(urb);
return status;
}
```

1. Before we can allocate space for the URB, we need to know how many pipe descriptors it will contain. The most common way to find this number is to go back to the grand unified configuration descriptor and find the descriptor for interface 0, alternate setting 1. That descriptor contains a count of endpoints, which is the same as the number of pipes that we're about to open.

2. *GET_SELECT_INTERFACE_REQUEST_SIZE* calculates the number of bytes needed to hold a select interface request that will open the specified number of pipes. We can then allocate memory for the URB and initialize it to 0. The real code sample in the companion content checks to make sure that the call to *ExAllocatePool* succeeded, by the way.

3. Here we build a URB to select alternate setting 1 (the last argument) of interface number 0 (the next-to-last argument).

4. We must do these two additional initialization steps to finish setting up the URB. Failing to set the interface information structure's length earns you a *STATUS_BUFFER_TOO_SMALL* failure right away. Failing to set the *MaximumTransferSize* fields of the pipe descriptors earns you a *STATUS_INVALID_PARAMETER* when you try to read or write the pipe.

5. When we submit this URB, the parent driver automatically closes the current setting of this interface, including all of its endpoints. Then the parent driver tells the device to enable the alternate setting, and it creates pipe descriptors for the endpoints that are part of the alternate setting. If opening the new interface fails for some reason, the parent driver reopens the previous interface, and all your previous interface and pipe handles remain valid.

6. My *SendAwaitUrb* helper function simply returns an error if it's unable to select the one and only alternate setting for this interface. I'll have a bit more to say about how you should handle errors after this numbered list.

7. In addition to selecting the new interface at the device level, the parent driver also creates an array of pipe descriptors from which we can extract handles for later use.

The select interface call might fail because not enough free bandwidth exists to accommodate our endpoint. We find out about the failure by examining the URB status:

```
if (URB STATUS(&urb) == USBD STATUS NO BANDWIDTH)
```

Dealing with lack of bandwidth poses a bit of a problem. The operating system doesn't currently provide a convenient way for competing drivers to negotiate a fair allocation. Neither does it provide for any sort of notification that some other driver has failed to acquire needed bandwidth so that we might give up some of ours. In this state of affairs, therefore, you have two basic choices. One choice is to provide multiple alternate interface settings, each of which has a different maximum transfer size for its isochronous endpoint or endpoints. When you detect an allocation failure, you can try to select progressively less demanding settings until you finally succeed.

A savvy end user who's able to launch the Windows XP Device Manager applet can display a property page for the USB host controller—see Figure 12-19—that displays information about the current allocation of bandwidth. Double-clicking one of the devices listed in the page brings up the property display for the device in question. A well-crafted page can perhaps communicate with the associated device driver to scale back its demand for bandwidth. This whole area seems ripe for a more automatic Microsoft-driven solution, though.

Your other choice for handling lack of bandwidth is to allow an IRP to fail in such a way that an application can alert the end user to the problem. Perhaps the end user can unplug something so that your device can be accommodated. This is the option I chose in the USBISO sample except that I didn't bother to put code in the test application that would respond to a bandwidth allocation failure—TEST.EXE will just fail. To adopt this option, you need to know how the failure shows up back in user

IPR2024-00292
KASPERSKY EX 1008 Page 350

mode. If the URB fails with *USBD_STATUS_NO_BANDWIDTH*, the *NTSTATUS* code you get back from the internal control IRP is *STATUS_DEVICE_DATA_ERROR*, which isn't very specific. An application call to *GetLastError* will retrieve *ERROR_CRC* as the error code. There's no easy way for an application to discover that the real cause of the error is a lack of bandwidth, unfortunately. If you're interested in diving down this particular rat hole to reach a conclusion, read the sidebar.

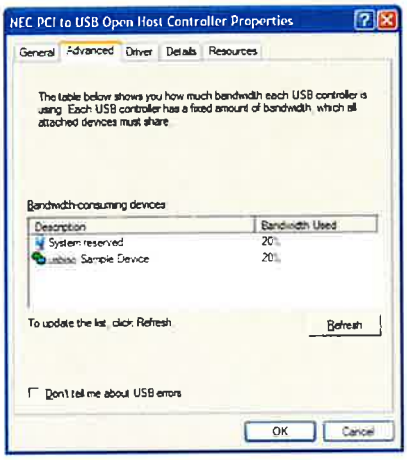


Figure 12-19. *A property page for the USB host controller.*

How an Application Discovers You're out of Bandwidth

Suppose you do what USBISO does and try to select the high-bandwidth alternate interface when you receive an *IRP_MJ_CREATE*. Further suppose you complete the IRP with the status code you get back when there's not enough bandwidth—namely, *STATUS_DEVICE_DATA_ERROR*. Your application caller will eventually see *ERROR_CRC*, as I said in the main text. What now? The application can't send you an IOCTL to find out the real cause of the error because it doesn't have a handle to your device. You allowed the *IRP_MJ_CREATE* to fail, remember? So maybe you need to have a way for people to open handles to your device that doesn't try to reserve bandwidth. Then you need some other way for an application to ask for bandwidth, perhaps by means of an IOCTL operation. Or perhaps your application just interprets *ERROR_CRC* from a call to *CreateFile* as meaning there's no bandwidth. Actual data errors are pretty unlikely, after all, so that interpretation would be correct much of the time.

But the best solution would be a specific *NTSTATUS* code and matching Win32 error code that mean "no bandwidth." Keep your eyes on NTSTATUS.H and WINERROR.H for future developments.

USBISO performs the converse operation of selecting the original default interface when it receives the *IRP_MJ_CLOSE* for the last remaining open handle. That operation entails issuing another select interface URB, but with the value 0 for the alternate interface index.

Initiating a Series of Isochronous Transfers

You can use an isochronous pipe either to read or write data in discrete chunks or to provide or consume data in a continuous stream. Data streaming is probably the most frequent occupation for an isochronous pipe, actually. But in addition to understanding the mechanics of working with the USB bus driver, you must understand and solve additional problems related to data buffering, rate matching, and so on if you want to operate a streaming pipe. The *kernel-streaming* component of the operating system deals with all these additional problems. Unfortunately, we didn't have time to include a chapter on kernel streaming in this book, even in the second edition. I'm therefore going to show you only how to program a discrete transfer over an isochronous pipe.

To read from or write to an isochronous pipe, you'll of course use a URB with the appropriate function code. But there are a few wrinkles that you haven't seen yet associated with creating and submitting the isochronous URB. First, you must be aware of how the device will break up a transfer into *packets*. In general, the device is free to accept or deliver any amount of data less than the endpoint's declared maximum. (Any leftover bandwidth on the bus simply won't be used.) The packet size the device will use doesn't have any other necessary relationship with the endpoint maximum, with the maximum amount of data you said you'd transfer in a URB, or with the amount of data the device and the application can exchange in a series of transactions. The firmware for the USBISO device, for example, works with 16-byte packets, even though the isochronous endpoint in question can handle up to 256 bytes per frame, according to its descriptor. You must have a priori knowledge of how big these packets will be before you construct a URB because the URB must include an array of descriptors for each packet that will be exchanged, and each of these descriptors must indicate how big the packet will be.

In an impractical simple situation, you can allocate an isochronous URB in the following way:

IPR2024-00292
KASPERSKY EX 1008 Page 351

```
ULONG length = MmGetMdlByteCount(Irp->MdlAddress);
ULONG packsize = 16; // a constant in USBISO
ULONG npackets = (length + packsize - 1) / packsize;
ASSERT(npackets <= 255);
ULONG size = GET ISO URB SIZE(npackets);
PURB urb = (PURB) ExAllocatePool(NonPagedPool, size);
RtlZeroMemory(urb, size);
```

The key step in this fragment is the use of the *GET_ISO_URB_SIZE* macro to calculate the total size needed for an isochronous URB to transfer a given number of data packets. A single URB can accommodate a maximum of 255 isochronous packets (1024 in the case of a high-speed device), by the way, which is why I put the *ASSERT* statement in this code. Limiting the application to just 255 packets is not practical, as I said, so we will do something more complex in the real USBISO sample driver. For the time being, though, I just want to describe the mechanics of building a single URB for an isochronous (ISO) transfer.

NOTE
As indicated in the text, a single URB can transfer up to 255 packets to a full-speed device in the course of that many 1-ms frames. For a high-speed device, the maximum packet count is 1024 in the course of up to 128 1-ms frames. Furthermore, each URB should contain a multiple of 8 packets. This makes sense because there are 8 microframes in one frame.

There being no *UsbBuildXxxRequest* macro for building an isochronous URB, we go on to initialize the new URB by hand:

```
urb->UrbIsochronousTransfer.Hdr.Length = (USHORT) size;
urb->UrbIsochronousTransfer.Hdr.Function =
  URB FUNCTION ISOCH TRANSFER;
urb->UrbIsochronousTransfer.PipeHandle = pdx->hinpipe;
urb->UrbIsochronousTransfer.TransferFlags =
  USBD TRANSFER DIRECTION IN | USBD SHORT TRANSFER OK;
urb->UrbIsochronousTransfer.TransferBufferLength = length;
urb->UrbIsochronousTransfer.TransferBufferMDL =
  Irp->MdlAddress;
urb->UrbIsochronousTransfer.NumberOfPackets = npackets;
urb->UrbIsochronousTransfer.StartFrame = frame;
for (ULONG i = 0; i < npackets; ++i, length -= packsize)
  {
  urb->UrbIsochronousTransfer.IsoPacket[i].Offset = i * packsize;
  }
```

The array of packet descriptors collectively describes the entire data buffer that we'll read in to or write out from. This buffer has to be contiguous in virtual memory, which basically means that you need a single MDL to describe it. It would be pretty hard to violate this rule. Reinforcing the idea of contiguity, each packet descriptor contains just the offset and the length for a portion of the entire buffer and not an actual pointer. The host controller driver is responsible for setting the length; you're responsible for setting the offset.

The second wrinkle with starting an isochronous transfer involves timing. USB uniquely identifies each frame or microframe, as the case may be, with an ever-increasing number. It's sometimes important that a transfer begin in a specific frame. The parent driver allows you to indicate this fact by explicitly setting the *StartFrame* field of the URB. USBISO doesn't depend on timing, however. You might therefore think it could set the *USBD_START_ISO_TRANSFER_ASAP* flag to indicate that the transfer should be started as soon as possible. Setting the flag would, in fact, work in versions of Windows prior to Windows XP. Unfortunately, Windows XP introduced a bug such that an ASAP transfer that would start more than 256 frames in the future gets scheduled right away. In the context of this sample, this bug causes the packets to be transferred 0, 256, 2, 3, ... To avoid this problem, I revised USBISO to use a specific frame number calculated as follows:

```
ULONG frame = GetCurrentFrame(pdx) + 2;
Where the GetCurrentFrame function is this one:
ULONG GetCurrentFrame(PDEVICE EXTENSION pdx)
  {
  URB urb;
  urb.UrbGetCurrentFrameNumber.Hdr.Length =
    sizeof(struct  URB GET CURRENT FRAME NUMBER);
  urb.UrbGetCurrentFrameNumber.Hdr.Function =
    URB FUNCTION GET CURRENT FRAME NUMBER;

  NTSTATUS status = SendAwaitUrb(pdx->DeviceObject, &urb);
  if (!NT SUCCESS(status))
    return 0;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 352

```
    return urb.UrbGetCurrentFrameNumber.FrameNumber;
    }
```

I don't necessarily recommend that you always get the current frame number for an ISO transfer. USBISO has to do so because it's reading 256 packets. In a more usual situation, you have a streaming driver that issues a few read or write URBs for just a few packets, and you're continually recycling these URBs as time goes on. You won't run afoul of the 256-frames-in-the-future problem in that case and can just use the *USBD_START_ISO_TRANSFER_ASAP* flag.

The final wrinkle in isochronous processing has to do with how the transfer ends. The URB itself will succeed overall, even though one or more packets had data errors. The URB has a field named *ErrorCount* that indicates how many packets encountered errors. If this ends up nonzero, you can loop through the packet descriptors to examine their individual status fields.

Achieving Acceptable Performance

Achieving acceptable performance with isochronous transfers will be something of a challenge in a streaming environment or in a situation in which you have a multistage transfer to orchestrate. One strategy is to arrange to run in a real-time thread at *DISPATCH_LEVEL* and submit URBs nearly directly to the bus driver via the *SubmitIsoOutUrb* function in the bus driver's direct-call interface. If you're doing input operations, however, or if you need to support platforms earlier than Windows XP, you need to submit multiple URBs, such that the bus driver has one to work with as soon as an earlier one finishes.

The USBISO sample in the companion content illustrates how to manage a large block transfer using multiple subsidiary URBs. The basic idea behind USBISO's read/write logic is to have the completion routine for subsidiary IRPs complete the main read/write IRP when the last subsidiary IRP finishes. To make this idea work, I declared the following special-purpose context structure:

```
typedef struct  RWCONTEXT {
  PDEVICE_EXTENSION pdx;
  PIRP mainirp;
  NTSTATUS status;
  ULONG numxfer;
  ULONG numirps;
  LONG numpending;
  LONG refcnt;
  struct {
    PIRP irp;
    PURB urb;
    PMDL mdl;
    } sub[1];
  } RWCONTEXT, *PRWCONTEXT;
```

The dispatch routine for *IRP_MJ_READ*—USBISO doesn't handle *IRP_MJ_WRITE* requests—calculates the number of subsidiary IRPs required for the complete transfer and allocates one of these context structures, as follows:

```
ULONG packsize = 16;
ULONG segsize = USBD_DEFAULT_MAXIMUM_TRANSFER_SIZE;
if (segsize / packsize > 255)
  segsize = 255 * packsize;
ULONG numirps = (length + segsize - 1);
ULONG ctxsize = sizeof(RWCONTEXT) +
  (numirps - 1) * sizeof(((PRWCONTEXT) 0)->sub);
PRWCONTEXT ctx = (PRWCONTEXT) ExAllocatePool(NonPagedPool,
  ctxsize);
RtlZeroMemory(ctx, ctxsize);
ctx->numirps = ctx->numpending = numirps;
ctx->pdx = pdx;
ctx->mainirp = Irp;
ctx->refcnt = 2;
Irp->Tail.Overlay.DriverContext[0] = (PVOID) ctx;
```

I'll explain the purpose of the last two statements in this sequence when I discuss USBISO's cancellation logic in the "Handling Cancellation of the Main IRP" section. We now perform a loop to construct *numirps* *IRP_MJ_INTERNAL_DEVICE_CONTROL* requests. At each iteration of the loop, we call *IoAllocateIrp* to create an IRP with one more stack location than is required by the device object immediately under us. We also allocate a URB to control one stage of the transfer and a partial MDL to describe the current stage's portion of the main I/O buffer. We record the address of the IRP, the URB, and the partial MDL in an element of the *RWCONTEXT* structure's *sub* array. We initialize the URB in the same way as I showed you earlier. Then we initialize the subsidiary IRP's first *two* I/O stack locations, as follows:

IPR2024-00292
KASPERSKY EX 1008 Page 353

```
IoSetNextIrpStackLocation(subirp);
PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(subirp);
stack->DeviceObject = fdo;
stack->Parameters.Others.Argument1 = (PVOID) urb;
stack->Parameters.Others.Argument2 = (PVOID) mdl;

stack = IoGetNextIrpStackLocation(subirp);
stack->MajorFunction = IRP MJ INTERNAL DEVICE CONTROL;
stack->Parameters.Others.Argument1 = (PVOID) urb;
stack->Parameters.DeviceIoControl.IoControlCode =
  IOCTL INTERNAL USB SUBMIT URB;

IoSetCompletionRoutine(subirp, (PIO COMPLETION ROUTINE)
  OnStageComplete, (PVOID) ctx, TRUE, TRUE, TRUE);
```

The first stack location is for use by the *OnStageComplete* completion routine we install. The second is for use by the lower-level driver.

Once we've built all the IRPs and URBs, it's time to submit them to the bus driver. Before we do so, however, it's prudent to check whether the main IRP has been cancelled, and it's necessary to install a completion routine for the main IRP. The logic at the end of the dispatch routine looks like the following code:

```
IoSetCancelRoutine(Irp, OnCancelReadWrite);
if (Irp->Cancel)
  {
  status = STATUS CANCELLED;
  if (IoSetCancelRoutine(Irp, NULL))
    --ctx->refcnt;
  }
else
  status = STATUS SUCCESS;

IoSetCompletionRoutine(Irp,
  (PIO COMPLETION ROUTINE) OnReadWriteComplete,
  (PVOID) ctx, TRUE, TRUE, TRUE);
IoMarkIrpPending(Irp);
IoSetNextIrpStackLocation(Irp);

if (!NT SUCCESS(status))
  {
  for (i = 0; i < numirps; ++i)
    {
    if (ctx->sub[i].urb)
      ExFreePool(ctx->sub[i].urb);
    if (ctx->sub[i].mdl)
      IoFreeMdl(ctx->sub[i].mdl);
    }
  CompleteRequest(Irp, status, 0);
  return STATUS PENDING;
  }

for (i = 0; i < numirps; ++i)
  IoCallDriver(pdx->LowerDeviceObject, ctx->sub[i].irp);

return STATUS_PENDING;
```

Handling Cancellation of the Main IRP

To explain the two completion routines that I'm using in this example—that is, *OnReadWriteComplete* for the main IRP and *OnStageComplete* for each subsidiary IRP—I need to explain how USBISO handles cancellation of the main IRP. Cancellation is a concern because we've submitted a potentially large number of subsidiary IRPs that might take some time to finish. We can't complete the main IRP until all of the subsidiary IRPs complete. We should, therefore, provide a way to cancel the main IRP and all outstanding subsidiary IRPs.

I'm sure you recall from Chapter 5 that IRP cancellation implicates a number of knotty synchronization issues. If anything, the situation in this driver is worse than usual.

USBISO's cancellation logic is complicated by the fact that we can't control the timing of calls to the subsidiary IRPs' completion routine—those IRPs are owned by the bus driver once we submit them. Suppose you wrote the following cancel

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 354

routine:

```
VOID OnCancelReadWrite(PDEVICE OBJECT fdo, PIRP Irp)
  {
  IoReleaseCancelSpinLock(Irp->CancelIrql);
1
  PRWCONTEXT ctx = (PRWCONTEXT)
    Irp->Tail.Overlay.DriverContext[0];
2
  for (ULONG i = 0; i < ctx->numirps; ++i)
    IoCancelIrp(ctx->sub[i].irp);
    <additional steps>
  }
```

1. We saved the address of the *RWCONTEXT* structure in the *DriverContext* area of the IRP precisely so that we could retrieve it here. *DriverContext* is ours to use so long as we own the IRP. Since we returned *STATUS_PENDING* from the dispatch routine, we never relinquished ownership.

2. Here we cancel all the subsidiary IRPs. If a subsidiary IRP has already completed or is currently active on the device, the corresponding call to *IoCancelIrp* won't do anything. If a subsidiary IRP is still in the host controller driver's queue, the host controller driver's cancel routine will run and complete the subsidiary IRP. In all three cases, therefore, we can be sure that all subsidiary IRPs will be completed sometime soon.

This version of *OnCancelReadWrite* is *almost* complete, by the way, but it needs an additional step that I'll show you after I've explained the synchronization problem we need to solve. I can illustrate the problem by showing the completion routines we'll use with two naive mistakes built in. Here's the completion routine for one stage of the total transfer:

```
NTSTATUS OnStageComplete(PDEVICE OBJECT fdo, PIRP subirp,
  PRWCONTEXT ctx)
  {
1
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PIRP mainirp = ctx->mainirp;
  PURB urb = (PURB) stack->Parameters.Others.Argument1;
  if (NT SUCCESS(Irp->IoStatus.Status))
    {
2
    InterlockedExchangeAdd((PLONG) &ctx->numxfer,
      (LONG) urb->UrbIsochronousTransfer.TransferBufferLength);
    }
  else
3
    ctx->status = Irp->IoStatus.Status;
4
  ExFreePool(urb);
  IoFreeMdl((PMDL) stack->Parameters.Others.Argument2);
5
  IoFreeIrp(subirp);  // <== don't do this
  if (InterlockedDecrement(&ctx->numpending) == 0)
    {
    IoSetCancelRoutine(mainirp, NULL); // <== also needs some work
    mainirp->IoStatus.Status = ctx->status;
6
    IoCompleteRequest(mainirp, IO NO INCREMENT);
    }
  return STATUS MORE PROCESSING REQUIRED;
  }
```

1. This stack location is the extra one that the dispatch routine allocated. We need the address of the URB for this stage, and the stack was the most convenient place to save that address.

2. When a stage completes normally, we update the cumulative transfer count for the main IRP here. The final value of *numxfer* will end up in the main IRP's *IoStatus.Information* field.

3. We initialized *status* to *STATUS_SUCCESS* by zeroing the entire context structure. If any stage completes with an error, this statement will record the error status. The final value will end up in the main IRP's *IoStatus.Status* field.

4. We no longer need the URB or the partial MDL for this stage, so we release the memory they occupied here.

5. This call to *IoFreeIrp* is the naive part of this completion routine, as I'll explain shortly.

IPR2024-00292
KASPERSKY EX 1008 Page 355

6. When the last stage completes, we'll also complete the main IRP. Once we've submitted the subsidiary IRPs, this is the only place where we complete the main IRP, so we can be sure that the main IRP pointer is valid.

Here's the naive version of the completion routine for the main IRP:

```
NTSTATUS OnReadWriteComplete(PDEVICE OBJECT fdo, PIRP Irp,
  PRWCONTEXT ctx)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) ctx->pdx;
1
  if (Irp->Cancel)
    Irp->IoStatus.Status = STATUS CANCELLED;
  else if (NT SUCCESS(Irp->IoStatus.Status))
    Irp->IoStatus.Information = ctx->numxfer;
2
  ExFreePool(ctx);     // <== don't do this
  return STATUS SUCCESS;
  }
```

1. If someone tried to cancel the main IRP, this statement will set the corresponding ending status.

2. Releasing the context structure's memory is a problem, as I'll explain.

I've been building up to a big and dramatic exposé of a synchronization problem associated with IRP cancellation, and here it finally is: Suppose our cancel routine gets called after one or more of the calls to *IoFreeIrp* has already happened inside *OnStageComplete*. You can see that we might call *IoCancelIrp* with an invalid pointer in such a case. Or suppose the cancel routine gets called more or less simultaneously with *OnReadWriteComplete*. In that case, we might have the cancel routine accessing the context structure after it gets deleted.

You might attempt to solve these problems with various subterfuges. Can *OnStageComplete* nullify the appropriate subsidiary IRP pointer in the context structure, and can *OnCancelReadWrite* check before calling *IoCancelIrp*? (Yes, but there's still no way to guarantee that the call to *IoFreeIrp* doesn't squeeze in between whatever test *OnCancelReadWrite* makes and the moment when *IoCancelIrp* is finally done modifying the cancel-related fields of the IRP.) Can you protect the various cleanup steps with a spin lock? (That's a horrible idea because you'll be holding the spin lock across calls to time-consuming functions.) Can you take advantage of knowing that the current release of Windows XP always cleans up completed IRPs in an asynchronous procedure call (APC) routine? (No, for the reasons I discussed back in Chapter 5.)

I struggled long and hard with this problem before inspiration finally struck. Why not, I finally realized, protect the context structure and the subsidiary IRP pointers with a reference count so that *both* the cancel routine and the main completion routines can share responsibility for cleaning them up? That's what I ended up doing. I put a reference count field (*refcnt*) in the context structure and initialized it to the value 2. One reference is for the cancel routine; the other is for the main completion routine. I wrote the following helper function to release the memory objects that are the source of the problem:

```
BOOLEAN DestroyContextStructure(PRWCONTEXT ctx)
  {
  if (InterlockedDecrement(&ctx->refcnt) > 0)
    return FALSE;
  for (ULONG i = 0; i < ctx->numirps; ++i)
    if (ctx->sub[i].irp)
      IoFreeIrp(ctx->sub[i].irp);
  ExFreePool(ctx);
  return TRUE;
  }
```

I call this routine at the end of the cancel routine:

```
VOID OnCancelReadWrite(PDEVICE OBJECT fdo, PIRP Irp)
  {
  IoReleaseCancelSpinLock(Irp->CancelIrql);
  PRWCONTEXT ctx = (PRWCONTEXT)
    Irp->Tail.Overlay.DriverContext[0];
  for (ULONG i = 0; i < ctx->numirps; ++i)
    IoCancelIrp(ctx->sub[i].irp);
  PDEVICE EXTENSION pdx = ctx->pdx;
  if (DestroyContextStructure(ctx))
    {
    CompleteRequest(Irp, STATUS CANCELLED, 0);
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    }
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 356

```
}
```

I omitted the call to *IoFreeIrp* in the stage completion routine and added one more line of code to decrement the reference count once it's certain that the cancel routine hasn't been, and can no longer, be called:

```
NTSTATUS OnStageComplete(PDEVICE OBJECT fdo, PIRP subirp,
  PRWCONTEXT ctx)
  {
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PIRP mainirp = ctx->mainirp;
  PURB urb = (PURB) stack->Parameters.Others.Argument1;
  if (NT SUCCESS(Irp->IoStatus.Status))
    ctx->numxfer +=
    urb->UrbIsochronousTransfer.TransferBufferLength;
  else
    ctx->status = Irp->IoStatus.Status;
  ExFreePool(urb);
  IoFreeMdl((PMDL) stack->Parameters.Others.Argument2);
  if (InterlockedDecrement(&ctx->numpending) == 0)
    {
    if (IoSetCancelRoutine(mainirp, NULL))
      InterlockedDecrement(&ctx->refcnt);
    mainirp->IoStatus.Status = ctx->status;
    IoCompleteRequest(mainirp, IO NO INCREMENT);
    }
  return STATUS MORE PROCESSING REQUIRED;
  }
```

Recall that *IoSetCancelRoutine* returns the previous value of the cancel pointer. If that's *NULL*, the cancel routine has already been called and will call *DestroyContextStructure*. If that's not *NULL*, however, it will no longer be possible for the cancel routine ever to be called, and we must use up the cancel routine's claim on the context structure.

I also replaced the unconditional call to *ExFreePool* in the main completion routine with a call to *DestroyContextStructure*:

```
NTSTATUS OnReadWriteComplete(PDEVICE OBJECT fdo, PIRP Irp,
  PRWCONTEXT ctx)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) ctx->pdx;
  if (Irp->Cancel)
    Irp->IoStatus.Status = STATUS CANCELLED;
  else if (NT SUCCESS(Irp->IoStatus.Status))
    Irp->IoStatus.Information = ctx->numxfer;

  if (DestroyContextStructure(ctx))
    {
    IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
    return STATUS SUCCESS;
    }
  else
    return STATUS MORE PROCESSING REQUIRED;
  }
```

Here's how this extra logic works. If the cancel routine ever gets called, it will run through the context structure calling *IoCancelIrp* for each of the subsidiary IRPs. Even if all of them have already completed, these calls will still be safe because we won't have called *IoFreeIrp* yet. The reference to the context structure will also be safe because we won't have called *ExFreePool* yet. The cancel routine finishes up by calling *DestroyContextStructure*, which will decrement the reference counter. If the main completion routine hasn't run yet, *DestroyContextStructure* will return *FALSE*, whereupon the cancel routine will return. The context structure still exists at this point, which is good because the main completion routine will reference it soon. The completion routine's eventual call to *DestroyContextStructure* will release the subsidiary IRPs and the context structure itself. The completion routine will then return *STATUS_SUCCESS* to allow the main IRP to finish completing.

Suppose calls to the cancel and main completion routines happen in the other order. In that case, *OnReadWriteComplete*'s call to *DestroyContextStructure* will simply decrement the reference count and return *FALSE*, whereupon *OnReadWriteComplete* will return *STATUS_MORE_PROCESSING_REQUIRED*. The context structure still exists. We can also be sure that we still own the IRP and the *DriverContext* field from which the cancel routine will fetch the context pointer. The cancel routine's call to *DestroyContextStructure* will, however, reduce the reference count to 0, release the memory, and return *TRUE*. The cancel routine will then release the remove lock and call *IoCompleteRequest* for the main IRP. That adds up to *two* calls to *IoCompleteRequest* for the same IRP. You know that you're not allowed to complete the same IRP twice, but the prohibition is not against calling *IoCompleteRequest* twice per se. If the first invocation of *IoCompleteRequest* results in calling a completion

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 357

routine that returns *STATUS_MORE_PROCESSING_REQUIRED*, a subsequent, duplicate call is perfectly OK.

The only remaining case in this analysis occurs when the cancel routine never gets called at all. This is, of course, the *normal* case because IRPs don't usually get cancelled. We discover this fact when we call *IoSetCancelRoutine* in preparation for completing the main IRP. If *IoSetCancelRoutine* returns a non-*NULL* value, we know that *IoCancelIrp* has not yet been called for the main IRP. (Had it been, the cancel pointer would *already* be *NULL*, and *IoSetCancelRoutine* would have returned *NULL*.) Furthermore, we know that our own cancel routine can now never be called and will therefore not have a chance to reduce the reference count. Consequently, we reduce the reference count by hand so that *OnReadWriteComplete*'s call to *DestroyContextStructure* will release the memory.

Where's the Synchronization?
You'll notice that I didn't use a spin lock to guard the code I just showed you for testing for cancellation inside the dispatch routine. Synchronization between that code and some hypothetical caller of IoCancelIrp is implicit in the facts that IoSetCancelRoutine is an interlocked exchange operation and that *IoCancelIrp* sets the *Cancel* flag before calling *IoSetCancelRoutine*. Refer to the discussion in Chapter 5 for a sketch of how *IoCancelIrp* works.

Our dispatch routine's first call to *IoSetCancelRoutine* might occur after *IoCancelIrp* sets the *Cancel* flag but before *IoCancelIrp* does its own call to *IoSetCancelRoutine*. Our dispatch routine will see that the *Cancel* flag is set and make a second call to *IoSetCancelRoutine*. If this second call happens to precede *IoCancelIrp's* call to *IoSetCancelRoutine*, the cancel routine won't be called. We'll also decrement the reference count on the context structure so that it gets released on the first call to *DestroyContextStructure*.

If our dispatch routine's second call to *IoSetCancelRoutine* follows *IoCancelIrp*'s, we won't decrement the reference count. Either the cancel routine or the completion routine will end up releasing the context structure.

If our dispatch routine tests the *Cancel* flag before *IoCancelIrp* sets it, or if *IoCancelIrp* has never even been called for this IRP, we'll go ahead and start the subsidiary IRPs. If *IoCancelIrp* was called in the distant past before we installed a cancel routine, it will have simply set the *Cancel* flag and returned. What happens after that is just the same as when our dispatch routine nullifies the cancel pointer before *IoCancelIrp* calls *IoSetCancelRoutine*.

So you see, you don't always need a spin lock to give you multiprocessor safety: sometimes an atomic interlocked operation will do the trick by itself.

Associated IRPs?
At first blush, *IoMakeAssociatedIrp* looks like an alternative way to create the subsidiary IRPs that USBISO needs. The idea behind *IoMakeAssociatedIrp* is that you can create a number of *associated* IRPs to fulfill a *master* IRP. When the last associated IRP completes, the I/O Manager automatically completes the master IRP.

Unfortunately, associated IRPs aren't a good way to solve any of the problems that USBISO grapples with. Most important, WDM drivers aren't supposed to use *IoMakeAssociatedIrp*. Indeed, the completion logic for associated IRPs is incorrect in Windows 98/Me—it doesn't call any completion routines for the master IRP when the last associated IRP finishes. Even in Windows XP, however, the I/O Manager won't cancel associated IRPs when the master IRP is cancelled. Furthermore, the call to *IoFreeIrp* for an associated IRP occurs inside *IoCompleteRequest*, in whatever thread context happens to be current. This fact makes it harder to safely cancel the associated IRPs.

12.2.7 Idle Power Management for USB Devices

I'm sure you'll agree with me that power management in a WDM driver isn't complicated enough. (Not!) In Chapter 8, I discussed strategies for keeping a device in a low-power state when it's not in use, whatever "in use" means for your particular device. Beginning with Windows XP, there is a special protocol for USB devices called *selective suspend*. In this final section of the chapter, I'll describe the mechanics a function driver should use to implement this protocol.

NOTE
The Chapter 8 WAKEUP sample in the companion content contains the code discussed in this section.

The selective suspend feature introduced in Windows XP solves a wake-up problem that arises in composite devices. Suppose you have a two-function device, with each function being managed by a separate function driver. Now suppose one of the function drivers issues both an *IRP_MN_WAIT_WAKE* and an *IRP_MN_SET_POWER* to put its interface in the D2 state, but the other function driver leaves its interface in the D0 state. The first function driver might be relying on a wake-up signal to repower its interface. If the physical device were *not* composite, the parent driver would have armed its wake-up feature and put it in D2. Resuming the device would then generate a wake-up signal, and the parent driver would complete the *WAIT_WAKE*.

But wake-up signaling doesn't happen in the composite device. The parent driver doesn't power down the real device (and doesn't arm its wake-up signaling) unless all function drivers independently ask to have their interfaces powered down. In the

IPR2024-00292
KASPERSKY EX 1008 Page 358

example I gave, only one out of two function drivers powered its interface down. Since the real device never suspends, it has no reason ever to generate a wake-up signal. The function driver for the interface that was supposed to have been suspended doesn't realize this, however, and is therefore not trying to communicate with the device. The end result is a dead function on the device.

Selective suspend provides coordination between the function drivers to solve the problem. Here's how this works. Instead of directly powering down its interface, a function driver issues an IOCTL to the parent driver. The import of the IOCTL is, "I'm ready to be suspended, and here's a function you can call so I can do that." When all function drivers for a particular composite device issue this IOCTL, the parent driver can call all of the callback routines. Each callback routine depowers its own interface. The parent driver then depowers the real device. A subsequent wake-up signal reactivates each function driver that issued an *IRP_MN_WAIT_WAKE*. Voilà! No more dead functions.

Most USB function drivers manage a single interface. You shouldn't assume that a particular interface will never be part of a composite device, and you *should* assume that function drivers for other interfaces on the same device will be relying on wake-up signalling even if you don't. Therefore, you should follow the selective suspend protocol I'm about to describe. If your driver happens to run on a platform earlier than Windows XP, you should disable your own wake-up and automatic-suspend features by default. You can provide instructions to allow the end user to enable one or both of such features in situations in which you won't trip on the dead-interface problem.

Note that even if your driver manages *all* the interfaces on a particular device, you should invoke the selective suspend protocol because the Microsoft drivers also rely on it to help them compensate for hardware bugs in various chip sets.

The first thing you need to do is declare some additional members in your device extension structure:

```
typedef struct  DEVICE EXTENSION {
:
  PIRP SuspendIrp;
  LONG SuspendIrpCancelled;
  USB IDLE CALLBACK INFO cbinfo;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Then, when you decide it's time to put your device in a low-power state because you consider it to be idle, you issue an internal control request to your parent driver to register a callback routine (*SelectiveSuspendCallback*):

```
NTSTATUS IssueSelectiveSuspendRequest(PDEVICE EXTENSION pdx)
  {
  PIRP Irp = IoAllocateIrp(pdx->LowerDeviceObject->StackSize,
    FALSE);
  pdx->cbinfo.IdleCallback =
    (USB IDLE CALLBACK) SelectiveSuspendCallback;
  pdx->cbinfo.IdleContext = (PVOID) pdx;
  PIO STACK LOCATION stack = IoGetNextIrpStackLocation(Irp);
  stack->MajorFunction = IRP MJ INTERNAL DEVICE CONTROL;
  stack->Parameters.DeviceIoControl.IoControlCode =
    IOCTL INTERNAL USB SUBMIT IDLE NOTIFICATION;
  stack->Parameters.DeviceIoControl.Type3InputBuffer =
    &pdx->cbinfo;
  pdx->SuspendIrp = Irp;
  pdx->SuspendIrpCancelled = 0;
  IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE)
    SelectiveSuspendCompletionRoutine, (PVOID) pdx,
    TRUE, TRUE, TRUE);
  IoCallDriver(pdx->LowerDeviceObject, Irp);
  return STATUS SUCCESS;
  }
```

You use an asynchronous IRP for this control operation because it might remain outstanding for a long period, and you might, therefore, need to cancel it. I followed my own advice from Chapter 5 in orchestrating the cancellation and completion routines:

```
VOID CancelSelectiveSuspend(PDEVICE EXTENSION pdx)
  {
  PIRP Irp = (PIRP) InterlockedExchangePointer(
    (PVOID*) &pdx->SuspendIrp, NULL);
  if (Irp)
    {
    IoCancelIrp(Irp);
    if (InterlockedExchange(&pdx->SuspendIrpCancelled, 1))
      IoFreeIrp(Irp);
```


IPR2024-00292
KASPERSKY EX 1008 Page 359

```
    }
  }

NTSTATUS SelectiveSuspendCompletionRoutine(PDEVICE OBJECT junk,
  PIRP Irp, PDEVICE EXTENSION pdx)
  {
  NTSTATUS status = Irp->IoStatus.Status;
  if (InterlockedExchangePointer((PVOID*) &pdx->SuspendIrp, NULL)
    || InterlockedExchange(&pdx->SuspendIrpCancelled, 1))
    IoFreeIrp(Irp);

  if (!NT SUCCESS(status) && status != STATUS POWER STATE INVALID)
    GenericWakeupFromIdle(pdx->pgx, FALSE);

  return STATUS MORE PROCESSING REQUIRED;
  }
```

(I'll explain in a moment why there's a call to *GenericWakeupFromIdle* here.)

In a normal case, the parent driver pends the idle-notification IRP until all devices attached to the same hub ask to be idled. When that occurs, the parent driver calls the callback routine, whereupon you carry out two steps. First you make sure that your wake-up feature (if any) is armed and that a *WAIT_WAKE* request is outstanding. Second you request a power IRP to put your device in your desired low-power state. For example, in a driver that uses GENERIC.SYS for power management, your callback routine could be this simple:

```
VOID SelectiveSuspendCallback(PDEVICE EXTENSION pdx)
  {
  GenericWakeupControl(pdx->pgx, ManageWaitWake);
  GenericIdleDevice(pdx->pgx, PowerDeviceD2, TRUE);
  }
```

The *TRUE* argument to *GenericIdleDevice* makes the power operation synchronous, which is a requirement in this situation. In fact, returning from the callback routine before your device is in its low-power state would cause the parent driver to incorrectly believe you couldn't power down, whereupon the whole hub and all attached devices would stay powered.

If the parent driver allows the idle notification request to fail, your device may have ended up in a low-power state, and you should repower it from your completion routine—hence the call to *GenericWakeupFromIdle* in the example. The only exception will be if the request completes with *STATUS_POWER_STATE_INVALID*, which happens if you put the device in the D3 state while the IRP is outstanding. *That* can happen, for example, if the system is hibernating.

Finally, don't forget to cancel your outstanding idle notification IRP at *StopDevice* time.

IPR2024-00292
KASPERSKY EX 1008 Page 360

Chapter 13

Human Interface Devices

A Human Interface Device (HID) is one that communicates with the host computer using structured reports. The HID class of devices consists primarily of devices that end users use to interact with computers. The class includes keyboards, mice, and game controllers of all kinds, but it can include any imaginable knob, switch, button, slider, exoskeletal device, or other type of control that a user might use to control a computer. Noninteractive devices such as bar code readers and measuring instruments might also be designed to fit into the HID class. In addition, HID devices can incorporate user-information components such as lights, displays, force feedback, and other indicators.

HID devices built for the Universal Serial Bus conform to the *Device Class Definition for Human Input Devices*. Additional specifications relevant to force feedback are in the USB *Device Class Definition for Physical Interface Devices*. HID relies on extensive sets of numeric constants, whose definition can be found in the *HID Usage Tables specification*. All of these specifications are available for free download from www.usb.org.

Although the HID specifications are oriented toward USB implementations, any sort of device can function, in whole or in part, as a HID device. The important characteristic of a HID device, once again, is that the host is able to perform input and output operations using report packets that conform to an extremely flexible structure definition, called the *report descriptor*.

Applications access HID-compliant keyboards and mice only indirectly, by handling window messages whose character and content haven't fundamentally changed in over twenty years. Applications access other sorts of HID device through COM interfaces that are part of the DirectX component of Windows and through Win32 API calls.

13.1 Drivers for HID Devices

The Microsoft class driver for HID devices, HIDCLASS.SYS, provides the overall framework for WDM drivers that manage HID devices on all the Windows platforms. Microsoft also supplies a *HIDCLASS* minidriver named HIDUSB.SYS to handle USB devices whose device or interface descriptor indicates that they belong to the HID class. Consequently, if your USB device belongs to the HID class, you may not have to write a special-purpose driver at all because the Microsoft class driver and minidriver fully support the USB specifications.

If you're designing a USB device that includes some HID-like functionality, don't forget that you can make it a *composite* device by defining several interfaces. The generic parent driver will separate the functions of your device so that the system will load the standard Microsoft drivers for the HID function.

Microsoft also provides drivers for standard PS2 keyboards and mice, and for serial-port mice. These drivers, along with HIDCLASS, sit below class filter drivers named KBDCLASS and MOUCLASS, which present a consistent interface to higher-level components.

You might need to write a custom minidriver to replace HIDUSB.SYS if your USB device or interface provides or consumes structured reports but doesn't belong to the HID class. In such a case, your minidriver will furnish a faux HID descriptor to *HIDCLASS*, and it will also create structured reports matching that descriptor in response to input events.

Even with a true HID-class USB device, you might have to write your own minidriver to support custom functionality. I've used this approach to build drivers for several specialized devices, including a gaming mouse with lots of buttons and lights and a head-tracking device that delivers sensor values that must be transformed into position reports. In these cases, the devices are nominally HID-class USB devices, but my clients want the devices to deliver different reports from the ones generated by the firmware. It was not practical in these cases to put the custom functionality into firmware.

Finally, if you have a non-USB device (other than a standard keyboard or mouse) that includes HID-like functionality, a custom *HIDCLASS* minidriver is the only practical way to make that device accessible to DirectX and thence to existing applications.

13.2 Reports and Report Descriptors

A HID device transfers information in a block known as a *report*. The report contains bit and integer fields formatted according to a *report descriptor*. Much of the HID specification and related documents describe the contents of reports and report descriptors in great detail. I'll analyze two sample report descriptors here to help you understand the specifications.

13.2.1 Sample Keyboard Descriptor

To start with, I suggest that you download the so-called HID Descriptor Tool (DT.EXE) from http://www.usb.org. The tool

IPR2024-00292
KASPERSKY EX 1008 Page 361

allows you to create and edit report descriptors using symbolic names. Figure 13-1 illustrates the user interface and one of the example descriptors that come with the tool.

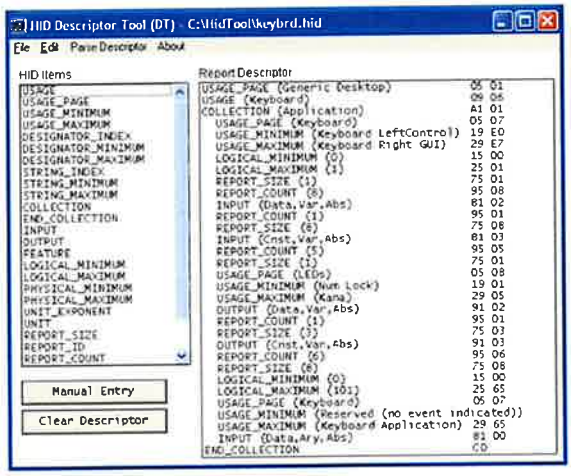


Figure 13-1. *Using the HID Tool to define a keyboard report descriptor.*

The first *item* in the sample report descriptor establishes the *usage page*, which essentially specifies a namespace for interpreting certain numeric constants in subsequent elements of the descriptor. You need the *HID Usage Tables* document to interpret the numbers. For example, the usage code 6 means *keyboard* in the generic desktop page but *sailing simulation device* in the simulation controls page.

The second item specifies the *usage* for the next top-level *collection* in the descriptor. In the HID specification, a collection serves to group related data items. For example, a *physical collection* groups items collected at one geometric point, whereas an *application collection* groups items that might be familiar to applications. A further concept, the *logical collection*, allows related items to be grouped into a composite data structure, such as a byte count followed by data. These concepts are so abstract as to be nearly meaningless, and Microsoft assigns additional meaning, as follows:

- A top-level collection, such as the one beginning with the third item in the keyboard sample, corresponds to an individually addressable entity. Acting as a bus driver, *HIDCLASS* creates a physical device object (PDO) for each top-level collection. The device identifier for the collection includes a generic compatible ID, based on the usage code. See Table 13-1. If the collection has any other usage, *HIDCLASS* won't create a compatible ID. Refer to Chapter 15 for more information about the importance of a compatible ID in locating a driver. The PDO then becomes the base of a PnP device stack for some type of device. Note that multiple top-level collections give rise to multiple device stacks. For this to work in practice, the device must use report identifiers to distinguish between the different collections.

- A *link collection* is one nested within a top-level collection. Link collections provide an organizational hierarchy that applications can use to group related controls in a complex device. On a game pad, for example, one can use link collections to distinguish between buttons actuated by the left and right hands. There seems little point to this generality, however, when applications typically require end users to assign meanings to controls based on numbers rather than position in a hierarchy. But perhaps I just haven't seen enough applications and HID devices to make a comprehensive judgment.

Usage Page	Usage	Compatible ID
Generic desktop	Pointer or mouse	*HID_DEVICE_SYSTEM_MOUSE*
	Keyboard or keypad	*HID_DEVICE_SYSTEM_KEYBOARD*
	Joystick or game pad	*HID_DEVICE_SYSTEM_GAME*
	System control	*HID_DEVICE_SYSTEM_CONTROL*
Consumer	(Any)	*HID_DEVICE_SYSTEM_CONSUMER*

Table 13-1. *HIDCLASS-Compatible ID for Each Supported Usage*

Within the single top-level collection for the sample keyboard, the most important items are the *main* items named INPUT and OUTPUT. An INPUT item corresponds to a field in an input report, whereas an OUTPUT item corresponds to a field in an output report. There can also be FEATURE items that define fields in a feature report, but the keyboard sample doesn't include any of them. A number of *global items* precede the main items in order to describe the presentation and meaning of the data itself.

It's important to realize that INPUT, OUTPUT, and FEATURE report items can be interleaved in the report descriptor. The logical collection structure within a top-level collection isn't important in determining which data items appear together in a given report. Rather, the *type* of the items governs. Thus, the example keyboard descriptor mixes INPUT and OUTPUT items

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 362

in a way that might suggest that there are five reports, or else a single bidirectional report. In reality, there is a single input report defined by the INPUT items and a single output report defined by the OUTPUT items.

The main items, along with all the qualifying global items, define the bit layout of a structured report. To visualize the report, assign bits from right to left and don't leave any unused bits for alignment purposes. Treat multibit values, including those that span byte boundaries, as little-endian (least significant bit on the right of the resulting picture). Subdivide the result into bytes, which the device transmits from right to left.

In the keyboard report, we have five data items in the collection, and they define an input report and an output report (see Figure 13-2):

- An input item consisting of eight (*REPORT_COUNT*) single-bit values (*REPORT_SIZE* 1), each of which can vary from 0 (*LOGICAL_MINIMUM*) to 1 (*LOGICAL_MAXIMUM*). The meaning of the bits corresponds to keyboard usages (*USAGE_PAGE*) E0 through E7 (*USAGE_MINIMUM* and *USAGE_MAXIMUM*). In other words, byte 0 of the input report contains flag bits to indicate which of the shift-type keys on the keyboard are currently depressed.
- A constant input item consisting of one (*REPORT_COUNT*) 8-bit value (*REPORT_SIZE*). This is byte 1 of the input report, and it's simply a placeholder that contains no valid data.
- An output item consisting of five (*REPORT_COUNT*) single-bit (*REPORT_SIZE*) values. The *LOGICAL_MINIMUM* and *LOGICAL_MAXIMUM* values previously specified apply to these values because they haven't been overridden. The meaning of the bits is different, however: they correspond to LEDs (*USAGE_PAGE*) with labels such as *Num Lock* (*USAGE_MINIMUM* and *USAGE_MAXIMUM*). In other words, the low-order 5 bits of byte 0 of the output report contain flags to control LEDs for the toggling keys.
- A constant output item consisting of one (*REPORT_COUNT*) 3-bit (*REPORT_SIZE*) value. These 3 bits pad out the output report to a full byte.
- An input item consisting of six (*REPORT_COUNT*) 8-bit values (*REPORT_SIZE*), ranging from 0 through 101 (*LOGICAL_MINIMUM* and *LOGICAL_MAXIMUM*) and corresponding to keys on a standard 101-key keyboard (*USAGE_PAGE*, *USAGE_MINIMUM*, and *USAGE_MAXIMUM*). In other words, bytes 2 through 7 of the input report contain the codes for up to six keys that are being simultaneously held down.

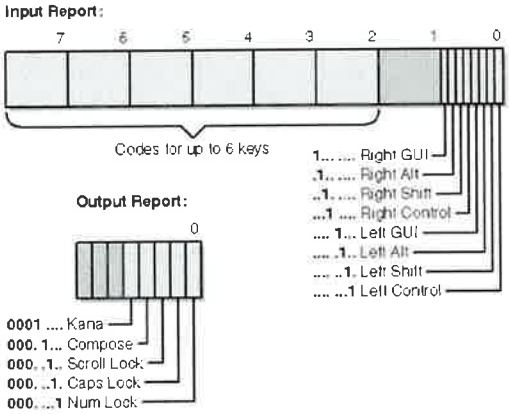


Figure 13-2. *Layout of keyboard input and output reports.*

13.2.2 HIDFAKE Descriptor

Figure 13-3 illustrates the report descriptor used in the HIDFAKE sample driver in the companion content. This report descriptor has a few features that are different from the keyboard sample:

- The top-level application's usage is "Gun Device" from the Gaming Controls page. This was an artificial choice that I made to avoid difficulty installing the sample driver. For any usage listed in Table 13-1, *HIDCLASS* will supply a compatible device identifier along with the device's specific ID. Windows XP will then prefer a signed driver matching the compatible ID to an unsigned driver (such as HIDFAKE.SYS) matching the device's specific ID. (See Chapter 15 for more information about how Windows XP chooses drivers.) It's nearly impossible to switch to the specific driver.
- I used three logical collections within the main collection. The logical collections merely serve to highlight the three-report structure of the descriptor. The sample would work perfectly well without them.
- The descriptor includes an input report and two feature reports. The input report (1) contains a single button usage. The first feature report (2) is for returning a driver version number, and the second feature report (3) is to allow the test applet to control the state of the fake button.

IPR2024-00292
KASPERSKY EX 1008 Page 363

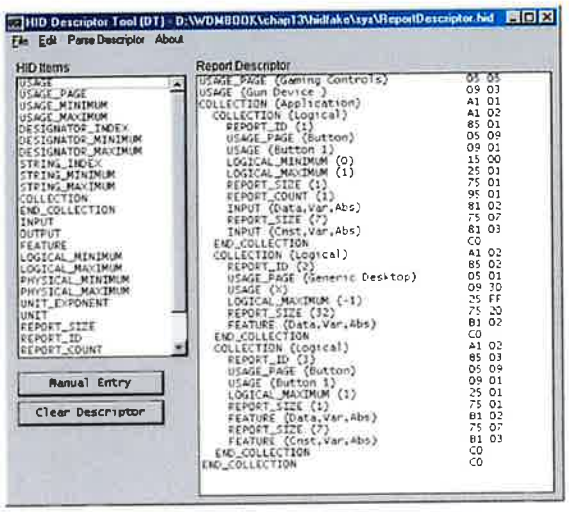


Figure 13-3. *Using the HID Tool to define the HIDFAKE report descriptor.*

HIDFAKE illustrates one fine point about report descriptors. Feature reports pretty much need to have identifying numbers because the HID specification calls for them in the *Get_Report_Request* and *Set_Report_Request* control pipe commands. If any report in a top-level collection has an identifier, all reports in that collection must. In reality, though, HIDFAKE models a notional device that has a real button report and *no* feature reports. I defined the feature reports as a way for the test applet to communicate "out of band" with the driver. If we were dealing with a real device, therefore, the driver would have to *insert* a report identifier in each input report that it read from the device.

13.3 *HIDCLASS* Minidrivers

As previously discussed, Microsoft supplies a driver (HIDUSB.SYS) for any USB device built according to the HID specification. This section describes how you can build a *HIDCLASS* minidriver for some other type of device that you want to have masquerade as HID.

13.3.1 DriverEntry

The *DriverEntry* function for a *HIDCLASS* minidriver is similar to that in a regular WDM driver, but only up to a point. In this routine, you initialize the *DRIVER_OBJECT* data structure with pointers to *AddDevice* and *DriverUnload* routines as well as to dispatch routines for just three types of I/O request packet (IRP): *IRP_MJ_PNP*, *IRP_MJ_POWER*, and *IRP_MJ_INTERNAL_DEVICE_CONTROL*. Then you build a *HID_MINIDRIVER_REGISTRATION* structure and call *HidRegisterMinidriver*, which is one of the functions exported by HIDCLASS.SYS. Table 13-2 describes the fields in the *HID_MINIDRIVER_REGISTRATION* structure.

Field Name	Description
Revision	(*ULONG*) Minidriver sets this field to *HID_REVISION*, which currently equals 1.
DriverObject	(*PDRIVER_OBJECT*) Minidriver sets this field to the same value passed as the *DriverObject* argument to *DriverEntry*.
RegistryPath	(*PUNICODE_STRING*) Minidriver sets this field to the same value passed as the *RegistryPath* argument to *DriverEntry*.
DeviceExtensionSize	(*ULONG*) Size in bytes of the device extension structure used by the minidriver.
DevicesArePolled	(*BOOLEAN*) *TRUE* if the minidriver's devices need to be polled for reports. *FALSE* if the devices spontaneously send reports when data becomes available.

The only field whose meaning isn't completely straightforward is the *DevicesArePolled* flag. Most devices will spontaneously generate a report whenever the end user does something, and they'll notify the host via some sort of interrupt. For this kind of device, you set the *DevicesArePolled* flag to *FALSE*. *HIDCLASS* will then attempt to keep two IRPs (called *ping-pong IRPs*) active to read reports. Your minidriver is expected to queue the IRPs and complete them in order when the device interrupts.

Some devices don't spontaneously generate reports. For that kind of device, set the *DevicesArePolled* flag to *TRUE*. *HIDCLASS* will then issue IRPs to read reports in a timing loop. Your minidriver reads report data from the device only in response to each IRP. Higher-level components, such as an application using DirectX interfaces, can specify the polling interval. Think twice before setting *DevicesArePolled* to *TRUE*: most devices require it to be *FALSE*.

Here's a nearly complete example of the *DriverEntry* function in a *HIDCLASS* minidriver:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 364

```
extern "C" NTSTATUS DriverEntry(PDRIVER OBJECT DriverObject,
  PUNICODE STRING RegistryPath)
  {
  DriverObject->DriverExtension->AddDevice = AddDevice;
  DriverObject->DriverUnload = DriverUnload;
  DriverObject->MajorFunction[IRP MJ INTERNAL DEVICE CONTROL] =
    DispatchInternalControl;
  DriverObject->MajorFunction[IRP MJ PNP] = DispatchPnp;
  DriverObject->MajorFunction[IRP MJ POWER] = DispatchPower;

  HID MINIDRIVER REGISTRATION reg;
  RtlZeroMemory(&reg, sizeof(reg));
  reg.Revision = HID REVISION;
  reg.DriverObject = DriverObject;
  reg.RegistryPath = RegistryPath;
  reg.DeviceExtensionSize = sizeof(DEVICE EXTENSION);
  reg.DevicesArePolled = FALSE;  // <== depends on your hardware

  return HidRegisterMinidriver(&reg);
  }
```

13.3.2 Driver Callback Routines

Most of the class/minidriver interfaces in Windows XP involve a set of callback routines that the minidriver specifies in a registration call made from *DriverEntry*. Most class drivers completely take over the management of the *DRIVER_OBJECT* while handling the registration call. This means that the class drivers each install their own *AddDevice* and *DriverUnload* functions and their own IRP dispatch routines. The class drivers then make calls to the minidriver callback routines to carry out vendor-specific actions.

HIDCLASS operates this way as well, but with a twist. When you call *HidRegisterMinidriver*, *HIDCLASS* installs its own function pointers in your *DRIVER_OBJECT*, just as most class drivers would. Instead of using a set of callback routines whose addresses your minidriver provides in the *HID_MINIDRIVER_REGISTRATION* structure (there are none), it remembers the *AddDevice* and *DriverUnload* pointers and the addresses of your dispatch routines for *PNP*, *POWER*, and *INTERNAL_DEVICE_CONTROL* requests. *These minidriver routines do not have exactly the same functions as like-named routines in regular WDM drivers, though.* I'll explain in this section how to write these callback routines.

AddDevice Callback

The *AddDevice* callback in a *HIDCLASS* minidriver has a prototype similar to that of a regular *AddDevice* function:

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT fdo);
```

There are two important differences between the minidriver callback and the regular function:

- The device object argument refers to a function device object (FDO) that *HIDCLASS* has already created.
- Prior to Windows XP, *HIDCLASS* ignored the return code from the minidriver callback.

Since *HIDCLASS* creates the FDO before calling your minidriver *AddDevice* callback, you don't need to call *IoCreateDevice* or, indeed, to do practically any of the things that you normally do in a WDM *AddDevice* function. Rather, you just need to initialize your device extension structure and return. Versions of Windows prior to Windows XP will ignore the return code from this callback. Consequently, if an error arises in your *AddDevice* callback, you need to set a flag in your device extension that you can inspect at *StartDevice* time:

```
typedef struct  DEVICE EXTENSION {

  NTSTATUS AddDeviceStatus;
  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Another issue to be aware of is that the FDO's *DeviceExtension* pointer is the address of an extension structure that's private to *HIDCLASS*. The first few members of that private structure are mapped by the *HID_DEVICE_EXTENSION* structure in the DDK:

```
typedef struct  HID DEVICE EXTENSION {
  PDEVICE OBJECT PhysicalDeviceObject;
  PDEVICE OBJECT NextDeviceObject;
  PVOID MiniDeviceExtension;
  } HID_DEVICE_EXTENSION, *PHID_DEVICE_EXTENSION;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 365

```
To find your device extension, you must follow this pointer chain:
PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) ((PHID DEVICE EXTENSION)
  (fdo->DeviceExtension))->MiniDeviceExtension;
```

You use similar constructions to get the PDO address and to get what I call the *LowerDeviceObject* in this book. (*HIDCLASS* calls it the *NextDeviceObject*.) Being such a lazy typist, I usually define macros to make my life easier while I'm writing the minidriver:

```
#define PDX(fdo) ((PDEVICE EXTENSION) ((PHID DEVICE EXTENSION) \
  ((fdo)->DeviceExtension))->MiniDeviceExtension)
#define PDO(fdo) (((PHID DEVICE EXTENSION) ((fdo)->DeviceExtension)) \
  ->PhysicalDeviceObject)
#define LDO(fdo) (((PHID DEVICE EXTENSION) ((fdo)->DeviceExtension)) \
  ->NextDeviceObject)
Using these macros and the preceding fragment of a DEVICE EXTENSION structure, your
minidriver's AddDevice callback might look like this:
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT fdo)
  {
  PDEVICE EXTENSION pdx = PDX(fdo);
  NTSTATUS status = STATUS SUCCESS;
  <initialization code for DEVICE EXTENSION members>
  pdx->AddDeviceStatus = status;  // <== whatever is left over
  return status; // in case you're running in >= XP
  }
```

The point of returning a real status code from *AddDevice* is that in Windows XP and later systems, HIDCLASS will fail its own *AddDevice* call if you do, and that will short-circuit the initialization of your device. But since HIDCLASS ignores the code in earlier versions of the operating system, you need to provide a way for your *StartDevice* function to return an error code.

DriverUnload Callback

HIDCLASS calls your minidriver *DriverUnload* callback as a subroutine from its own *DriverUnload* routine. If you created any global objects in your *DriverEntry* routine, you'll clean those up in the *DriverUnload* callback.

DispatchPnp Callback

You specify the *DispatchPnp* callback as if it were the dispatch function for *IRP_MJ_PNP*, by setting an array element in the driver object's *MajorFunction* table. *HIDCLASS* calls your callback as a subroutine while handling Plug and Play IRPs of various types. Your callback routine can perform most of the same operations that a function driver would perform for this type of IRP. See Chapter 6 for full details. There are two exceptions:

- Your *IRP_MN_START_DEVICE* handler needs to test the error flag set by your *AddDevice* callback (I called it *AddDeviceStatus* in the earlier fragment) and to fail the IRP if the flag indicates an error. This is how you cope with the fact that *HIDCLASS* ignores the return code from *AddDevice* in versions of Windows prior to Windows XP.
- Your *IRP_MN_REMOVE_DEVICE* handler does *not* call *IoDetachDevice* or *IoDeleteDevice*. Instead, it should simply release any resources that were allocated by the *AddDevice* callback. *HIDCLASS* itself will take care of detaching and deleting the FDO.

The HIDFAKE sample driver uses GENERIC.SYS. Its *DispatchPnp* routine therefore looks like this:

```
NTSTATUS DispatchPnp(PDEVICE OBJECT fdo, PIRP Irp)
  {
  return GenericDispatchPnp(PDX(fdo)->pgx, Irp);
  }
```

Apart from using the PDX macro to locate the device extension structure, this code is the same as would appear in a regular function driver that uses GENERIC.SYS. The *RemoveDevice*, *StartDevice*, and *StopDevice* functions are different from regular ones, though:

```
VOID RemoveDevice(PDEVICE OBJECT fdo)
{
A
  }

NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
```


IPR2024-00292
KASPERSKY EX 1008 Page 366

```
    PCM PARTIAL RESOURCE LIST raw,
    PCM PARTIAL RESOURCE LIST translated)
    {
    PDEVICE EXTENSION pdx = PDX(fdo);
    if (!NT SUCCESS(pdx->AddDeviceStatus))
      return pdx->AddDeviceStatus;
B
    :

    return STATUS SUCCESS;
    }

  VOID StopDevice(PDEVICE OBJECT fdo, BOOLEAN oktouch)
C
    :

    }
```

HIDFAKE itself has no code at the points labeled A, B, and C. If you use this sample as a template for your own minidriver, you'll write code to do the following:

A. Clean up any resources (such as memory, lookaside lists, and so on) allocated in AddDevice. HIDFAKE has no such resources.

B. Configure the device as discussed in previous chapters. HIDFAKE has no hardware and therefore has nothing to do in this step.

C. Deconfigure the device by reversing the steps done in StartDevice. Since HIDFAKE does nothing in StartDevice, it doesn't need to do anything here either.

DispatchPower Callback

You specify the *DispatchPower* callback as if it were the dispatch routine for *IRP_MJ_POWER*, by setting an array element in the driver object's *MajorFunction* table. *HIDCLASS* calls your callback as a subroutine while handling power IRPs of various types. In most cases, your callback should simply pass the IRP down to the next driver without performing any other actions because *HIDCLASS* contains all the power-management support needed by typical devices (including *WAIT_WAKE* support).

If you set the *DevicesArePolled* flag to *FALSE* in your call to *HidRegisterMinidriver*, *HIDCLASS* will cancel its ping-pong IRPs before forwarding a power request that reduces power. If you have simply piggybacked on these IRPs to send requests further down the PnP stack, you therefore won't need to worry about cancelling them. If you have cached pointers to these IRPs somewhere, you should provide a cancel routine.

NOTE
If your minidriver uses GENERIC.SYS, consider using the *GenericCacheControlRequest* and *GenericUncacheControlRequest* functions to keep track of IRPs that you pend. These functions include race-safe cancel logic.

Here's an example of the *DispatchPower* callback in a *HIDCLASS* minidriver:

```
NTSTATUS DispatchPower(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = PDX(fdo);
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  IoCopyCurrentIrpStackLocationToNext(Irp);
1
  if (stack->MinorFunction == IRP MN SET POWER
    && stack->Parameters.Power.Type == DevicePowerState)
    {
    DEVICE POWER STATE newstate =
      stack->Parameters.Power.State.DeviceState;
    if (newstate == PowerDeviceD0)
      {
2
      IoSetCompletionRoutine(Irp, (PIO COMPLETION ROUTINE)
        PowerUpCompletionRoutine, (PVOID) pdx, TRUE, TRUE, TRUE);
      }
    else if (pdx->devpower == PowerDeviceD0)
      {
3
      // TODO save context information, if any
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 367

```
        pdx->devpower = newstate;
        }
      }
4
    return PoCallDriver(LDO(fdo), Irp);
    }

  NTSTATUS PowerUpCompletionRoutine(PDEVICE OBJECT fdo, PIRP Irp,
    PDEVICE EXTENSION pdx)
    {
5
    // TODO restore device context without blocking this thread
    pdx->devpower = PowerDeviceD0;
    return STATUS SUCCESS;
    }
```

1. You needn't do anything special with any power IRP except a *SET_POWER* for a device power state.
2. When restoring power, you install a completion routine before forwarding the IRP down the stack.
3. When removing power, you save any context information before forwarding the IRP. To deal with the possibility that *HIDCLASS* might lower power in steps (for example, first D2 and later D3), you also need to keep track of the current device power state. Whether or not your device has context information to save, this is also the time to cancel any subsidiary IRPs that your driver has issued, terminate polling threads, and so on. *HIDCLASS* will be calling you at *PASSIVE_LEVEL* in a system thread that you're allowed to block if necessary while performing these tasks.
4. As usual, you call *PoCallDriver* to forward the IRP. You need not call *PoStartNextPowerIrp* because *HIDCLASS* has already done so.
5. The completion routine is called only after the bus driver completes a Set-D0 operation. Your device has now been repowered, and you can reverse the steps you performed when you removed power. Since you're potentially running at *DISPATCH_LEVEL* and in an arbitrary thread, however, you must perform these steps without blocking the current thread.

DispatchInternalControl Callback

You specify the *DispatchInternalControl* callback as if it were the dispatch routine for *IRP_MJ_INTERNAL_DEVICE_CONTROL*, by setting an array element in the driver object's *MajorFunction* table. *HIDCLASS* calls your callback routine as a subroutine in order to obtain reports and other information or to provide instructions to your minidriver. You can program this callback as if it were an ordinary IRP dispatch routine handling the control codes listed in Table 13-3.

Internal Control Code	Description
IOCTL_GET_PHYSICAL_DESCRIPTOR	Gets USB-standard physical descriptor
IOCTL_HID_GET_DEVICE_ATTRIBUTES	Returns information about device as if it were USB
IOCTL_HID_GET_DEVICE_DESCRIPTOR	Returns a USB-standard HID descriptor
IOCTL_HID_GET_FEATURE	Reads a feature report
IOCTL_HID_GET_INDEXED_STRING	Returns a USB-standard string descriptor
IOCTL_HID_GET_STRING	Returns a USB-standard string descriptor
IOCTL_HID_GET_REPORT_DESCRIPTOR	Returns a USB-standard report descriptor
IOCTL_HID_READ_REPORT	Reads a report conforming to the report descriptor
IOCTL_HID_SEND_IDLE_NOTIFICATION	Idles the device (new in Windows XP)
IOCTL_HID_SET_FEATURE	Writes a feature report
IOCTL_HID_WRITE_REPORT	Writes a report

Table 13-3. *HIDCLASS Minidriver Control Operations*

NOTE
The list of minidriver control operations in the DDK differs from the one presented here. I relied on a particular version of the source code for *HIDCLASS* in compiling this list. It's possible that the DDK documentation is based on earlier versions or on support that was originally planned but never implemented.

I'll discuss these control operations in detail in the next section of this chapter. They all share several common features, however:

- In general, *HIDCLASS* can call your *DispatchInternalControl* callback at any interrupt request level (IRQL) less than or equal to *DISPATCH_LEVEL* and in an arbitrary thread. These facts imply that your callback, and all the data objects it

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 368

uses, must be in nonpaged memory. Furthermore, you cannot block the calling thread. If you create subsidiary IRPs to communicate with your hardware, you must use asynchronous IRPs. Finally, any driver to which you send an IRP directly from this callback must be able to tolerate receiving the IRP at *DISPATCH_LEVEL*. As a point of information, the standard SERIAL.SYS driver allows IRPs at *DISPATCH_LEVEL*, and the USB bus driver allows you to send USB request blocks (URBs) at *DISPATCH_LEVEL* as well.

- Most of the control operations use *METHOD_NEITHER*, which means that the input and output data buffers are found in the stack *Parameters.DeviceIoControl.Type3InputBuffer* and the IRP *UserBuffer* fields, respectively.
- The control operations are heavily oriented toward the USB specification for HID devices. If you're writing a *HIDCLASS* minidriver, it's probably because you have either a nonstandard USB device or some other type of device altogether. You'll therefore have to fit your device into the USB model. For example, you'll have to come up with dummy vendor and product identifier values, dummy string descriptors, and so on.
- The IRPs you receive in this callback have sufficient *IO_STACK_LOCATION* slots for you to pass the IRP down the PnP stack to your own bus driver. This fact allows you to piggyback on the control IRP to carry out a device-specific function that requires an IRP.

Here's a skeleton for coding this callback function in a minidriver:

```
#pragma LOCKEDCODE

NTSTATUS DispatchInternalControl(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = PDX(fdo);
  NTSTATUS status = STATUS SUCCESS;
  ULONG info = 0;

  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
  ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
  ULONG code =  stack->Parameters.DeviceIoControl.IoControlCode;
  PVOID buffer = Irp->UserBuffer;

  switch (code)
    {
  case IOCTL HID XXX:

    break;
  default:
    status = STATUS NOT SUPPORTED;
    break;
    }

  if (status != STATUS PENDING)
    CompleteRequest(Irp, status, info);
  return status;
  }
```

13.3.3 Internal IOCTL Interface

The major interface between *HIDCLASS* and a minidriver is through the *DispatchInternalControl* callback summarized at the end of the preceding section on callback routines. In this section, I'll describe how to perform each of the control operations, in the order in which *HIDCLASS* normally presents them. Note that *HIDCLASS* doesn't invoke this callback at all until after the minidriver successfully completes an *IRP_MN_START_DEVICE* request.

IOCTL_HID_GET_DEVICE_ATTRIBUTES

HIDCLASS sends an *IOCTL_HID_GET_DEVICE_ATTRIBUTES* request as part of its processing of the *IRP_MN_START_DEVICE* request and conceivably, at other times, to obtain information that a USB device records in its device descriptor. The *UserBuffer* field of the IRP points to an instance of the following structure, which you should complete:

```
typedef struct  HID DEVICE ATTRIBUTES {
  ULONG  Size;
  USHORT VendorID;
  USHORT ProductID;
  USHORT VersionNumber;
  USHORT Reserved[11];
  } HID_DEVICE_ATTRIBUTES, * PHID_DEVICE_ATTRIBUTES;
```


IPR2024-00292
KASPERSKY EX 1008 Page 369

For example, you can complete the structure in the context of the skeletal *DispatchInternalControl* routine shown earlier:

```
case IOCTL_HID_GET_DEVICE_ATTRIBUTES:
  {
  if (cbout < sizeof(HID_DEVICE_ATTRIBUTES))
    {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
    }
  #define p ((PHID_DEVICE_ATTRIBUTES) buffer)
  RtlZeroMemory(p, sizeof(HID_DEVICE_ATTRIBUTES));
  p->Size = sizeof(HID_DEVICE_ATTRIBUTES);
  p->VendorID = 0;
  p->ProductID = 0;
  p->VersionNumber = 1;
  #undef p
  info = sizeof(HID_DEVICE_ATTRIBUTES);
  break;
  }
```

If your device is simply a nonstandard USB device, it's obvious which values you should supply for the *VendorID*, *ProductID*, and *VersionNumber* fields of this structure: the *idVendor*, *idProduct*, and *bcdDevice* fields from the real device descriptor. If your device isn't a USB device, you have to come up with dummy values. I used 0, 0, and 1, respectively, in this code fragment, and those choices will suffice for every type of HID device except a joystick. For a joystick device, you need to pick unique values that match what you specify in the OEM registry subkey you create for the joystick. I have no advice about how to pick those values.

Opening a HID Collection in User Mode

Opening a HID collection handle in user mode is simplicity itself if you assign unique values to the *VendorID* and *ProductID* fields of the *HID_DEVICE_ATTRIBUTES* structure. Suppose, for example, that your company owns USB vendor ID 0x1234 and that you have assigned product ID 0x5678 to your device. You'll use those values when answering the *IOCTL_HID_GET_DEVICE_ATTRIBUTES* request.

An MFC application that uses the *CDeviceList* class mentioned in Chapter 2 can open a handle to one of the collections exposed by your driver with code like the following (see the TEST program accompanying the HIDFAKE sample driver):

```
HANDLE CtestDlg::FindFakeDevice()
  {
  GUID hidguid;
1
  HidD_GetHidGuid(&hidguid);
  CDeviceList devlist(hidguid);
  int ndevices = devlist.Initialize();
2
  for (int i = 0; i < ndevices; ++i)
    {
3
    HANDLE h = CreateFile(devlist.m_list[i].m_linkname, 0,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
      OPEN_EXISTING, 0, NULL);

    if (h == INVALID_HANDLE_VALUE)
      continue;
3
    HIDD_ATTRIBUTES attr = {sizeof(HIDD_ATTRIBUTES)};
4
    BOOLEAN okay = HidD_GetAttributes(h, &attr);
    CloseHandle(h);

    if (!okay)
      continue;

5
    if (attr.VendorID != HIDFAKE_VID ||
      attr.ProductID != HIDFAKE_PID)
      continue;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 370

```
6
    return CreateFile(devlist.m_list[i].m_linkname,
      GENERIC_READ | GENERIC_WRITE, 0, NULL,
      OPEN_EXISTING, 0, NULL);
    }

  return INVALID_HANDLE_VALUE;
  }
```

1. Use *HidD_GetHidGuid* to determine the interface globally unique identifier (GUID) for HID devices.
2. We enumerate all HID devices. In practice, the enumeration doesn't include standard devices such as mice and keyboards.
3. Opening a handle this way (with no access rights and allowing full sharing) allows us to issue queries. Unlike most device drivers, *HIDCLASS* pays close attention to the access rights and sharing attributes specified with *IRP_MJ_CREATE*, and we're taking advantage of the flexibility that that behavior creates by opening a handle to a device that might not actually be accessible with a normal open.
4. *HidD_GetAttributes* returns an attribute structure derived from the *HID_DEVICE_ATTRIBUTES* filled in by the minidriver.
5. This is the crucial statement in this example. If the device and product ID match what we're looking for, we'll quit scanning devices and open a real handle to this one.

This call to *CreateFile* opens an exclusive, nonoverlapped handle for reading and writing. This is the right action for HIDFAKE's test applet to perform. You may have different requirements for sharing, access rights, and overlapped I/O. Note that the call to *CreateFile* might fail, even if the earlier one succeeded, if another application has snuck in to open a handle.

(continue)

Your application logic can get more complicated if your device has more than one top-level collection or if you need to provide for more than one instance of your hardware.

An entirely different approach to reading input from HID devices is available through the WM_INPUT window message and related user-mode APIs. This facility is new with Windows XP, and I didn't try it out. Maybe for the third edition....

It isn't possible to open a handle to a mouse or keyboard collection because the system input thread has opened these devices exclusively. Furthermore, these devices don't appear in the enumeration of the HID interface GUID. (*HIDCLASS* doesn't advertise the HID interface GUID for keyboard and mice so as to prevent some random user-mode program from opening those devices before the system's own raw input thread can do so.) It does you no good to register a private interface GUID for your device because *HIDCLASS* will fail an *IRP_MJ_CREATE* directed to the main device object. Consequently, there is no way to communicate with a custom mouse or keyboard driver using standard methods.

13.3.4 IOCTL_HID_GET_DEVICE_DESCRIPTOR

HIDCLASS sends an *IOCTL_HID_GET_DEVICE_DESCRIPTOR* request as part of its processing of the *IRP_MN_START_DEVICE* request and conceivably, at other times, in order to obtain a description of the device's HID characteristics. The *UserBuffer* field of the IRP points to an instance of a USB-standard HID descriptor structure, which you should complete:

```
typedef struct  HID_DESCRIPTOR {
  UCHAR   bLength;
  UCHAR   bDescriptorType;
  USHORT  bcdHID;
  UCHAR   bCountry;
  UCHAR   bNumDescriptors;
  struct  HID_DESCRIPTOR_DESC_LIST {
    UCHAR   bReportType;
    USHORT  wReportLength;
    } DescriptorList [1];
  } HID_DESCRIPTOR, * PHID_DESCRIPTOR;
```

Notwithstanding the apparent generality of this structure, *HIDCLASS* currently reserves sufficient space for only one element in the *DescriptorList* array, and it has to be the report descriptor. The Microsoft developers recommend that you nevertheless inspect the size of the output buffer and arrange your code to copy any additional descriptors—such as a physical

IPR2024-00292
KASPERSKY EX 1008 Page 371

descriptor—that you might have.

Your job in the minidriver is to fill in the descriptor structure as if you were a USB-standard HID device. For example:

```
case IOCTL_HID_GET_DEVICE_DESCRIPTOR:
  {
  #define p ((PHID_DESCRIPTOR) buffer)
  if (cbout < sizeof(HID_DESCRIPTOR))
    {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
    }
  RtlZeroMemory(p, sizeof(HID_DESCRIPTOR));
  p->bLength = sizeof(HID_DESCRIPTOR);
  p->bDescriptorType = HID_HID_DESCRIPTOR_TYPE;
  p->bcdHID = HID_REVISION;
  p->bCountry = 0;
  p->bNumDescriptors = 1;
  p->DescriptorList[0].bReportType = HID_REPORT_DESCRIPTOR_TYPE;
  p->DescriptorList[0].wReportLength = sizeof(ReportDescriptor);
  #undef p
  info = sizeof(HID_DESCRIPTOR);
  break;
  }
```

The only aspect of this code that isn't going to be the same from one driver to the next is the length you specify for the *wReportLength* member of the single *DescriptorList* entry you provide. This value should be the length of whatever real or dummy report descriptor you'll deliver in response to the *IOCTL_HID_GET_REPORT_DESCRIPTOR* request.

NOTE
The *bCountry* code in the HID descriptor is the language, if any, to which the device is localized. According to section 6.2.1 of the HID specification, this value is entirely optional. If you were imitating a keyboard with localized keycaps, for example, you might specify a nonzero value for this field.

IOCTL_HID_GET_REPORT_DESCRIPTOR

HIDCLASS sends an *IOCTL_HID_GET_REPORT_DESCRIPTOR* request as part of its processing of the *IRP_MN_START_DEVICE* request and conceivably, at other times, in order to obtain a USB-standard HID report descriptor. The *UserBuffer* field of the IRP points to a buffer as large as you indicated would be necessary in your reply to an *IOCTL_HID_GET_DEVICE_DESCRIPTOR* request.

Suppose you have a static data area named *ReportDescriptor* that contains a report descriptor in standard format. You could handle this request this way:

```
case IOCTL_HID_GET_REPORT_DESCRIPTOR:
  {
  if (cbout < sizeof(ReportDescriptor))
    {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
    }
  RtlCopyMemory(buffer, ReportDescriptor,
    sizeof(ReportDescriptor));
  info = sizeof(ReportDescriptor);
  break;
  }
```

Your first step in building the report descriptor is to design the report layout. The USB specification for HID devices makes it seem that you're pretty much free to design any sort of report you want, with the implication that Windows will somehow just figure out what to do with the resulting data. In my experience, however, you really don't have that much freedom. The Windows components that process keyboard and mouse input are used to receiving reports that meet certain expectations. Applications, such as games, differ greatly in their ability to decode joystick reports. I've also found that the HIDPARSE driver, which *HIDCLASS* uses to parse a HID descriptor, is rather fussy about which apparently conforming descriptors it will actually accept. Consequently, my advice is to closely mimic an existing Microsoft device when designing reports for common devices.

One of the options when you save your work in the HID Tool is to create a C-language header file, like this one (corresponding to the descriptor shown in Figure 13-3):

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 372

```
char ReportDescriptor[64] = {
  0x05, 0x05,   // USAGE PAGE (Gaming Controls)
  0x09, 0x03,   // USAGE (Gun Device )
  0xa1, 0x01,   // COLLECTION (Application)
  0xa1, 0x02,   //   COLLECTION (Logical)
  0x85, 0x01,   //     REPORT ID (1)
  0x05, 0x09,   //     USAGE PAGE (Button)
  0x09, 0x01,   //     USAGE (Button 1)
  0x15, 0x00,   //     LOGICAL MINIMUM (0)
  0x25, 0x01,   //     LOGICAL MAXIMUM (1)
  0x75, 0x01,   //     REPORT SIZE (1)
  0x95, 0x01,   //     REPORT COUNT (1)
  0x81, 0x02,   //     INPUT (Data,Var,Abs)
  0x75, 0x07,   //     REPORT SIZE (7)
  0x81, 0x03,   //     INPUT (Cnst,Var,Abs)
  0xc0,         //   END COLLECTION
  0xa1, 0x02,   //   COLLECTION (Logical)
  0x85, 0x02,   //     REPORT ID (2)
  0x05, 0x01,   //     USAGE PAGE (Generic Desktop)
  0x09, 0x30,   //     USAGE (X)
  0x25, 0xff,   //     LOGICAL MAXIMUM (-1)
  0x75, 0x20,   //     REPORT SIZE (32)
  0xb1, 0x02,   //     FEATURE (Data,Var,Abs)
  0xc0,         //   END COLLECTION
  0xa1, 0x02,   //   COLLECTION (Logical)
  0x85, 0x03,   //     REPORT ID (3)
  0x05, 0x09,   //     USAGE PAGE (Button)
  0x09, 0x01,   //     USAGE (Button 1)
  0x25, 0x01,   //     LOGICAL MAXIMUM (1)
  0x75, 0x01,   //     REPORT SIZE (1)
  0xb1, 0x02,   //     FEATURE (Data,Var,Abs)
  0x75, 0x07,   //     REPORT SIZE (7)
  0xb1, 0x03,   //     FEATURE (Cnst,Var,Abs)
  0xc0,         //   END COLLECTION
  0xc0          // END COLLECTION
};
```

You can simply include this header file in your driver to define the *ReportDescriptor* you return from *IOCTL_HID_GET_REPORT_DESCRIPTOR*.

IOCTL_HID_READ_REPORT

IOCTL_HID_READ_REPORT is the workhorse operation of a *HIDCLASS* minidriver. *HIDCLASS* issues this request to obtain a raw HID report. *HIDCLASS* uses the raw reports to satisfy *IRP_MJ_READ* and *IOCTL_HID_GET_INPUT_REPORT* requests issued to it from higher-level components, including user-mode applications that call *ReadFile*, *HidD_GetInputReport*, *IDirectInputDevice8::GetDeviceData*, or *IDirectInputDevice8::Poll*.

A minidriver can employ any of several strategies to provide reports:

- If your device is a programmed I/O (PIO) type of device attached to a traditional bus such as Peripheral Component Interconnect (PCI), you can perhaps perform hardware abstraction layer (HAL) function calls to derive data for a structured report. You'll then immediately complete the *IOCTL_HID_READ_REPORT* request.
- If your device attaches to a traditional bus and uses a hardware interrupt to notify the host when report data is available, you need to implement a scheme for satisfying requests with reports when they become available. Using an interlocked list lets you read and save report data in an interrupt service routine (ISR). Other schemes would require your ISR to queue a deferred procedure call (DPC), which would then read and save report data.
- If your device is a nonstandard USB device, you can perhaps submit a single URB to derive data from which you can compose a structured report. You can piggyback the URB on the *IOCTL_HID_READ_REPORT* request if your device's raw report is no bigger than the report *HIDCLASS* is expecting. In this case, your dispatch routine will presumably allocate memory for the URB from nonpaged memory, install a completion routine, and forward the IRP down the PnP stack to the USB bus driver. Your completion routine will free the URB, reformat the report data and set *IoStatus.Information* equal to the size of the reformatted report, and return *STATUS_SUCCESS* to allow the IRP to complete.
- In still other situations, you may need to pend the *IOCTL_HID_READ_REPORT* request while you perform one or more I/O operations to fetch raw data from your device, which you then reformat into the desired report packet. With this design, you have the usual issues associated with caching a pointer to the *IOCTL_HID_READ_REPORT* request in a cancel-safe way and with cancelling whatever subsidiary IRPs you create.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 373

No matter what scheme you devise, your driver will implement this IRP by filling the *UserBuffer* buffer with a report. For example:

```
case IOCTL_HID_READ_REPORT:
  {
  if (cbout < <size of report>)
    {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
    }
  <obtain report data>
  RtlCopyMemory(buffer, <report>, <size of report>);
  info = <size of report>;
  break;
  }
```

Bear in mind that if your report descriptor includes more than one report, the report data you return to *HIDCLASS* begins with a 1-byte report identifier.

IOCTL_HID_WRITE_REPORT

HIDCLASS issues the *IOCTL_HID_WRITE_REPORT* request to service *IRP_MJ_WRITE* and *IOCTL_HID_SET_OUTPUT_REPORT* requests issued from a higher-level component, such as a user-mode application that calls *WriteFile*, *HidD_SetOutputReport*, or *IDirectInputDevice8::SendDeviceData*.

Output reports are commonly used to set indicators of various kinds, such as LEDs and panel displays. Your job in a minidriver is to transmit the output report data to the device or to simulate the operation of a HID device receiving such a report by some means. USB devices implement the class-specific control-pipe command *Set_Report_Request* (or else they define an interrupt-out endpoint) for receiving output reports, but your architecture may call for a different approach.

Unlike other *HIDCLASS* internal control operations, *IOCTL_HID_WRITE_REPORT* uses *METHOD_BUFFERED*. This means that the *AssociatedIrp.SystemBuffer* field of the IRP contains the address of the output data and that the *Parameters.DeviceIoControl.OutputBufferLength* field of the *IO_STACK_LOCATION* contains its length.

IOCTL_HID_GET_FEATURE and IOCTL_HID_SET_FEATURE

HIDCLASS issues the *IOCTL_HID_GET_FEATURE* and *IOCTL_HID_SET_FEATURE* requests in order to read or write a so-called *feature report*. An application might trigger these requests by calling *HidD_GetFeature* or *HidD_SetFeature*, respectively.

You can embed feature reports within a report descriptor. According to the HID specification, feature reports are useful for getting and setting configuration information rather than for polling the device on a regular basis. With a USB-standard device, the driver uses *Get_Report_Request* and *Set_Report_Request* class-specific commands to implement this functionality. In the minidriver for some other kind of HID device, you need to provide some sort of analogue if your report descriptor includes feature reports.

These I/O control (IOCTL) operations are also the way Microsoft would prefer you perform out-of-band communication between an application and a HID minidriver. Bear in mind that *HIDCLASS* doesn't allow anyone to open a handle to the device itself (handles may be opened only to top-level collections) and fails any nonstandard control operations that it happens to receive. Without resorting to sleazy methods, as to which I won't say anything, there is actually no other way for an application and a *HIDCLASS* minidriver to communicate.

The "output" buffer for this request is an instance of the following structure:

```
typedef struct  HID_XFER_PACKET {
  PUCHAR reportBuffer;
  ULONG  reportBufferLen;
  UCHAR  reportId;
  } HID_XFER_PACKET, *PHID_XFER_PACKET;
```

HIDCLASS uses the same structure for both *GET_FEATURE* and *SET_FEATURE* requests, and it sets *Irp->UserBuffer* to point to it in both cases too. In fact, the only difference between the two requests is that the length of the structure (a constant) is in the *InputBufferLength* parameter for *SET_FEATURE* and in the *OutputBufferLength* parameter for *GET_FEATURE*. (You won't even care about this difference. Since *HIDCLASS* is a trusted kernel-mode caller, there's no particular reason to validate the length of this parameter structure.)

Your job when handling one of these requests is to decode the *reportId* value, which designates one of the feature reports your driver supports. For a *GET_FEATURE* request, you should place up to *reportBufferLen* bytes of data in the *reportBuffer* buffer and complete the IRP with *IoStatus.Information* set to the number of bytes you copy. For a *SET_FEATURE* request, you should extract *reportBufferLen* bytes of data from the *reportBuffer* buffer.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 374

Here's a skeleton for handling these two requests:

```
case IOCTL_HID_GET_FEATURE:
  {
  #define p ((PHID_XFER_PACKET) buffer)
  switch (p->reportId)
    {
  case FEATURE_CODE_XX:
    if (p->reportBufferLen < sizeof(FEATURE_REPORT_XX))
      {
      status = STATUS_BUFFER_TOO_SMALL;
      break;
      }
    RtlCopyMemory(p->reportBuffer, FeatureReportXx,
      sizeof(FEATURE_REPORT_XX));
    info = sizeof(FEATURE_REPORT_XX);
    break;
    .
    .
    }
  break;
  #undef p
  }

case IOCTL_HID_SET_FEATURE:
  {
  #define p ((PHID_XFER_PACKET) buffer)
  switch (p->reportId)
    {
  case FEATURE_CODE_YY:
    if (p->reportBufferLen > sizeof(FEATURE_REPORT_YY))
      {
      status = STATUS_INVALID_PARAMETER;
      break;
      }
    RtlCopyMemory(FeatureReportYy, p->reportBuffer,
      p->reportBufferLen);
    break;
    .
    .
    }
  break;
  #undef p
  }
```

CAUTION
When your driver supports feature reports, you'll normally be using report identification bytes to identify the different feature reports and your input and output reports. In that case, the *reportBuffer* buffer begins with a single-byte identifier, which will equal the *reportId* value in the *HID_XFER_PACKET* structure—*HIDCLASS* makes that so. The count in *reportBufferLen* includes the identifier byte. When you don't use report identifiers, however, *reportId* will be 0, *reportBuffer* won't have room for an identifier byte, and the *reportBufferLen* count won't include an identifier byte. This arrangement is true even though the caller of *HidD_GetFeature* or *HidD_SetFeature* supplies a buffer that *does* include a zero identification byte. To put it another way, you copy the actual feature report data beginning at *reportBuffer + 1* if you're using report identifiers but beginning at *reportBuffer* if you're not using report identifiers.

In these fragments, *FEATURE_CODE_XX* and *FEATURE_CODE_YY* are placeholders for manifest constants that you would define to correspond to feature report identifiers in your device's scheme. *FEATURE_REPORT_XX* and *FEATURE_REPORT_YY* are structures that include an identifier byte and the actual report data, and *FeatureReportXx* and *FeatureReportYy* are instances of those structures.

IOCTL_GET_PHYSICAL_DESCRIPTOR

HIDCLASS sends an *IOCTL_GET_PHYSICAL_DESCRIPTOR* when a higher-level component requests the physical descriptor for a device. Physical descriptors provide information about which part or parts of the body activate one or more controls on a device. If you have a nonstandard HID device for which this request is relevant, you'll need to support the request by returning a dummy descriptor meeting the HID specification, Section 6.2.3. For example:

IPR2024-00292
KASPERSKY EX 1008 Page 375

```
case IOCTL GET PHYSICAL DESCRIPTOR:
  {
  if (cbout < sizeof(PhysicalDescriptor))
    {
    status = STATUS BUFFER TOO SMALL;
    break;
    }
  PUCHAR p =
    (PUCHAR) MmGetSystemAddressForMdlSafe(Irp->MdlAddress);
  if (!p)
    {
    status = STATUS INSUFFICIENT RESOURCES;
    break;
    }
  RtlCopyMemory(p,
    PhysicalDescriptor, sizeof(PhysicalDescriptor));
  info = sizeof(PhysicalDescriptor);
  break;
  }
```

Note that this IOCTL uses *METHOD_OUT_DIRECT* instead of *METHOD_NEITHER*.

In addition, bear in mind the following statement in the HID specification: "Physical descriptors are entirely optional. They add complexity and offer very little in return for most devices."

IOCTL_HID_GET_STRING

HIDCLASS sends an *IOCTL_HID_GET_STRING* to retrieve a string describing the manufacturer, product, or serial number of a device. A user-mode application can trigger this IOCTL by calling *HidD_GetManufacturerString*, *HidD_GetProductString*, or *HidD_GetSerialNumberString*. The strings correspond to optional strings specified by the device descriptor of a standard USB device. A parameter to the operation indicates which string you should return, and in which language.

A skeleton for handling this control operation is as follows:

```
case IOCTL HID GET STRING:
  {
  #define p ((PUSHORT) \
    &stack->Parameters.DeviceIoControl.Type3InputBuffer)
  USHORT istring = p[0];
  LANGID langid = p[1];
  #undef p

  PWCHAR string = NULL;
  switch (istring)
    {
  case HID STRING ID IMANUFACTURER:
    string = <manufacturer name>;
    break;
  case HID STRING ID IPRODUCT:
    string = <product name>;
    break;
  case HID STRING ID ISERIALNUMBER:
    string = <serial number>;
    break;
    }
  if (!string)
    {
    status = STATUS INVALID PARAMETER;
    break;
    }
  ULONG lstring = wcslen(string);
  if (cbout < lstring * sizeof(WCHAR))
    {
    status = STATUS INVALID BUFFER SIZE;
    break;
    }
  RtlCopyMemory(buffer, string, lstring * sizeof(WCHAR));
  info = lstring * sizeof(WCHAR);
  if (cbout >= info + sizeof(WCHAR))
```


IPR2024-00292
KASPERSKY EX 1008 Page 376

```
    {
    ((PWCHAR) buffer)[lstring] = UNICODE NULL;
    info += sizeof(WCHAR);
    }
  break;
  }
```

Some of the key points about this code fragment are these:

- Like most other minidriver IOCTL requests, this one uses *METHOD_NEITHER*. In the context of the *DispatchInternalControl* callback presented earlier, *buffer* is the output buffer to be filled.
- The serial number, if there is one, should be unique for each device.
- The minidriver should fail the request with *STATUS_INVALID_PARAMETER* if an invalid string index appears and *STATUS_INVALID_BUFFER_SIZE* if a buffer is supplied but is too small to hold the entire string.
- The minidriver returns the whole string or nothing. It appends a null terminator to the string if the output buffer is big enough.

The DDK doesn't specify what to do if the requested language isn't one of the ones your device or minidriver happens to support. I would suggest failing the request with *STATUS_DEVICE_DATA_ERROR* to mimic what a real USB device is supposed to do. If, however, the unsupported language is 0x0409 (American English), I recommend returning a default string of some kind—perhaps even the first language in your list of supported languages—because *HIDCLASS* always uses 0x0409 for the language id parameter in Windows XP and earlier versions of the system.

IOCTL_HID_GET_INDEXED_STRING

HIDCLASS sends an *IOCTL_HID_GET_INDEXED_STRING* to retrieve a string whose USB-standard index and language identifiers are specified. A user-mode program can trigger this IOCTL by calling *HidD_GetIndexedString*. You handle this request much like *IOCTL_HID_GET_STRING* except for these two points:

- This control operation uses a curious mix of two buffering methods: the input data containing the string index and the language id is in *stack->Parameters.DeviceIoControl.Type3InputBuffer* (as would be true of a *METHOD_NEITHER* request), and the output buffer is described by the memory descriptor list (MDL) at *Irp->MdlAddress*, as would be true of a *METHOD_OUT_DIRECT* request.
- The string index in the low-order 16 bits of the *Type3InputBuffer* is a USB-standard string index instead of a constant such as *HID_STRING_ID_IMANUFACTURER*.

The purpose of this request is to allow applications to retrieve string values corresponding to string usages in a HID report. USB devices may make up to 255 string values accessible in this way. With a nonstandard USB device or a non-USB device, your minidriver needs to provide an analogue if the report descriptor contains string usages.

IOCTL_HID_SEND_IDLE_NOTIFICATION_REQUEST

HIDCLASS sends an *IOCTL_HID_SEND_IDLE_NOTIFICATION_REQUEST* to power down an idle device. With a real USB device, this request dovetails with the USB selective suspend feature discussed in the preceding chapter.

The input buffer for this *METHOD_NEITHER* request is an instance of the following structure:

```
typedef struct  HID SUBMIT IDLE NOTIFICATION CALLBACK INFO {
  HID SEND IDLE CALLBACK IdleCallback;
  PVOID IdleContext;
  } HID SUBMIT IDLE NOTIFICATION CALLBACK INFO,
    *PHID SUBMIT IDLE NOTIFICATION CALLBACK INFO;
where HID SEND IDLE CALLBACK is declared as follows:
typedef void (*HID_IDLE_CALLBACK)(PVOID Context);
```

Note that this structure is identical in layout and meaning to the one used with USB selective suspend. In fact, if your device happened to be a USB device, you could just forward the IRP down the stack after changing the function code:

```
case IOCTL HID SEND IDLE NOTIFICATION REQUEST:
  {
  IoCopyCurrentIrpStackLocationToNext(Irp);
  stack = IoGetNextIrpStackLocation(Irp);
  stack->Parameters.DeviceIoControl.IoControlCode =
    IOCTL INTERNAL USB SUBMIT IDLE NOTIFICATION;
  return IoCallDriver(pdx->LowerDeviceObject, Irp);
  }
```


IPR2024-00292
KASPERSKY EX 1008 Page 377

```
If your device is not a USB device, however, you should call back right away to HIDCLASS
and complete the IRP, as shown here:
case IOCTL HID SEND IDLE NOTIFICATION REQUEST:
  {
  PHID SUBMIT IDLE NOTIFICATION CALLBACK INFO p =
    (PHID SUBMIT IDLE NOTIFICATION CALLBACK INFO)
    stack->Parameters.DeviceIoControl.Type3InputBuffer;
  (*p->IdleCallback)(p->IdleContext);
  break;
  }
```

Calling back tells *HIDCLASS* that it can immediately power the device down.

13.4 Windows 98/Me Compatibility Notes

The footprint of history is heavy on the HID architecture in Windows 98/Me because of the necessity of supporting legacy methods of handling keyboards, mice, and joysticks. In general, it's been my experience that each new attempt to port a working HID minidriver from Windows XP has provided new opportunities for learning and reverse engineering. I plainly don't know everything there is to know in this area, but I'll describe two situations I encountered and how I dealt with them.

13.4.1 Handling *IRP_MN_QUERY_ID*

If you're writing a minidriver for fake hardware, such as HIDFAKE attempts to do, you must special-case the handling of *IRP_MN_QUERY_ID*. Left to itself, the root enumerator succeeds this IRP but provides a *NULL* list of identifiers. *HIDCLASS* then induces a crash deep in the Virtual Machine Manager (VMM). Here's the code in HIDFAKE to cope with this problem:

```
NTSTATUS DispatchPnp(PDEVICE OBJECT fdo, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = PDX(fdo);
  PIO STACK LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  if (win98
    && stack->MinorFunction == IRP MN QUERY ID
    && !NT SUCCESS(Irp->IoStatus.Status))
    {
    PWCHAR idstring;

    switch (stack->Parameters.QueryId.IdType)
      {

    case BusQueryInstanceID:
      idstring = L"0000";
      break;

    case BusQueryDeviceID:
      idstring = L"ROOT\\*WCO0D01";
      break;

    case BusQueryHardwareIDs:
      idstring = L"*WCO0D01";
      break;

    default:
      return CompleteRequest(Irp);
      }

    ULONG nchars = wcslen(idstring);
    ULONG size = (nchars + 2) * sizeof(WCHAR);
    PWCHAR id = (PWCHAR) ExAllocatePool(PagedPool, size);
    if (!id)
      return CompleteRequest(Irp, STATUS INSUFFICIENT RESOURCES);
    wcscpy(id, idstring);
    id[nchars + 1] = 0;
    return CompleteRequest(Irp, STATUS SUCCESS, (ULONG PTR) id);
    }

  return GenericDispatchPnp(PDX(fdo)->pgx, Irp);
  }
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 378

(Note the use of two override versions of my *CompleteRequest* helper here.)

Actually, you need to do something special with the *BusQueryHardwareIDs* even in Windows XP because *HIDCLASS* omits the creation of compatible IDs if the bus driver fails the request, which the root enumerator will do. You can't create a fake device of one of the standard classes unless you know this trick.

13.4.2 Joysticks

In Windows, there are two different code paths for interpreting the axis and button information for a joystick. One code path relies on the HID descriptor. Another relies on settings in the OEM registry key. If these code paths don't produce consistent results, you end up with a nonfunctional joystick in that every attempt to read its position generates an error. I know of no way except trial and error to get past this problem. The one time I had to do it for a client, I ended up writing an elaborate HID simulator that we could quickly program to create new joystick devices with specified attributes. After some number of iterations, we ended up with a working device. I'd be hard pressed to reproduce the effort.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 379

Chapter 14

Specialized Topics

In the first eight chapters, I described most of the features of a full-blown WDM driver suitable for any random sort of hardware device. But you should understand a few more general-purpose techniques, and I'll describe them in this chapter. In the chapter's first section, I'll explain how to log errors for eventual viewing by a system administrator. In addition, I'll provide instructions about how to create your own system threads, how to queue work items for execution within the context of existing system threads, and how to set up watchdog timers for unresponsive devices.

14.1 Logging Errors

In the discussions of error handling up until now, I've been concerned only with detecting (and propagating) status codes and with doing various things in the checked build to help debug problems that show up as errors. Even in the free build of a driver, however, some errors are serious enough that we want to be sure the system administrator knows about them. For example, maybe a disk driver discovers that the disk's physical surface has an unusually large number of bad sectors. Or maybe a driver is encountering unexpectedly frequent data errors or some sort of difficulty configuring or starting the device.

To deal with these types of situations, a driver can write an entry to the system error log. The Event Viewer applet—one of the administrative tools on a MicrosoftWindows XP system—can later display this entry so that an administrator can learn about the problem. See Figure 14-1 for an illustration of the Event Viewer. Another way to indicate sudden errors is by signaling a Windows Management Instrumentation (WMI) event. I'll discuss event logging in this section; WMI was the subject of Chapter 10.

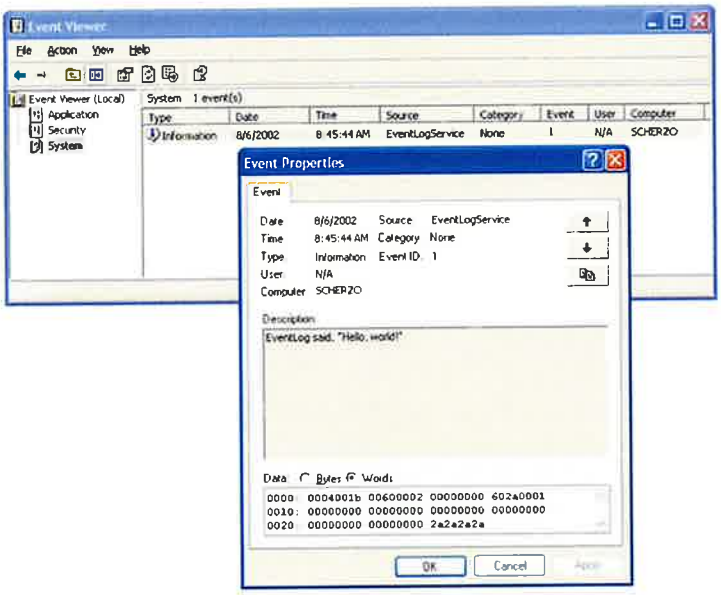


Figure 14-1. *The Windows XP Event Viewer.*

Production of an administrative report from the error log involves the steps diagrammed in Figure 14-2. A driver uses the kernel-mode service function *IoWriteErrorLogEntry* to send an *error log packet* data structure to the event logger service. The packet contains a numeric code instead of message text. As time permits, the event logger writes packets to a logging file on disk. Later the Event Viewer combines the packets in the log file with message text drawn from a collection of *message files* to produce the report. The message files are ordinary 32-bit DLLs containing text appropriate to all possible logged events in the local language.

Your job as a driver author is to create appropriate error log packets when noteworthy events occur. As a practical matter, you'll probably also be the person who has to build the message file in at least one natural language. I'll describe both aspects of error logging in the next two sections.

IPR2024-00292
KASPERSKY EX 1008 Page 381

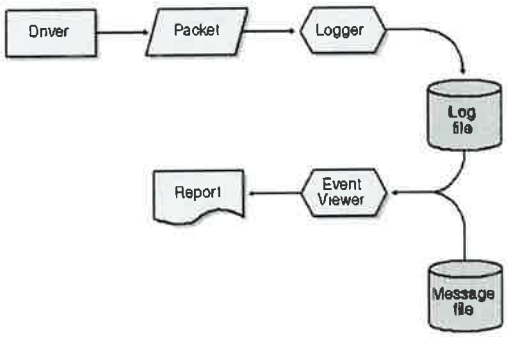


Figure 14-2. *Overview of event logging and reporting.*

14.1.1 Creating an Error Log Packet

To log an error, a driver creates an *IO_ERROR_LOG_PACKET* data structure and sends it to the kernel-mode logger. The packet is a variable-length structure—see Figure 14-3—with a fixed-size header containing general information about the event you're logging. *ErrorCode* indicates the event you're logging; it correlates with the message text file I'll describe shortly. After the fixed header comes an array of doublewords named *DumpData*, which contains *DumpDataSize* bytes of data that the Event Viewer will display in hexadecimal notation when asked for detailed information about this event. The size is in bytes even though the array is declared as consisting of 32-bit integers. After *DumpData*, the packet can contain zero or more null-terminated Unicode strings that will end up being substituted into the formatted message text by the Event Viewer. The string area begins StringOffset bytes from the start of the packet and contains NumberOfStrings strings.

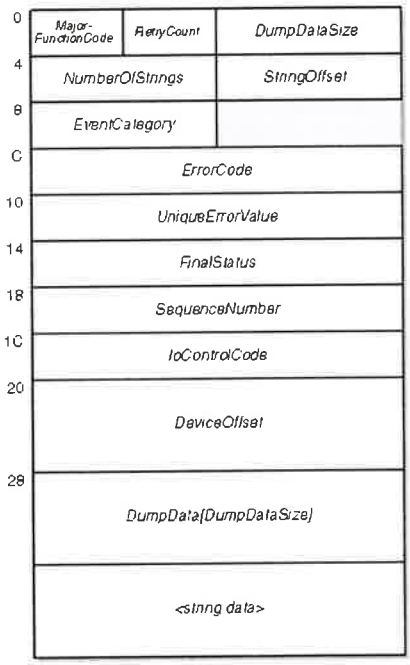


Figure 14-3. *The IO_ERROR_LOG_PACKET structure.*

You don't have to fill in any of the fixed-header members besides the ones I just mentioned. But they add, perhaps, diagnostic utility to the log entries, which might help you track down problems.

Since the logging packet is of variable length, your first job is to determine how much memory is needed for the packet you want to create. Add the size of the fixed header to the number of bytes of *DumpData* to the number of bytes occupied by the substitution strings (including their null terminators). For example, the following code fragment, taken from the EVENTLOG sample in the companion content, allocates an error log packet big enough to hold 4 bytes of dump data plus a single string:

```
VOID LogEvent(NTSTATUS code, PDEVICE_OBJECT fdo)
  {
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 382

```
PWSTR myname = L"EventLog";
ULONG packetlen = (wcslen(myname) + 1) * sizeof(WCHAR)
  + sizeof(IO ERROR LOG PACKET) + 4;
if (packetlen > ERROR LOG MAXIMUM SIZE)
  return;
PIO ERROR LOG PACKET p = (PIO ERROR LOG PACKET)
  IoAllocateErrorLogEntry(fdo, (UCHAR) packetlen);
if (!p)
  return;
:
}
```

One trap for the unwary in this sequence is that error log packets have a maximum length of 152 bytes, the value of *ERROR_LOG_MAXIMUM_SIZE*. Furthermore, the size argument to *IoAllocateErrorLogEntry* is a *UCHAR*, which is only 8 bits wide. It would be very easy to ask for a packet that was, say, 400 bytes long and be embarrassed when only 144 bytes get allocated. (400 is 0x190; 144 is 0x90, which is what you'd get after the truncation to 8 bits.)

Notice that the first argument to *IoAllocateErrorLogEntry* is the address of a device object. The name, if any, of that device object will appear in eventual log entries in place of the %1 substitution escape, which I will discuss more in the next section.

This code fragment also illustrates the action you should take in response to a problem allocating a log entry: *none*. It's not considered an error if you can't log some other error, so you don't want to fail any I/O request packet, generate a bug check, or do anything else that will cause your processing to terminate. In fact, you'll notice that this *LogEvent* helper function is *VOID* because no programmer should be concerned enough about whether it succeeds or fails to have to put a check in his or her code.

After successfully allocating the log packet, your next job is to initialize the structure and hand off control of it to the logger. For example:

```
:
memset(p, 0, sizeof(IO ERROR LOG PACKET));
p->ErrorCode = code;

p->DumpDataSize = 4;
p->DumpData[0] = <whatever>;

p->StringOffset = sizeof(IO ERROR LOG PACKET) + p->DumpDataSize;
p->NumberOfStrings = 1;
wcscpy((PWSTR) ((PUCHAR) p + p->StringOffset), myname);

IoWriteErrorLogEntry(p);
}
```

When logging a device error, you'd fill in more of the fields in the header than just the error code. For information about these other fields, consult the *IoAllocateErrorLogEntry* function in the DDK documentation.

Error log entries remain in system memory until the logger service gets around to writing them to disk. A system crash might intervene and prevent them from showing up later when you run the Event Viewer. If you're running a kernel debugger at the time, or if you have a crash dump, the *!errorlog* command will let you see the queued entries.

14.1.2 Creating a Message File

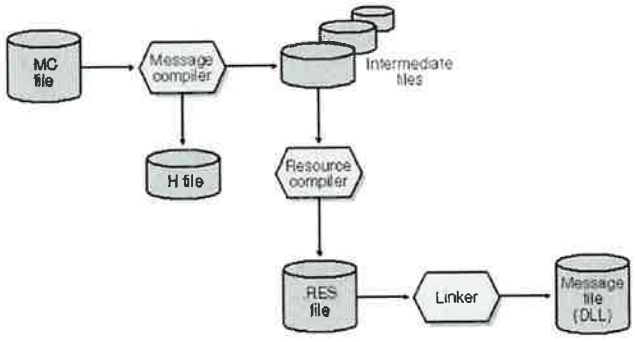


Figure 14-4. *Creating a message file.*

IPR2024-00292
KASPERSKY EX 1008 Page 383

The Event Viewer uses the *ErrorCode* in an error packet to locate the text of an appropriate message in one of the message files associated with your driver. A message file is just a DLL with a message resource containing text in one or more natural languages. Since a WDM driver uses the same executable file format as a DLL, the message file for your private messages can just be your driver file itself. I'll give you an introduction here to building a message file. You can find additional information on MSDN and in James D. Murray's *Windows NT Event Logging* (O'Reilly & Associates, 1998) at pages 125-57.

Figure 14-4 illustrates the process by which you attach message text to your driver. You begin by creating a message source file with the file extension MC. Your build script uses the message compiler (MC.EXE) to translate the messages. One of the outputs of the message compiler is a header file containing symbolic constants for your messages; you include that file in your driver, and the constants end up being the *ErrorCode* values for the events you log. The other outputs from the message compiler are a set of intermediate files containing message text in one or more natural languages and a resource script file (.RC) that lists those intermediate files. Your build script goes on to compile the resource file and to specify the translated resources as input to the linkage editor. At the end of the build, your driver contains the message resources required to support the Event Viewer.

The following is an example of a simple message source file. (This code is part of the EVENTLOG sample program.)

1
```
MessageIdTypedef = NTSTATUS
```

2
```
SeverityNames = (
  Success         = 0x0:STATUS SEVERITY SUCCESS
  Informational   = 0x1:STATUS SEVERITY INFORMATIONAL
  Warning         = 0x2:STATUS SEVERITY WARNING
  Error           = 0x3:STATUS SEVERITY ERROR
  )
```

3
```
FacilityNames = (
  System      = 0x0
  Eventlog    = 0x2A:FACILITY EVENTLOG ERROR CODE
  )
```

4
```
LanguageNames = (
  English     = 0x0409:msg00001
  German      = 0x0407:msg00002
  French      = 0x040C:msg00003
  Spanish     = 0x040A:msg00004
  )
```

5
```
MessageId = 0x0001
Facility = Eventlog
Severity = Informational
SymbolicName = EVENTLOG MSG TEST
```

6
```
Language = English
%2 said, "Hello, world!"
.
Language = German
%2 hat gesagt, «Wir sind nicht mehr in Kansas!»
.
Language = French
%2 a dit, «Mon chien a mangé mon devoir!»
.
Language = Spanish
%2 habló, ¡La lluvia en España permanece principalmente en el llano!
.
```

1. The *MessageIdTypedef* statement allows you to specify a symbol that will appear as a cast operator in the definition of each of the message identifier constants generated by this message file. For example, later we'll define a message with the symbolic name *EVENTLOG_MSG_TEST*. The presence of the *MessageIdTypedef* statement causes the header file generated by the message compiler to define this symbol as *((NTSTATUS)0x602A0001L)*.

2. The *SeverityNames* statement allows you to define your own names for the four possible severity codes. The names on the left side of the equal signs (*Success*, *Informational*, and so on) appear in the definition of messages elsewhere in this very file. The symbol after the colon ends up being defined—in the header output file—as equal to the number before the

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 384

colon. For example, *#define STATUS_SEVERITY_SUCCESS 0x0*.

3. The *FacilityNames* statement allows you to define your own names for the facility codes that will be included in the message identifier definitions. Here we've said we'll use the name *Eventlog* in *Facility* statements later. The message compiler generates the statement *#define FACILITY_EVENTLOG_ERROR_CODE 0x2A* as a result of the third line of the *FacilityNames* statement.

4. The *LanguageNames* statement allows you to define your own names for the languages into which you've translated your messages. Here we've said we'll use the name English elsewhere in the file when we mean to specify *LANGID 0x0409*, which is Standard English in the normal Microsoft Windows NT scheme of languages. The name after the colon is the name of the intermediate binary file that receives the compiled messages for this particular language.

5. Each individual message definition contains some header statements followed by the text of the message in each of the languages supported by this message source file. The *MessageId* statement can specify an absolute number, as in this example, or it can specify a delta from the last message (such as *MessageId = +1*). You specify the facility code and severity by using names defined at the start of the message source file. You also specify, with the *SymbolicName* statement, a symbolic name for this message. The message compiler will define this symbol in the header file it generates.

6. For each language you specified in the *LanguageNames* statement, you have a message text definition like this one. It begins with a *Language* statement that uses one of the language names you defined. Text for the message follows. Each message text definition ends with a line containing just a period.

Within the message texts, you can indicate by means of a percent symbol followed by an integer the places where you want string substitution to occur. *%1* refers to the name of the device object that generated the message. That name is an implicit parameter when you create an error log entry; you don't have to specify it directly. *%2*, *%3*, and so on correspond to the first, second, and so on Unicode strings you append to the log entry. In the example we've been following, *%2* will be replaced by *EventLog* because we put that string in our error packet.

This way of indicating substitution is especially useful in that you're free to put strings in the text in whatever order is appropriate for the language you're dealing with. So if your message text read "The %1 %2 fox jumped over the %3 dog" in English, it might read "Der %3 Hund wurde vom %1 %2 Fuchs übergesprungen" in German. (This is a silly example, of course. If the driver supplied "quick", "brown", and "lazy" for the substitution strings, they'd appear in English in all displayed versions of the message. But I think you get the point I'm trying to make about word order.)

The Event Viewer can't find your message file without a little bit of help in the form of some registry entries. A key named *EventLog* resides in the services branch of the Windows NT registry—that is, the collection of subkeys below *HKLM\System\CurrentControlSet\Services*. Each driver or other service that logs events has its own subkey below that. Each service-specific subkey has values named *EventMessageFile* and *TypesSupported*. The *EventMessageFile* value is a *REG_SZ* or *REG_EXPAND_SZ* type that names all of the message files that the Event Viewer might need to access to format the messages your driver generates. This value would have a data string like "%SystemRoot%\System32\iologmsg.dll; %SystemRoot%\System32\Drivers\EventLog.sys". IOLOGMSG.DLL contains the text of all the standard NTSTATUS.H codes, by the way. Consult the following sidebar for some tantalizing hints about how to automatically set these registry entries when you install your driver. The *TypesSupported* value should just be a *REG_DWORD* type equaling 7 to indicate that your driver can generate all possible events—that is, errors, warnings, and informational messages. (The fact that you even need to specify this value seems like a historical artifact of some kind.)

A Practical Note About Message Files

Two practical facts about putting message resources into your driver are difficult to discover: how precisely you make your build script compile your messages, and how you convince the system's hardware installer to put the necessary entries in the registry so the Event Viewer will find your messages.

Like the other sample programs in this book, the EVENTLOG sample is based on a Microsoft Visual C++ 6.0 project file. I modified the project definition to include a custom build step for EVENTLOG.MC and to include the resulting RC file in the build. If you open the project settings, you'll see what I mean. It's even easier if you use the DDK BUILD utility, with which you can just list your MC file as one of the SOURCES.

Later in this book (in Chapter 15), I'll discuss the general topic of how you use an INF file to install drivers. To see how you specify your message file in an INF file, take a look at DEVICE.INF in the EVENTLOG project directory and, specifically, at its *AddService* statement. You'll see that the *AddService* line points to an *EventLogLogging* section that, in turn, uses the *AddReg* statement to point to an *EventLogAddReg* section. The latter section adds *EventMessageFile* and *TypesSupported* values to the service-specific subkey of the event logger service.

14.2 System Threads

In all the device drivers considered so far in the book, we haven't been overly concerned about the thread context in which our driver subroutines have executed. Much of the time, our subroutines run in an arbitrary thread context, which means we can't block and can't directly access user-mode virtual memory. Some devices are difficult to program when faced with the first of these constraints.

IPR2024-00292
KASPERSKY EX 1008 Page 385

Some devices are best handled by *polling*. A device that can't asynchronously interrupt the CPU, for example, needs to be interrogated from time to time to check its state. In other cases, the natural way to program the device might be to perform an operation in steps with waits in between. A floppy disk driver, for example, goes through a series of steps to perform an operation. In general, the driver has to command the drive to spin up to speed, wait for the spin-up to occur, commence the transfer, wait a short while, and then spin the drive back down. You can design a driver that operates as a finite state machine to allow a callback function to properly sequence operations. It would be much easier, though, if you could just insert event and timer waits at the appropriate spots of a straight-line program.

Dealing with situations that require you to periodically interrogate a device is easy with the help of a *system thread* belonging to the driver. A system thread is a thread that operates within the overall umbrella of a process belonging to the operating system as a whole. I'll be talking exclusively about system threads that execute solely in kernel mode. In the next section, I'll describe the mechanism by which you create and destroy your own system threads. Then I'll give an example of how to use a system thread to manage a polled input device.

14.2.1 Creating and Terminating System Threads

To launch a system thread, you call *PsCreateSystemThread*. One of the arguments to this service function is the address of a *thread procedure* that acts as the main program for the new thread. When the thread procedure is going to terminate the thread, it calls *PsTerminateSystemThread*, which does not return. Generally speaking, you need to provide a way for a PnP event to tell the thread to terminate and to wait for the termination to occur. Combining all these factors, you'll end up with code that performs the functions of these three subroutines:

```
typedef struct  DEVICE EXTENSION {
  .
1
  KEVENT evKill;
  PKTHREAD thread;
  };

NTSTATUS StartThread(PDEVICE EXTENSION pdx)
  {
  NTSTATUS status;
  HANDLE hthread;
2
  OBJECT ATTRIBUTES oa;
  InitializeObjectAttributes(&oa, NULL, OBJ KERNEL HANDLE,
    NULL, NULL);
3
  status = PsCreateSystemThread(&hthread, THREAD ALL ACCESS,
    &oa, NULL, NULL, (PKSTART ROUTINE) ThreadProc, pdx);
  if (!NT SUCCESS(status))
    return status;
4
  ObReferenceObjectByHandle(hthread, THREAD ALL ACCESS, NULL,
    KernelMode, (PVOID*) &pdx->thread, NULL);
5
  ZwClose(hthread);
  return STATUS SUCCESS;
  }

VOID StopThread(PDEVICE EXTENSION pdx)
  {
6
  KeSetEvent(&pdx->evKill, 0, FALSE);
7
  KeWaitForSingleObject(pdx->thread, Executive,
    KernelMode, FALSE, NULL);
8
  ObDereferenceObject(pdx->thread);
  }

VOID ThreadProc(PDEVICE EXTENSION pdx)
  {
  .
9
  KeWaitForXxx(<at least pdx->evKill>);
  .
10
  PsTerminateSystemThread(STATUS_SUCCESS);
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 386

```
}
```

1. Declare a *KEVENT* named *evKill* in the device extension to provide a way for a PnP event to signal the thread to terminate. Initialize the event in your *AddDevice* function.

2. It's important to create the thread using the *OBJ_KERNEL_HANDLE*. If you happen to be running in the context of a user thread, failing to do so exposes you briefly to allowing the application to close the handle from user mode.

3. This statement launches the new thread. The return value for a successful call is a thread handle that appears at the location pointed to by the first argument. The second argument specifies the access rights you require to the thread; *THREAD_ALL_ACCESS* is the appropriate value to supply here. The next two arguments pertain to threads that are part of user-mode processes and should be *NULL* when a WDM driver calls this function. The next-to-last argument (*ThreadProc*) designates the main program for the thread. The last argument (*pdx*) is a context argument that will be the one and only argument to the thread procedure.

4. To wait for the thread to terminate, you need the address of the underlying *KTHREAD* object instead of the handle you get back from *PsCreateSystemThread*. This call to *ObReferenceObjectByHandle* gives you that address.

5. We don't actually need the handle once we have the address of the *KTHREAD*, so we call *ZwClose* to close that handle.

6. A routine such as *StopDevice*—which performs the device-specific part of *IRP_MN_STOP_DEVICE* in my scheme of driver modularization—can call *StopThread* to halt the system thread. The first step is to set the *evKill* event.

7. This call illustrates how to wait for the thread to finish. A kernel thread object is one of the dispatcher objects on which you can wait. It assumes the signalled state when the thread finally finishes. In Windows XP, you always perform this wait to avoid the embarrassment of having your driver's image unmapped while one of your system threads executes the last few instructions of its shutdown processing. That is, don't just wait for a special "kill acknowledgment" event that the thread sets just before it exits—the thread has to execute *PsTerminateSystemThread* before your driver can safely unload. *Refer also to an important Windows 98/Me compatibility note ("Waiting for System Threads to Finish") at the end of this chapter.*

8. This call to *ObDereferenceObject* balances the call to *ObReferenceObjectByHandle* that we made when we created the thread in the first place. It's necessary to allow the Object Manager to release the memory used by the *KTHREAD* object that formerly described our thread.

9. The thread procedure will contain miscellaneous logic that depends on the exact goal you're trying to accomplish. If you block while waiting for some external event, you should call *KeWaitForMultipleObjects* and specify the *evKill* event as one of the objects.

10. When you detect that *evKill* has been signalled, you call the *PsTerminateSystemThread* function, which terminates the thread. Consequently, it doesn't return. Note that you can't terminate a system thread except by calling this function in the context of the thread itself.

14.2.2 Using a System Thread for Device Polling

If you have to write a driver for a device that can't interrupt the CPU to demand service, a system thread devoted to polling the device may be the way to go. I'll show you one way to use a system thread for this purpose. This example is based on a hypothetical device with two input ports. One port acts as a control port; it delivers a 0 byte when no input data is ready and a 1 byte when input data is ready. The other port delivers a single byte of data and resets the control port.

In the sample I'll show you, we spawn the system thread when we process the *IRP_MN_START_DEVICE* request. We terminate the thread when we receive a Plug and Play request such as *IRP_MN_STOP_DEVICE* or *IRP_MN_REMOVE_DEVICE* that requires us to release our I/O resources. The thread spends most of its time blocked. When the *StartIo* routine begins to process an *IRP_MJ_READ* request, it sets an event that the polling thread has been waiting for. The polling thread then enters a loop to service the request. In the loop, the polling thread first blocks for a fixed polling interval. After the interval expires, the thread reads the control port. If the control port is 1, the thread reads a data byte. The thread then repeats the loop until the request is satisfied, whereupon it goes back to sleep until *StartIo* receives another request.

The thread routine in the POLLING sample is as follows:

```
VOID PollingThreadRoutine(PDEVICE_EXTENSION pdx)
  {
  NTSTATUS status;
1
  KTIMER timer;
  KeInitializeTimerEx(&timer, SynchronizationTimer);
2
  PVOID mainevents[] = {
    (PVOID) &pdx->evKill,
    (PVOID) &pdx->evRequest,
```


IPR2024-00292
KASPERSKY EX 1008 Page 387

```
    };

  PVOID pollevents[] = {
    (PVOID) &pdx->evKill,
    (PVOID) &timer,
    };

  C_ASSERT(arraysize(mainevents) <= THREAD_WAIT_OBJECTS);
  C_ASSERT(arraysize(pollevents) <= THREAD_WAIT_OBJECTS);

  BOOLEAN kill = FALSE;

3
  while (!kill)
    {    // until told to quit
4
    status = KeWaitForMultipleObjects(arraysize(mainevents),
      mainevents, WaitAny, Executive, KernelMode, FALSE,
      NULL, NULL);
    if (!NT_SUCCESS(status) || status == STATUS_WAIT_0)
      break;
    ULONG numxfer = 0;
    LARGE_INTEGER duetime = {0};
    #define POLLING_INTERVAL 500
5
    KeSetTimerEx(&timer, duetime, POLLING_INTERVAL, NULL);

    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);

6
    while (TRUE)
      {    // read next byte
7
      if (Irp->Cancel)
        {
        status = STATUS_CANCELLED;
        break;
        }
      status = AreRequestsBeingAborted(&pdx->dqReadWrite);
      if (!status)
        break;
8
      status = KeWaitForMultipleObjects(arraysize(pollevents),
        pollevents, WaitAny, Executive, KernelMode, FALSE,
        NULL, NULL);
      if (!NT_SUCCESS(status))
        {
        kill = TRUE;
        break;
        {
      if (status == STATUS_WAIT_0)
        {
        status = STATUS_DELETE_PENDING;
        kill = TRUE;
        break;
        }
9
      if (pdx->nbytes)
        {
        if (READ_PORT_UCHAR(pdx->portbase) == 1)
          {
          *pdx->buffer++ = READ_PORT_UCHAR(pdx->portbase + 1);
          --pdx->nbytes;
          ++numxfer;
          }
        }
      if (!pdx->nbytes)
        break;
      }    // read next byte
    KeCancelTimer(&timer);
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 388

```
        StartNextPacket(&pdx->dqReadWrite, pdx->DeviceObject);
        if (Irp)
          {
          IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
          CompleteRequest(Irp, STATUS_SUCCESS, numxfer);
          }
        }    // until told to quit

    PsTerminateSystemThread(STATUS_SUCCESS);
    }
```

1. We'll be using this kernel timer later to control the frequency with which we poll the device. This timer has to be a *SynchronizationTimer* so that it automatically goes into the not-signalled state each time we wait for it to expire.

2. We'll call *KeWaitForMultipleObjects* twice in this function to block the polling thread until something of note happens. These two arrays provide the addresses of the dispatcher objects on which we'll wait. The *C_ASSERT* statements verify that we're waiting for few enough events that we can use the array of wait blocks that's built into the thread object.

3. This loop terminates when an error occurs or when *evKill* becomes signalled. We'll then terminate the entire polling thread.

4. This wait terminates when either *evKill* or *evRequest* becomes signalled. Our *StartIo* routine will signal *evRequest* to indicate that an IRP exists for us to service.

5. The call to *KeSetTimerEx* starts our timer counting. This is a repetitive timer that expires once based on the due time and periodically thereafter. We're specifying a 0 due time, which will cause us to poll the device immediately. The *POLLING_INTERVAL* is measured in milliseconds.

6. This inner loop terminates when either the kill event becomes signalled or we're done with the current IRP.

7. While we're going about our business in this loop, the current IRP might get cancelled, or we might receive a PnP or power IRP that requires us to abort this IRP.

8. In this call to *KeWaitForMultipleObjects*, we take advantage of the fact that a kernel timer acts like an event object. The call finishes when either *evKill* is signalled (meaning we should terminate the polling thread altogether) or the timer expires (meaning we should execute another poll).

9. This is the actual polling step in this driver. We read the control port, whose address is the base port address given to us by the PnP Manager. If the value indicates that data is available, we read the data port.

The *StartIo* routine that works with this polling routine first sets the *buffer* and *nbytes* fields in the device extension; you saw the polling routine use them to sequence through an input request. Then it sets the *evRequest* event to wake up the polling thread.

You can organize a polling driver in other ways besides the one I just showed you. For example, you can spawn a new polling thread each time an arriving request finds the device idle. The thread services requests until the device becomes idle, whereupon it terminates. This strategy is better than the one I illustrated if long periods elapse between spurts of activity on the device, because the polling thread isn't occupying virtual memory during the long intervals of quiescence. If, however, your device is more or less continuously busy, the first strategy might be better because it avoids repeating the overhead of starting and stopping the polling thread.

You can also organize a polling driver to use a cancel-safe queue instead of a *DEVQUEUE* and *StartIo* routine, as does POLLING. It's six of one, half a dozen of the other.

14.3 Work Items

From time to time, you might wish that you could temporarily *lower* the processor's interrupt request level (IRQL) to carry out some task or another that must be done at *PASSIVE_LEVEL*. Lowering IRQL is, of course, a definite no-no. So long as you're running at or below *DISPATCH_LEVEL*, however, you can queue a *work item* to request a callback into your driver later. The callback occurs at *PASSIVE_LEVEL* in the context of a worker thread owned by the operating system. Using a work item can save you the trouble of creating your own thread that you only occasionally wake up.

NOTE
Don't hijack one of the system worker threads by scheduling a work item that takes a long time to execute. There aren't a great number of worker threads, and you can lock up the system by preventing other drivers' work items from executing.

You would ordinarily declare a context structure of some sort to tell your work item callback routine what to do. Whatever else it contains, it will need a pointer to an *IO_WORKITEM* structure, as shown here:

IPR2024-00292
KASPERSKY EX 1008 Page 389

```
typedef struct  RANDOM JUNK {
:
  PIO WORKITEM item;
  } RANDOM JUNK, *PRANDOM JUNK;
At the point where you want to queue the work item, you create an instance of the context
structure, and an IO WORKITEM:
PRANDOM JUNK stuff = (PRANDOM JUNK) ExAllocatePool(NonPagedPool,
  sizeof(RANDOM JUNK));
stuff->item = IoAllocateWorkItem(fdo);
```

where *fdo* is the address of a *DEVICE_OBJECT*, ordinarily one with which the work item is associated in some way. You now initialize the context structure and issue the following call to actually place the work item in the queue for a system worker thread:

```
IoQueueWorkItem(stuff->item, (PIO WORKITEM ROUTINE) Callback,
  QueueIdentifier, stuff);
```

QueueIdentifier can be either of these two values:

- *DelayedWorkQueue* indicates that you want your work item executed in the context of a system worker thread that executes at variable priority—that is, not at a real-time priority level.
- *CriticalWorkQueue* indicates that you want your work item executed in the context of a system worker thread that executes at a real-time priority.

You choose the delayed or the critical work queue depending on the urgency of the task you're trying to perform. Putting your item in the critical work queue will give it priority over all noncritical work in the system at the possible cost of reducing the CPU time available for other critical work. In any case, the activities you perform in your callback can always be preempted by activities that run at an elevated IRQL.

After you queue the work item, the operating system will call you back in the context of a system worker thread having the characteristics you specified as the third argument to *IoQueueWorkItem*. You'll be at IRQL *PASSIVE_LEVEL*. What you do inside the callback routine is pretty much up to you except for one requirement: you must release or otherwise reclaim the memory occupied by the work queue item. Here's a skeleton for a work-item callback routine:

```
VOID Callback(PDEVICE OBJECT fdo,PRANDOM JUNK stuff)
  {
  PAGED CODE();
:
  IoFreeWorkItem(stuff->item);
  ExFreePool(stuff);
  }
```

This callback receives a single argument (*stuff*), which is the context parameter you supplied earlier in the call to *IoQueueWorkItem*. This fragment also shows the calls to *ExFreePool* and *IoFreeWorkItem* that balance the allocations we did earlier.

In between the time you call *IoQueueWorkItem* and the time your callback routine returns, the I/O Manager owns an extra reference to the device object you specified in the original call to *IoAllocateWorkItem*. The extra reference pins your driver in memory at least until the callback routine returns. Without this protection, it would be perfectly possible for your driver to queue a work item and then unload before the callback finished executing. A bug check would then occur because the system would attempt to execute code at a suddenly invalid address. There is nothing you can do inside your own driver to totally avoid this kind of problem because you have to execute at least a return instruction to get out of your own code and back to the system.

In Windows versions prior to Windows 2000, there was a routine named *ExQueueWorkItem* and a macro named *ExInitializeWorkItem* for creating and queuing work items. These functions are now deprecated because of the driver unload problem. In fact, the Windows Hardware Quality Lab (WHQL) test suite flags calls to *ExQueueWorkItem*. This actually poses a bit of an obstacle to creating a binary-compatible driver for all WDM platforms, as discussed in "Windows 98/Me Compatibility Notes" at the end of this chapter.

On the CD The WORKITEM sample in the companion content illustrates the mechanics of using the *IoXxxWorkItem* functions discussed in the text.

14.3.1 Watchdog Timers

Some devices won't notify you when something goes wrong—they simply don't respond when you talk to them. Each device object has an associated *IO_TIMER* object that you can use to avoid indefinitely waiting for an operation to finish. While the

IPR2024-00292
KASPERSKY EX 1008 Page 390

timer is running, the I/O Manager will call a timer callback routine once a second. Within the timer callback routine, you can take steps to terminate any outstanding operations that should have finished but didn't.

You initialize the timer object at AddDevice time:

```
NTSTATUS AddDevice(...)
  {
  IoInitializeTimer(fdo, (PIO_TIMER_ROUTINE) OnTimer, pdx);
  }
```

where *fdo* is the address of your device object, *OnTimer* is the timer callback routine, and *pdx* is a context argument for the I/O Manager's calls to *OnTimer*.

You start the timer counting by calling *IoStartTimer*, and you stop it from counting by calling *IoStopTimer*. In between, your *OnTimer* routine is called once a second.

The PIOFAKE sample in the companion content illustrates one way of using the *IO_TIMER* as a watchdog. I put a *timer* member in the device extension for this fake device. I also defined a *BOOLEAN* flag to indicate when the driver is actually busy handling an IRP:

```
typedef struct  DEVICE_EXTENSION {
  LONG timer;
  BOOLEAN busy;

} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

When I process an *IRP_MJ_CREATE* after a period with no handles open to the device, I start the timer counting. When I process the *IRP_MJ_CLOSE* that closes the last handle, I stop the timer:

```
NTSTATUS DispatchCreate(...)
  {
  if (InterlockedIncrement(&pdx->handles == 1))
    {
    pdx->timer = -1;
    IoStartTimer(fdo);
    }
  }

NTSTATUS DispatchClose(...)
  {
  if (InterlockedDecrement(&pdx->handles) == 0)
    IoStopTimer(fdo);
  }
```

The *timer* cell begins life with the value -1. I set it to 10 (meaning 10 seconds) in the *StartIo* routine and again after each interrupt. Thus, I allow 10 seconds for the device to digest an output byte and to generate an interrupt that indicates readiness for the next byte. The work to be done by the *OnTimer* routine at each 1-second tick of the timer needs to be synchronized with the interrupt service routine (ISR). Consequently, I use *KeSynchronizeExecution* to call a helper routine (*CheckTimer*) at device IRQL (DIRQL) under protection of the interrupt spin lock. The timer-tick routines dovetail with the ISR and deferred procedure call (DPC) routines as shown in this excerpt:

```
VOID OnTimer(PDEVICE_OBJECT fdo, PDEVICE_EXTENSION pdx)
  {
  KeSynchronizeExecution(pdx->InterruptObject,
    (PKSYNCHRONIZE_ROUTINE) CheckTimer, pdx);
  }

VOID CheckTimer(PDEVICE_EXTENSION pdx)
  {
  if (pdx->timer <= 0 || --pdx->timer > 0)
```


IPR2024-00292
KASPERSKY EX 1008 Page 391

```
    return;
  if (!pdx->busy)
    return;
2
  PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
  if (!Irp)
    return;
3
  pdx->busy = FALSE;
  Irp->IoStatus.Status = STATUS_IO_TIMEOUT;
  Irp->IoStatus.Information = 0;
  IoRequestDpc(pdx->DeviceObject, Irp, NULL);
  }

BOOLEAN OnInterrupt(...)
  {
4
  if (!pdx->busy)
    return TRUE;
  if (!pdx->nbytes)
    {
    pdx->busy = FALSE;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = pdx->numxfer;
    IoRequestDpc(pdx->DeviceObject, Irp, NULL);
    }
5
  pdx->timer = 10;
  }

VOID DpcForIsr(...)
  {
  PIRP Irp = StartNextPacket(&pdx->dqReadWrite, fdo);
6
  IoCompleteRequest(Irp, IO_NO_INCREMENT);

  }
```

1. The timer value -1 means that no request is currently pending. The value 0 means that the current request has timed out. In either case, we don't want or need to do any more work in this routine. The second part of the *if* expression decrements the timer. If it hasn't counted down to 0 yet, we return without doing anything else.

2. This driver uses a *DEVQUEUE*, so we call the *DEVQUEUE* routine *GetCurrentIrp* to get the address of the request we're currently processing. If this value is *NULL*, the device is currently idle.

3. At this point, we've decided we want to terminate the current request because nothing has happened for 10 seconds. We request a DPC after filling in the IRP status fields. This particular status code (*STATUS_IO_TIMEOUT*) turns into a Win32 error code (*ERROR_SEM_TIMEOUT*) for which the standard error text ("The semaphore timeout period has expired") doesn't really indicate what's gone wrong. If the application that has requested this operation is under your control, you should provide a more meaningful explanation.

4. If the *busy* flag is *FALSE*, this interrupt is not expected and will be ignored. The flag might be *FALSE* if the device has generated a spurious interrupt. (PIOFAKE is for fake hardware, so the "device" is really a dialog box with an Interrupt button that you can press at a time when no test program is trying to write a string to the device.) Or else a request might have timed out, and *CheckTimer* would have cleared the flag precisely to prevent the ISR from doing anything.

5. We allow 10 seconds between interrupts.

6. Whatever requested this DPC also filled in the IRP's status fields. We therefore need to call only *IoCompleteRequest*.

The *busy* flag plays an important role in guarding against a race between the interrupt service routine (*OnInterrupt*) and the timeout routine (*CheckTimer*). The *StartIo* routine sets *busy*. One or the other of *OnInterrupt* or *CheckTimer* will clear the flag before requesting the DPC that completes the current IRP. Once either of these routines sets the flag, the other will start returning immediately until *StartIo* starts a new IRP. To properly synchronize access, all routines that touch the *busy* flag must run in synchrony with the interrupt routine. Hence the use of *KeSynchronizeExecution* to call *CheckTimer* and to call the routine (not shown here in the text) that initially sets *busy* to *TRUE*.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 392

14.4 Windows 98/Me Compatibility Notes

Some minor differences exist between Windows 98/Me and Windows XP insofar as the material discussed in this chapter goes.

14.4.1 Error Logging

Windows 98/Me doesn't implement an error-logging file or an Event Viewer. When you call *IoWriteErrorLogEntry* in Windows 98/Me, all that happens is that several lines of data appear on your debugging terminal. I find the formatting of this information unaesthetic, so I prefer simply not to use the error-logging facility under Windows 98/Me. Refer to Appendix A for suggestions about how to determine whether you're running Windows 98/Me or Windows XP.

14.4.2 Waiting for System Threads to Finish

Windows 98/Me doesn't support the use of a pointer to a thread object (a *PKTHREAD*) as an argument to *KeWaitForSingleObject* or *KeWaitForMultipleObjects*. Those support functions simply pass their object pointer arguments through to VWIN32.VXD without any sort of validity checking, and VWIN32 crashes because the thread objects don't have the structure members needed to support synchronization use.

If you need to wait for a kernel-mode thread to complete in Windows 98/Me, therefore, you'll need to have the thread signal an event just before it calls *PsTerminateSystemThread*. It's possible that signalling this event will cause the terminating thread to lose control to a thread waiting for the same event. The terminating thread will then still be alive technically, but I don't think anything awful can happen as a result in Windows 98/Me. The POLLING sample shows how to elevate the priority of the terminating thread to diminish the risk.

14.4.3 Work Items

Windows 98 and Windows Me don't export the *IoXxxWorkItem* routines. This fact doesn't pose any particular reliability problem since these systems are not likely to invalidate or overstore driver program code before a queued work item executes. If you have a driver that calls *ExQueueWorkItem* in order to run in Windows 98/Me, you can't currently pass WHQL tests, *even if you have a run-time check so as to avoid calling ExQueueWorkItem in Windows 2000 or later systems*. But if your driver calls the *IoXxxWorkItem* routines that are needed for robustness in Windows 2000 and later, Window 98 and Windows Me won't load your driver because of the unresolved import. This situation is tailor-made for the WDMSTUB.SYS solution discussed in Appendix A. WDMSTUB defines the *IoXxxWorkItem* routines so you can load your driver.

My own GENERIC.SYS also contains calls to *ExQueueWorkItem* in order to work around the fact that power IRPs must be sent and completed at *PASSIVE_LEVEL* in Windows 98 and Windows Me. By the time you read this, I hope to have persuaded WHQL to move their work item tests into the Driver Verifier so as to allow safe uses of the older routine to pass muster.

IPR2024-00292
KASPERSKY EX 1008 Page 393

Chapter 15

Distributing Device Drivers

Early in the device driver development process, it's important to devote some thought to how you'll distribute your driver and how an end user will install your driver and the hardware it serves. MicrosoftWindows XP and Microsoft Windows 98/Me use a text file with the file extension INF to control most of the activities associated with installing drivers. You provide the INF file. It goes either on a diskette or on a disc that you package with the hardware, or else Microsoft makes it available on line or on a setup disc. In the INF file, you tell the operating system which file or files to copy onto the end user's hard disk, which registry entries to add or modify, and so on.

In this chapter, I'll discuss several aspects of installing your driver. I'll discuss the important role played by the registry in driver installation and initialization. I'll lead you through the important parts of a simple INF file to help you tie together the DDK documentation about INF file syntax. I'll explain in detail the format of device identifiers used for various types of devices. Since I had to define a custom device class for all the sample "devices" used in this book, I thought it would help you to see how I did that.

Microsoft operates the Windows Hardware Quality Lab (WHQL) to help ensure the quality of the hardware and device drivers that end users purchase for use with Windows operating systems. WHQL provides a Hardware Compatibility Test kit (HCT) for use with many common classes of device. You should aim to pass these tests to qualify for Microsoft's logo licensing programs and to obtain a digital signature file that will greatly simplify installation on end user machines. I'll guide you through the process of putting together a WHQL submission in this chapter.

15.1 The Role of the Registry

The PnP Manager and setup subsystems rely heavily on four keys in the *HKEY_LOCAL_MACHINE* branch of the registry. These are called the *hardware key*, the *class key*, the *driver key*, and the *service key*. (See Figure 15-1.) To be clear, these are not the proper names of specific subkeys; they are generic names of four keys whose pathnames depend on the device to which they belong. Broadly speaking, the hardware and driver keys contain information about a single device, the class key concerns all devices of the same type, and the service key contains information about the driver. People sometimes use the name *instance key* to refer to the hardware key and *software key* to refer to the driver key. The multiplicity of names derives from the fact that Windows 95/98/Me and Windows XP were written (mostly) by different people. A fifth key, the *hardware parameters* key, might also exist; it contains nonstandard parameter information about the device.

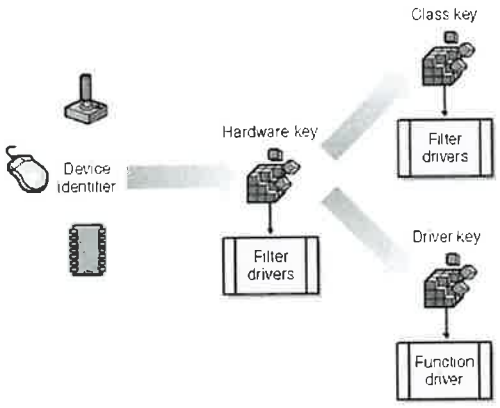


Figure 15-1. *Meaningful registry keys for a device.*

In this section, I'll describe the contents of these registry keys. You'll never alter these keys directly on an end user system. The keys and the values in them are created or maintained automatically by the setup program, the Device Manager, and the PnP Manager. I'll show you later how you can inspect and modify these values using user-mode and kernel-mode APIs. You should plan to use those APIs instead of directly tampering with the registry. In fact, even administrator accounts lack permission to write to some of these keys.

When you're developing and debugging drivers, though, and especially when you're debugging your installation procedures, you often need to directly muck about with registry settings using REGEDIT. In Windows XP, REGEDIT allows you to alter the security permissions in the registry. (In Windows 2000, you would use REGEDT32 for this purpose. In Windows 98/Me, of

IPR2024-00292
KASPERSKY EX 1008 Page 395

course, the registry isn't secured in the first place.)

15.1.1 The Hardware (Instance) Key

Device hardware keys appear in the *\System\CurrentControlSet\Enum* subkey of the local machine branch of the registry. Figure 15-2 illustrates the hardware key for a sample device (namely, the USB42 sample from Chapter 12). The subkeys on the first level below the *Enum* key correspond to the different bus enumerators in the system. The description of all past or present USB devices is in the *...\Enum\USB* subkey. I've expanded the key for the USB42 sample to show you how the device's hardware ID (vendor 0547, product 102A) has turned into the name of a key (Vid_0547&Pid_102A) and how a particular instance of the device that has that ID appears as a further subkey named 6&16f0a439&0&2. The 6&16f0a439&0&2 key is the hardware, or instance, key for this device.

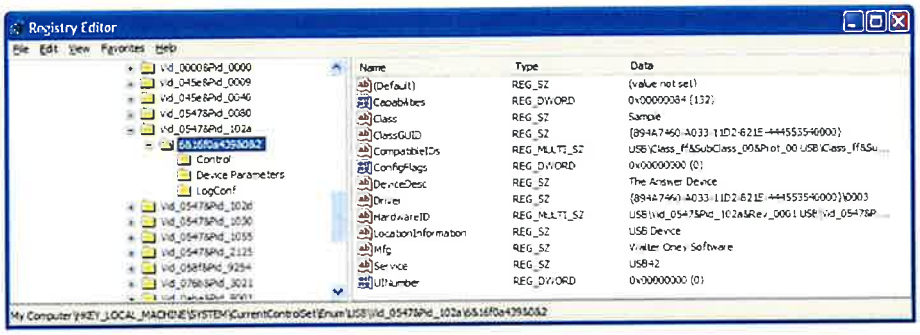


Figure 15-2. *A hardware key in the registry.*

Some of the values in the hardware key provide descriptive information that user-mode components such as the Device Manager can use. Figure 15-3 shows how the Device Manager portrays the properties of USB42. If you compare Figures 15-2 and 15-3, you'll notice some common things. In particular, the *DeviceDesc* string in the hardware key is the title of the device (unless there happens to be a *FriendlyName* property in the registry, which isn't the case here), and the *Mfg* property appears as the Manufacturer name in the property page. I'll explain later where some of the other property information comes from. I'll also explain later how you can access these properties from a user-mode application or from a WDM driver.

TIP
You can open the Device Manager from the Hardware page of My Computer properties or the System control panel applet, or from the Computer Management console under Administrative Tools. Since I use the Device Manager so frequently, I created a desktop shortcut to devmgmt.msc, which is normally in the \windows\system32 directory.

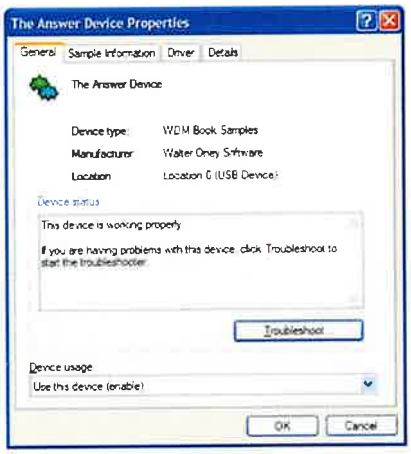


Figure 15-3. *Device Manager properties for the USB42 device.*

The hardware key also contains several values that identify the class of device to which the device belongs and the drivers for the device. *ClassGUID* is the ASCII representation of a globally unique identifier (GUID) that uniquely identifies a device setup class; in effect, it's a pointer to the class key for this device. *Class* is the name of the setup class. *Driver* names the driver key, which is a subkey of the class key. *Service* is a pointer to the service key in *HKLM\System\CurrentControlSet\Services*. Optional values (which USB42 doesn't have) named *LowerFilters* and *UpperFilters*, if present, would identify the service names for any lower or upper filter drivers.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 396

A hardware key might have overriding values named *Security*, *Exclusive*, *DeviceType*, and *DeviceCharacteristics* that force the device object the driver will create to have certain attributes. USB42 doesn't have these overrides.

Finally, the hardware key might contain a subkey named *Device Parameters*, which contains nonstandard configuration information about the hardware. See Figure 15-4. *SampleInfo*, the only property in the figure for USB42, specifies the help file for the sample driver. (The other values in the figure are artifacts of a failed run of one of the Hardware Compatibility Tests. Their presence does no harm.)

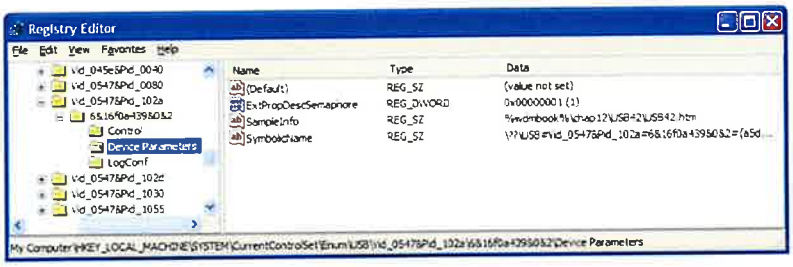


Figure 15-4. *A device parameters key in the registry.*

15.1.2 The Class Key

The class keys for all classes of device appear in the *HKLM\System\CurrentControlSet\Control\Class* key. Figure 15-5 illustrates the class key for *SAMPLE* devices, which is the class to which the USB42 sample and all the other sample drivers in this book belong. The values in this key serve these purposes:

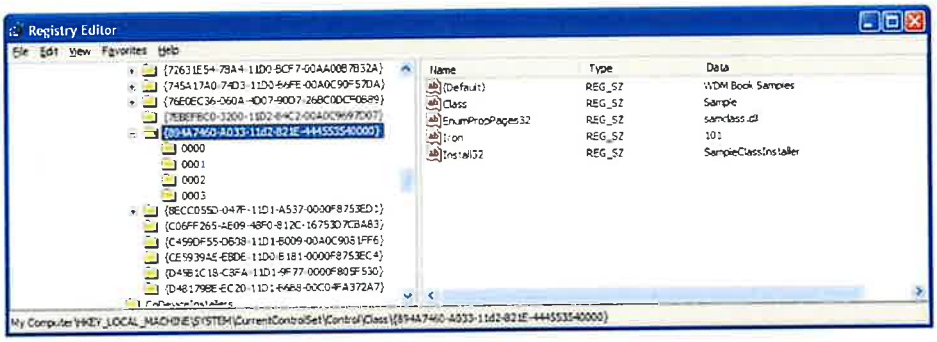


Figure 15-5. *A class key in the registry.*

- *(Default)* specifies the friendly name of the class. This is the class title that the Device Manager uses. (Refer back to Figure 15-3.)
- *Class* is the name of the class. The class name and the GUID go together here and in the hardware keys for devices belonging to this class.
- *EnumPropPages32* specifies a property-page provider DLL that provides custom property pages for the Device Manager to use when displaying properties for this class of device. The provider for this class, samclass.dll, presents the page labeled "Sample Information." In general, this value can include a DLL name and the name of an entry point. If the entry point name is omitted, as in this example, the system assumes it is *EnumPropPages*.
- *Install32* specifies the class installer DLL that the setup system uses whenever it performs setup actions on devices belonging to the class. This value can include a DLL name and the name of an entry point. If the DLL name is omitted, as in this example, the system assumes it is the same as the property-page DLL.
- *Icon* specifies the resource identifier for an icon in the class installer DLL. The Device Manager and the setup system use this icon whenever they display information about the class. The DDK suggests that an icon will be taken from the property page DLL if no class installer is present, but that was not the case in Windows 2000, and I've been in the habit of providing at least a degenerate class installer entry point just so I can have a custom icon.

The *SAMPLE* class lacks some of the optional values that might be present, such as the following:

- *NoInstallClass*, if present and not equal to 0, indicates that some enumerator will automatically detect any device belonging to this class. If the class has this attribute, the hardware wizard won't include this class in the list of device classes it presents to the end user.
- *SilentInstall*, if present and not equal to 0, causes the PnP manager to install devices of this class without presenting any

IPR2024-00292
KASPERSKY EX 1008 Page 397

dialog boxes to the end user.

- *UpperFilters* and *LowerFilters* specify service names for filter drivers. The PnP Manager loads these filters for *every* device belonging to the class. (You specify filter drivers that apply to just one device in the device's hardware key.)
- *NoDisplayClass*, if present and not equal to 0, suppresses devices of this class from the Device Manager display.

There can be a *Properties* subkey of the class key, which can contain values named *Security*, *Exclusive*, *DeviceType*, and *DeviceCharacteristics*. These values override default settings of certain device object parameters for all devices of this class. Refer to the subsection "Device Object Properties" a bit further on for information about these settings.

15.1.3 The Driver Key

Each device also has its own subkey below the class key. The name of this key (actually, the name of this key relative to *CurrentControlSet\Control\Class*) is the *Driver* value in the device's hardware key. Refer to Figure 15-6 for an illustration of the contents of this subkey, the purpose of which is to correlate all these registry entries with the INF file used to install the device and to provide a repository for driver-specific configuration information that concerns this device.

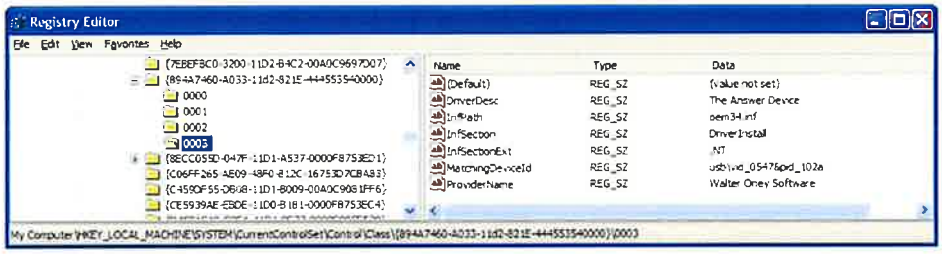


Figure 15-6. *A driver key in the registry.*

Both the driver key and the hardware parameters key can contain parameter information about a device. The difference between the two keys is a bit subtle. The DDK says that the driver key contains "driver-specific information," whereas the hardware parameters key contains "device-specific information." In both cases, the "information" in question pertains to a particular instance of the device. Microsoft's concept is that the driver-specific information would be peculiar to a given driver and not relevant to some other driver for the same hardware. I confess that this distinction pretty much escapes me, inasmuch as I'm used to thinking that any given device will have just one driver.

15.1.4 The Service (Software) Key

The last key that's important for a device driver is the service key. It indicates where the driver's executable file is on disk and contains some other parameters that govern the way the driver is loaded. Service keys appear in the *HKLM\System\CurrentControlSet\Services* key. Refer to Figure 15-7 for USB42's service key.

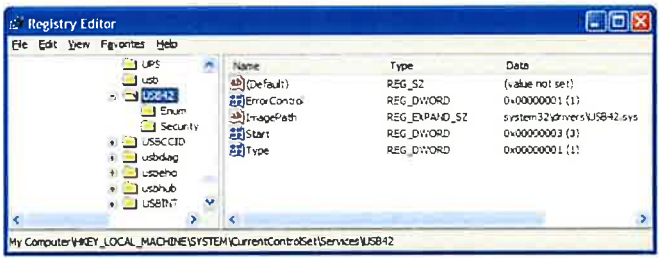


Figure 15-7. *A service key in the registry.*

I won't rehash all the possible settings in the service key, which is splendidly documented in several places, including under the heading "Service Install" in the Platform Software Development Kit (Platform SDK). In this particular case, the values have the following significance:

- *ImagePath* indicates that the executable file for the driver is named USB42.SYS and can be found in %SystemRoot%\system32\drivers. Note that the registry setting in this case is a relative pathname starting from the system root directory.
- *Type* (1) indicates that this entry describes a kernel-mode driver.
- *Start* (3) indicates that the system should load this driver when it's needed to support a newly arrived device. (This numeric value corresponds to the *SERVICE_DEMAND_START* constant in a call to *CreateService*. When applied to a

IPR2024-00292
KASPERSKY EX 1008 Page 398

kernel-mode driver, it has the meaning I just described—it's not necessary to explicitly call *StartService* or issue a NET START command to start the driver.)

- *ErrorControl* (1) indicates that a failure to load this driver should cause the system to log the error and display a message box.

15.1.5 Accessing the Registry from a Program

As I said earlier, you should plan to use a set of kernel-mode and user-mode APIs for accessing the registry rather than accessing it directly. In this subsection, I'll discuss the relevant APIs.

Accessing the Registry from a Driver

Table 15-1 shows which kernel-mode functions you should use to access information in the various registry keys I've just discussed. In most cases, you only need to use a special way to open the key. Thereafter, you use regular *ZwXxx* functions to read and write values and *ZwClose* to close the key. Refer to the examples in Chapter 3 for full details.

Registry Key	Function to Use for Access
Hardware	Read individual standard properties via IoGetDeviceProperty. You can't change these properties from a driver, and you shouldn't try to figure out the name of this key in order to open it directly.
Hardware Parameters	*IoOpenDeviceRegistryKey* (*PLUGPLAY_REGKEY_DEVICE* option)
Driver	*IoOpenDeviceRegistryKey* (*PLUGPLAY_REGKEY_DRIVER* option)
Class	No access method provided, and you shouldn't try to figure out the name of this key in order to open it directly.
Service	*ZwOpenKey* using *RegistryPath* parameter to *DriverEntry*.

To access "standard" parameters kept in the hardware key, you call *IoGetDeviceProperty* with one of the property codes listed in Table 15-2. The entries in the Source column refer to items in the INF file, which I'll discuss in the next major section of this chapter. Note that this function has still more property codes than shown in the table, but they refer to property values that the PnP Manager maintains elsewhere than in the registry.

Property Name	Value Name	Source	Description
DeviceProperty-DeviceDescription	*DeviceDesc*	First parameter in model statement	Description of device
DevicePropertyHardwareId	*HardwareID*	Second parameter in model statement	Identifies device
DevicePropertyCompatibleIDs	*CompatibleIDs*	Created by bus driver during detection	Device types that can be considered to match
DevicePropertyClassName	*Class*	*Class* parameter in Version section of INF	Name of device class
DevicePropertyClassGuid	*ClassGUID*	*ClassGuid* parameter in Version section of INF	Unique identifier of device class
DevicePropertyDriverKeyName	*Driver*	First parameter in AddService statement	Name of service key that specifies driver
DevicePropertyManufacturer	*Mfg*	Manufacturer in whose model section device was found	Name of hardware manufacturer
DevicePropertyFriendlyName	*FriendlyName*	Explicit AddReg in INF file, or class installer	"Friendly" name suitable for presentation to the user

Table 15-2. *Standard Device Properties in the Hardware Key*

For example, to retrieve the description of a device, use the following code. (See the *AddDevice* function in the DEVPROP sample.)

```
WCHAR name[256];
ULONG junk;
status = IoGetDeviceProperty(pdo,
  DevicePropertyDeviceDescription, sizeof(name), name, &junk);
KdPrint((DRIVERNAME " - AddDevice has succeeded for '%ws' device\n", name));
```

On the CD The DEVPROP sample in the companion content illustrates how to obtain standard property information from kernel mode and user mode.

IPR2024-00292
KASPERSKY EX 1008 Page 399

Accessing the Registry from User Mode

Table 15-3 lists the user-mode APIs you would use to access the registry keys we've just discussed. Most of these functions are oriented toward setup programs, including class installers and co-installers. To use them successfully, you need to have an *HDEVINFO* handle and an *SP_DEVINFO_DATA* structure that refers to the specific device you're interested in.

Registry Key	Function to Use for Access
Hardware	Read or write individual standard properties via *SetupDiGetDeviceRegistryProperty* and *SetupDiSetDeviceRegistryProperty*.
Hardware Parameters	*SetupDiOpenDevRegKey* (*DIREG_DEV* option).
Driver	*SetupDiOpenDevRegKey* (*DIREG_DRV* option).
Class	*SetupDiOpenClassRegKey*. Starting in Windows XP, read or write device object properties via *SetupDiGetClassRegistryProperty* and *SetupDiSetClassRegistryProperty*.
Service	*QueryServiceConfig*, *ChangeServiceConfig*.

Table 15-3. *User-Mode Interfaces to the Registry*

For example, within the context of an enumeration of registered interfaces using the setup APIs, you can retrieve the friendly name of a device:

```
HDEVINFO info = SetupDiGetClassDevs(...);
SP_DEVINFO_DATA did = {sizeof(SP_DEVINFO_DATA)};
SetupDiGetDeviceInterfaceDetail(info, ..., &did);
TCHAR fname[256];
SetupDiGetDeviceRegistryProperty(info, &did,
  SPDRP_FRIENDLYNAME, NULL, (PBYTE) fname,
  sizeof(fname), NULL);
```

Refer to the DDK documentation of *SetupDiGetDeviceRegistryProperty* for a list of the *SPDRP_XXX* values you can specify to retrieve the various properties.

If all you have is the symbolic name of the device, you can use the following trick:

```
LPCTSTR devname;    // <== someone gives you this
HDEVINFO info = SetupDiCreateDeviceInfoList(NULL, NULL);
SP_DEVICE_INTERFACE_DATA ifdata = {sizeof(SP_DEVICE_INTERFACE_DATA)};
SetupDiOpenDeviceInterface(info, devname, 0, &ifdata);
SP_DEVINFO_DATA did = {sizeof(SP_DEVINFO_DATA)};
SetupDiGetDeviceInterfaceDetail(info, &ifdata, NULL, 0, NULL, &did);
```

You can go on to call routines such as *SetupDiGetDeviceRegistryProperty* in the normal way at this point.

NOTE
In Windows 98 and Windows NT version 4, application programs used the *CFGMGR32* set of APIs to obtain information about devices and to interact with the PnP Manager. These APIs continue to be supported for purposes of compatibility in Windows 98/Me and Windows XP, but Microsoft discourages their use in new code. For that reason, I'm not even showing you examples of calling them.

15.1.6 Device Object Properties

As you know, you call *IoCreateDevice* to create a device object. In a WDM driver, your *AddDevice* function ordinarily creates a single device object and links it into the PnP driver stack by calling *IoAttachDeviceToDeviceStack*. Once the function driver and all filter drivers have finished these steps, the PnP Manager consults the registry to apply optional overrides to some of the settings in the device objects. The settings in question are these few:

- The security descriptor attached to the physical device object (PDO), which can be overridden by a *Security* value in the registry.
- The device type (*FILE_DEVICE_XXX*) for the PDO, which can be overridden by a *DeviceType* value in the registry.
- The device characteristics flags, which can be overridden by a *DeviceCharacteristics* value.
- The exclusivity option for the PDO, which can be overridden by an *Exclusive* value.

The PnP Manager looks first in the hardware key and then in the *Properties* subkey of the class key to find these overrides. After modifying the PDO, the PnP Manager then merges the characteristics flags from all device objects in the stack and sets

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 400

certain ones (selected by the *FILE_CHARACTERISTICS_PROPAGATED* mask in ntddk.h) to be the same in all the device objects. At the present time, the characteristics flags that are propagated are these:

- FILE_REMOVABLE_MEDIA
- FILE_READ_ONLY_DEVICE
- FILE_FLOPPY_DISKETTE
- FILE_WRITE_ONCE_MEDIA
- FILE_DEVICE_SECURE_OPEN

For the security and exclusivity overrides to be effective, none of the filter or function drivers in the PnP stack should name their device objects. They should instead use *IoRegisterDeviceInterface* as the only method of establishing a symbolic link. The registered interface approach forces the I/O and Object Managers to refer to the PDO when opening a handle to the device, thereby giving effect to these two overrides.

The overriding values come to exist in the registry in one of two ways. You can specify them with special syntax in your INF file. Alternatively, you or some standard management application can use the *SetupDiSetXxxRegistryProperty* functions to change them in the hardware or class keys.

15.2 The INF File

With the preceding background in how the PnP Manager and setup subsystem use the registry, we can begin to make sense of the INF file you supply. The purpose of an INF file is to instruct the setup subsystem how to install device-related files on an end user system and how to modify the registry. The DDK contains a very thorough discussion of INF-file syntax, which I won't repeat here. I would, however, like to provide a guide to the most commonly used portions of an INF.

An INF file contains a collection of *sections* introduced by a section name in brackets. Most sections contain a series of directives of the form "keyword = value." The INF file begins with a Version section that identifies the type of device described by entries in the file and that specifies other global characteristics of a driver installation package. The following Version section contains the minimum of required information:

```
[Version]

Signature=$CHICAGO$

Class=Sample

ClassGuid={894A7460-A033-11d2-821E-444553540000}
CatalogFile=whatever.cat
DriverVer=mm/dd/yyyy
; Copyright 2002 by Proseware, Inc.
```

Signature can be one of several magic values. I use *$ChicagoS*, which works on all WDM platforms. *Class* identifies the class of device. Table 15-4 lists the predefined classes that Windows XP already supports. *ClassGuid* uniquely identifies the device class. The DDK header file DEVGUID.H defines the GUIDs for standard device classes, and the DDK documentation entry for the Version section documents them as well. *CatalogFile* names the digital signature file that WHQL will send you after certifying your driver package; do your testing with an empty file. *DriverVer* specifies the date of the driver package and, optionally, a version number. The system uses the version information in ranking digitally signed drivers. You should also have a comment (that is, any line that starts with a semicolon) containing the word *copyright*. Note that you don't have to specify any actual copyright information, but you probably want to.

The setup subsystem will process an INF file whose Version section contains just the *Signature* and *Class* values. The other values shown in the preceding example are needed, however, if you want to have WHQL certify your driver.

I find it useful to think of the bulk of an INF file as the linear description of a tree structure. Each section is a node in the tree, and each directive is a pointer to another section. Figure 15-8 illustrates the concept.

IPR2024-00292
KASPERSKY EX 1008 Page 401

INF Class Name	Description
1394	IEEE 1394 host bus controllers (but not peripherals)
Battery	Battery devices
CDROM	CD-ROM drives, including SCSI and IDE
DiskDrive	Hard disk drives
Display	Video adapters
FDC	Floppy disk controllers
FloppyDisk	Floppy disk drives
HDC	Hard disk controllers
HIDClass	Human input devices
Image	Still-image capture devices, including cameras and scanners
Infrared	Network Driver Interface Specification (NDIS) miniport drivers for Serial-IR and Fast-IR ports
Keyboard	Keyboards
MediumChanger	SCSI media changer devices
Media	Multimedia devices, including audio, DVD, joysticks, and full-motion video capture devices
Modem	Modems
Monitor	Display monitors
Mouse	Mouse and other pointing devices
MTD	Memory technology driver for memory devices
Multifunction	Combination devices
MultiportSerial	Intelligent multiport serial cards
Net	Network adapter cards
NetClient	Network file system and print providers (client side)
NetService	Server-side support for network file systems
NetTrans	Network protocol drivers
PCMCIA	Personal Computer Memory Card International Association (PCMCIA) and CardBus host controllers (but not peripherals)
PNPPrinter	Bus-specific print class driver
Ports	Serial and parallel ports
Printer	Printers
SCSIAdapter	SCSI and RAID controllers, host bus adapter miniports, and disk array controllers
SmartCardReader	Smart card readers
System	System devices
TapeDrive	Tape drives
USB	USB host controllers and hubs (but not peripherals)
Volume	Logical storage volume drivers

Table 15-4. *Device Setup Classes*

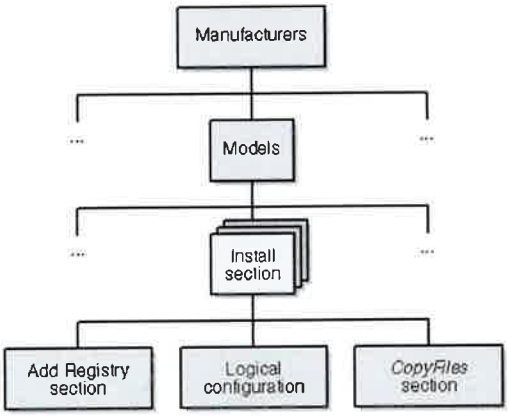


Figure 15-8. *Tree structure of an INF file.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 402

At the apex of the tree is a *Manufacturer* section that lists all the companies with hardware described in the file. For example:

```
[manufacturer]
"Walter Oney Software"=DeviceList
"Finest Organization On Earth Yet"=FOOEY

[DeviceList]
:

[FOOEY]
:
Each individual manufacturer's model section (DeviceList and FOOEY in the example)
describes one or more devices:
[DeviceList]
Description=InstallSectionName,DeviceId,CompatibleIds
```

where *Description* is a human-readable description of the device and *DeviceId* identifies a hardware device. *CompatibleIds*, if present, is a list of other device identifiers with which the same driver will work. The *InstallSectionName* parameter identifies (or points to, in my tree metaphor) another section of the INF file that contains instructions for installing the software for a particular device. An example of an entry for a single type of device might be this (drawn from the PKTDMA sample in Chapter 7):

```
[DeviceList]
"AMCC S5933 Development Board (DMA)"=DriverInstall,PCI\VEN_10E8&DEV_4750
```

The information in the Manufacturer section and in the model section (or sections) for individual manufacturers comes into play when the system needs to install a driver for a piece of hardware. A Plug and Play (PnP) device announces its presence and identity electronically. A bus driver detects it automatically and constructs a device identifier using on-board data. The system then attempts to locate preinstalled INF files that describe that particular device. INF files reside in the INF subdirectory of the Windows directory. If the system can't find a suitable INF file, it asks the end user to specify one.

A legacy device can't announce its own presence or identity. The end user therefore launches the add hardware wizard to install a legacy device and helps the wizard locate the right INF file. Key steps in this process include specifying the type of device being installed and the name of the manufacturer. See Figure 15-9.

The hardware wizard constructs dialog boxes such as the one shown in Figure 15-9 by enumerating all the INF files for a particular type of device, all of the statements in their Manufacturer sections, and all of the model statements for each of the manufacturers. You can guess that the manufacturer names that appear in the left pane of the dialog box come from the left sides of Manufacturer statements and that the model names that appear in the right pane come from the left sides of model statements.



Figure 15-9. *Selecting a device during installation.*

More About Hardware Wizard Dialog Boxes
Once the wizard is past the stage of looking for PnP devices, it builds a list of device classes and uses various *SetupDiXxx* routines from SETUPAPI.DLL to retrieve icons and descriptions. The information that SETUPAPI uses to implement these routines ultimately comes from the registry, where it was placed by entries in *ClassInstall32* sections. Not every device class will be represented in the list—the wizard will suppress information about classes that have the *NoInstallClass* attribute.

After the end user selects a device class, the wizard calls SETUPAPI functions to construct lists of manufacturers and devices as described in the text. Devices mentioned in *ExcludeFromSelect* statements will be absent from these lists.

IPR2024-00292
KASPERSKY EX 1008 Page 403

15.2.1 Install Sections

An *install section* contains the actual instructions that the installer needs to install software for a device. We've been considering the PKTDMA sample. For that device, the DeviceList model section specifies the name DriverInstall. I find it useful to think of this name as identifying an *array* of sections, one for each Windows platform. The "zero" element in this array has the base name of the section (DriverInstall). You can have platform-specific array elements whose names start with the base name and contain one of the decorations listed in Table 15-5. The device installer looks for the install section having the most specialized decoration. Suppose, for example, that you have install sections with no decoration and with the .NTx86 decoration. If you're installing into Windows XP on an Intel x86 platform, the installer will use the .NTx86 section. If you're installing into Windows 98/Me, it will use the section without a decoration.

Platform	*Install Section Decoration*
Any platform, including Windows 98/Me	[none]
Any Windows XP platform	.NT
Windows XP on Intel x86	.NTx86
Windows XP on Intel 64-bit processor	.NTIA64

Table 15-5. *Install Section Decorations for Each Platform*

Because of the search rules I just outlined, all of the INF files for my sample drivers have the no-decoration and .NTx86-decoration install sections. That makes the INF files work fine on any Intel x86 platform.

Distinguishing Among Operating Systems

For operating systems after and including Windows XP, the INF syntax provides a rather tedious way to specify different drivers for different operating systems. You create multiple model sections with unique names, and you distinguish among them by appending *TargetOsVersion* strings to the model statements in the [Manufacturer] section. Refer to the DDK documentation for the full syntax of these strings. As an example, you can specify the following in an INF file to get different drivers installed for Windows 98, Windows 2000, and Windows XP:

```
[Manufacturer]
"Walter Oney Software"=DeviceList             ; for 98/ME and 2K
"Walter Oney Software"=DeviceList,NTx86.5.1   ; XP on x86
"Walter Oney Software"=DeviceList,Ntia64.5.1 ; XP on IA64
;
```

Following this section, you have three model sections listing all the same hardware models and specifying four uniquely named install sections for each model. There is an additional .NTx86 install section for Windows 2000, making a total of four install sections per hardware model. That is, you have model sections named [DeviceList], [DeviceList.NTx86.5.1], and so on, which point to install sections named (for example) [WidgetInstall], [WidgetInstall.NTx86], [WidgetInstall.NTx86.5.1], and so on. You'll want to comment this INF heavily so you can figure out later what you were doing!

The reason I called this mechanism "tedious" is that you can't simply append an operating system-specific decoration to an install section name. You have to create a separate tree of model statements that point to uniquely named install sections. The reason there are two different schemes for identifying platform-dependent and operating system-dependent sections is historical. Windows 98/Me doesn't see the decorated section names at all. Windows 2000 will append the decorations to install section (and a few other) names but has no notion of operating system dependence. To add the operating system dependence without breaking existing INF files, Microsoft pretty much needed a different scheme.

Further along in this chapter, I'll be discussing other INF sections whose names begin with the name of the install section. If you have multiple install sections in your "array," these other sections have to include the platform-dependent decoration in their names too. For example, I'll be discussing a Services section that you use to install a description of the driver in the registry. You form the name of this section by taking the base name of the install section (for example, DriverInstall) plus the platform decoration (for example, NTx86) and adding the word *Services*, ending up with [DriverInstall.NTx86.Services].

A typical Windows XP install section will contain a *CopyFiles* directive and nothing else:

```
[DriverInstall.ntx86]
CopyFiles=DriverCopyFiles
```

This CopyFiles directive indicates that we want the installer to use the information in another INF section for copying files onto the end user hard disk. For the PKTDMA sample, the other section is named DriverCopyFiles:

```
[DriverCopyFiles]
pktdma.sys,,,2
```


IPR2024-00292
KASPERSKY EX 1008 Page 404

This section directs the installer to copy PKTDMA.SYS to the end user's hard disk.

The statements in a CopyFiles section have this general form:

```
Destination,Source,Temporary,Flags
```

Destination is the name (without any directory name) of the file as it will eventually exist on the end user system. *Source* is the name of the file as it exists on the distribution media if that name is different from the Destination name; otherwise, it's just blank as in the example. In Windows 98/Me, if you might be installing a file that will be in use at the time of installation, you specify a temporary name in the *Temporary* parameter. Windows 98/Me will rename the temporary file to the Destination name on the next reboot. It's not necessary to use this parameter for Windows XP installs because the system automatically generates temporary names.

The *Flags* parameter contains a bit mask that governs whether the system will decompress a file and how the system deals with situations in which a file by the same name already exists. The interpretation of the flags depends in part on whether the INF and the driver are part of a package that Microsoft has digitally signed after certification. Refer to the DDK documentation for a full explanation of these flags. I ordinarily specify *2* for this parameter, which basically means that the setup will be considered to have failed if this file isn't successfully copied.

The filename by itself is not sufficient to tell the installer what it needs to know to copy a file. It also needs to know which directory you want the file copied to. In addition, if you have multiple diskettes in the installation set, it needs to know which diskette contains the source file. These pieces of information come from other sections of the INF file, as suggested by Figure 15-10. In the PKTDMA example, these sections are as follows:

```
[DestinationDirs]
DefaultDestDir=10,System32\Drivers

[SourceDisksFiles]
pktdma.sys=1,objchk~1\i386,

[SourceDisksNames]
1="WDM Book Companion Disc",disk1
```

The *SourceDisksFiles* section indicates that the installer can find PKTDMA.SYS on disk number 1 of the set, in a subdirectory whose 8.3 pathname is objchk~1\i386. The *SourceDisksNames* section indicates that disk number 1 has the human-readable label WDM Book Companion Disc and contains a file named disk1 that the installer can look for to verify that the correct diskette is in the drive. Note that these section names have an interior *s* that's very easy to miss.

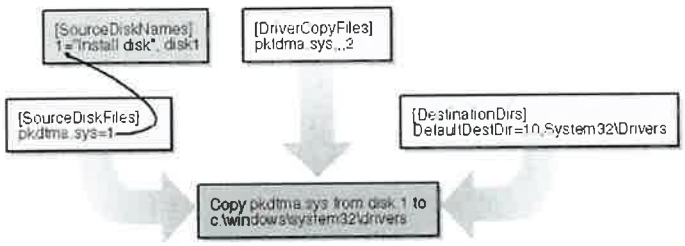


Figure 15-10. *Source and destination information for file copies.*

The *DestinationDirs* section specifies the target directories for copy operations. *DefaultDestDir* is the target directory to use for any file whose target directory isn't otherwise specified. You use a numeric code to specify the target directory because the end user might choose to install Windows XP in a directory with a nonstandard name. Please refer to the DDK documentation entry for the DestinationDirs section for a complete list of the codes—only a few of them are in common use, as follows:

- Directory 10 is the Windows directory (for example, \Windows or \Winnt).
- Directory 11 is the System directory (for example, \Windows\System or \Winnt\System32).
- Directory 12 is the Drivers directory on a Windows XP system (for example, \Winnt\System32\Drivers). Unfortunately, this number has a different meaning on a Windows 98/Me system (for example, \Windows\System\Iosubsys).

WDM drivers reside in the Drivers directory. If your CopyFiles section applies only to a Windows XP installation, you can just specify directory number 12. If you want to share a CopyFiles section between Windows 98/Me and Windows XP installs, however, I recommend that you specify "10,System32\Drivers" instead because it identifies the Drivers directory in both cases.

Defining the Driver Service

The INF syntax I've described so far is sufficient for your driver file (or files) to be copied onto the end user's hard disk. You

IPR2024-00292
KASPERSKY EX 1008 Page 405

must also arrange for the PnP Manager to know which files to load. A *Services* section accomplishes that goal, as in this example:

```
[DriverInstall.NTx86.Services]
AddService=PKTDMA,2,DriverService

[DriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\pktdma.sys
```

The 2 in the *AddService* directive indicates that the PKTDMA service will be the function driver for the device. You form the name of this section by appending the word *Services* to the name of the install section to which it applies.

The end result of these directives will be a key in the *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services* branch of the registry named PKTDMA (the first parameter in the AddService directive). It will define the service entry for the driver as a kernel-mode driver (*ServiceType* equal to 1) that should be demand-loaded by the PnP Manager (*StartType* equal to 3). Errors that occur during loading should be logged but should not by themselves prevent the system from starting (*ErrorControl* equal to 1). The executable image can be found in \Winnt\System32\Drivers\pktdma.sys (the value of *ServiceBinary*). By the way, when you look in the registry, you'll see that the name of the executable file is stored under the name *ImagePath* rather than *ServiceBinary*.

It's a good idea to make the name of the service (PKTDMA in this example) the same as the filename (PKTDMA.SYS in this example) of your driver binary file. Not only does this make it obvious which service name corresponds to which driver, but it also avoids a problem that can arise when two different service keys point to the same driver: any device that uses the same driver as a then-started device but under a different service name can't itself start.

15.2.2 Populating the Registry

Many INF sections can contain an *AddReg* verb that points to an add-registry section elsewhere in the INF file:

```
[SomeSection]
AddReg=SomeAddReg
```

The add-registry section in turn contains statements that use a positional syntax to define registry values:

```
[SomeAddReg]
key, subkey, value-name, flags, value
:
```

In this syntax template, *key* denotes either one of the standard root keys (HKCR, HKCU, HKLM, or HKU) or the context-dependent value HKR. HKR stands for "relative root key" and refers to a key that depends on where you put the *AddReg* verb that points to this add-registry section. Table 15-6 indicates what HKR means for various different *AddReg* sources.

Section Containing AddReg	Meaning of HKR
Install section—for example, [DriverInstall]	Driver key
Hardware section—for example, [DriverInstall.ntx86.hw]	Hardware parameters key
Service section—for example, [DriverInstall.ntx86.services]	Service key
[ClassInstall32] or [ClassInstall]	Class key
Event logging section (specified by optional fourth parameter in AddService directive)	Event logger's subkey for this driver
Add interface section (specified by optional third parameter in AddInterface directive)	Interface key (not discussed in this book)
Co-installer section—for example, [DriverInstall.CoInstallers]	Driver key

Table 15-6. *Meaning of HKR in an Add-Registry Section*

The *subkey* parameter in the add-registry syntax specifies an optional subkey of *key*. I rarely specify a subkey when I use HKR for the main key, but it's essential to do so if you specify a different root key. For example, if I wanted to add an entry to the *RunOnce* key, I'd have the following statement in an *AddReg* section:

```
HKLM,Software\Microsoft\Windows\CurrentVersion\RunOnce,<more stuff>
```

Incidentally, if you specify an absolute registry path like this, it doesn't matter which INF section contains the *AddReg* verb.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 406

The *value-name* parameter in an add-registry statement specifies the value you want to set. Omit this parameter (that is, put nothing between the commas) if you want to set the default value in some registry key. Within a [ClassInstall32] AddReg section, the value names *DeviceCharacteristics*, *DeviceType*, *Security*, and *Exclusive* are treated specially—they refer to values that are actually in the *Properties* subkey of the class key.

The *flags* parameter indicates the data format of the value you're setting and also specifies some optional behavior. Consult the DDK entry named "INF AddReg Directive" for complete details about these flags. I use only a few specific values in my own INF files, as follows:

- 0, which can be abbreviated just by omitting the parameter, indicates a *REG_SZ* value.
- 1 indicates a *REG_BINARY* value.
- 0x00010001 is something of a special case. It indicates a *REG_DWORD* value to Windowx XP but a *REG_BINARY* value to Windows 98/Me (which truncates the *flags* value to 16 bits and therefore never sees the 0x00010000 bit).

The *value* parameter is the value you're trying to set. For a *REG_SZ* value, you can supply a quoted string (within which text quotes are indicated by two consecutive double-quotation characters). Refer to the DDK entry entitled "INF Strings Section" for a full exposition of the syntax rules, and note that a string that contains no blanks or special characters can be coded without any quotation marks at all. Thus, the following are equivalent:

```
HKR,,NTMPDriver,,"devprop.sys"
       -or-HKR,,NTMPDriver,,devprop.sys
```

You indicate a binary value as a series of 8-bit values. This requirement is a bit of a pain in the neck when you're trying to define a *REG_DWORD* value in an INF section that you want to use both in Windows XP and in Windows 98/Me. This is how you specify a *REG_DWORD* parameter in a portable way:

```
[DriverInstall.ntx86.hw]
AddReg=HwAddReg

[DriverInstall.hw]
AddReg=HwAddReg

[HwAddReg]
HKR,,ProgrammersShoeSize,0x00010001, 0x2A, 0, 0, 0
```

Both systems infer that the registry should end up with a *REG_DWORD* equaling 42 (decimal) from the fact that 32 bits' worth of data are here.

Now that we've covered this syntax information, let's consider a few examples of registry settings in INF files.

Initializing Hardware Configuration Settings

Configuration parameters that relate to the hardware belong in the hardware parameters key. I've previously shown the whimsical example of a *ProgrammersShoeSize* parameter. A more realistic example would be a driver that services two different types of device that can't be distinguished at *AddDevice* time but that need to be treated somewhat differently. (There shouldn't be major differences between the devices because you should write two drivers if there are.)

I would code the *AddDevice* function for this driver to open the hardware parameters key and interrogate a value that I'll call *BoardType*. This will be a *REG_DWORD* value that can be 0 or 1. The code inside *AddDevice* would look like this (except that we'd really have some error checks):

```
HANDLE hkey;
status = IoOpenDeviceRegistryKey(pdo,
  PLUGPLAY REGKEY DEVICE, KEY READ, &hkey);

UNICODE STRING valname;
RtlInitUnicodeString(&valname, L"BoardType");

KEY VALUE PARTIAL INFORMATION value;
ULONG junk;
status = ZwQueryValueKey(hkey, &valname,
  KeyValuePartialInformation, &value, sizeof(value), &junk);

ULONG BoardType = *(PULONG) value.Data;

ZwClose(hkey);
```

In this fragment, I rely on the fact that the C compiler will pad the *value* variable to the next address boundary consistent with

IPR2024-00292
KASPERSKY EX 1008 Page 407

the most stringent alignment required for one of its members. In this case, this fact means that *value.Data* will really be 4 bytes long, even though it's declared as being just 1 byte long.

In the INF file, I'd probably have two different model statements and install sections, like this (note that I'm leaving out quite a few of the other sections that would be in the INF file):

```
[DeviceList]
"Widget Model A"=WidgetInstallA,...
"Widget Model B"=WidgetInstallB,...

[WidgetInstallA.ntx86.hw]
AddReg=HwAddReg.A

[WidgetInstallB.ntx86.hw]
AddReg=HwAddReg.B

[WidgetInstallA.hw]     ; for Win98/Me
AddReg=HwAddReg.A

[WidgetInstallB.hw]     ; ditto
AddReg=HwAddReg.B

[HwAddReg.A]
HKR,,BoardType,0x00010001, 0,0,0,0

[HwAddReg.B]
HKR,,BoardType,0x00010001, 1,0,0,0
```

Initializing the Driver Key

The only time I use the driver key is in the Windows 98/Me sections of my INF files, and then only because the Configuration Manager requires two values to be there to specify the WDM driver for the device. Nearly all of the samples in the companion content have the following entries in the INF file for this purpose:

```
[DriverInstall]
AddReg=DriverAddReg

[DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,whatever.sys
```

Initializing the Service Key

I rarely use the service key in a WDM driver. For the debug version of the drivers I build for my consulting clients, however, I've built a general-purpose set of tracing functions that dovetail with a Device Manager property page for setting various options. One aspect of the tracing facility is that the *DriverEntry* function needs to initialize a flag word. The only pertinent registry key that's available to *DriverEntry* is the service key, the name to which the *RegistryPath* argument points.

Part of the *DriverEntry* initialization in one of these drivers looks like this:

```
ULONG DriverTraceFlags;
KObjectAttributes oa(RegistryPath,
  OBJ CASE INSENSITIVE | OBJ KERNEL HANDLE);
HANDLE hkey;
NTSTATUS status = ZwOpenKey(&hkey, KEY READ, oa);
if (NT SUCCESS(status))
  {
  GetRegistryValue(hkey, L"DriverTraceFlags", DriverTraceFlags);
  ZwClose(hkey);
  }
```

(This initialization is part of an elaborate class library that I use so I don't have to keep retyping or cutting and pasting standardized code. You can undoubtedly figure out what's going on.)

My property page provider presents a page containing some standard and some custom options. The custom options originate in some INF-file syntax like this:

```
[DriverService]
```


IPR2024-00292
KASPERSKY EX 1008 Page 408

```
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\whatever.sys
AddReg=TraceFlags
[TraceFlags]
HKR,,CustomTraceName1,,"Board interrupts"
HKR,,CustomTraceFlag1,0x00010001, 01,00,00,00
HKR,,CustomTraceName2,,"Reads && Writes from/to board registers"
HKR,,CustomTraceFlag2,0x00010001, 02,00,00,00
```

The *AddReg* directive within the [DriverService] section is the crucial part of this example. Within the *AddReg* section it references, HKR refers to the service key.

Event Logging

You must prepare the way for your driver to log events by setting up a subkey in the event logger's own service key. You do this with syntax like the following (taken from the INF file for the EVENTLOG sample):

```
[DriverInstall.ntx86.Services]
AddService=EventLogService,2,DriverService,EventLogging
[DriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\eventlog.sys

[EventLogging]
AddReg=EventLogAddReg

[EventLogAddReg]
HKR,,EventMessageFile,0x00020000, \
  "%10%\System32\iologmsg.dll;%10%\system32\drivers\EventLog.sys"
HKR,,TypesSupported,0x00010001,7
```

Incidentally, *EventMessageFile* has the type *REG_EXPAND_SZ*.

15.2.3 Security Settings

You can specify two kinds of security settings in an INF file. One setting applies to the registry entries you create in an add-registry section:

```
[SomeSection]
AddReg=SomeAddReg

[SomeAddReg]
.
.

[SomeAddReg.security]
"<security descriptor>"
```

The other setting supplies an override for device object security descriptors, either for an entire class of device or for just one instance:

```
[ClassInstall32]
AddReg=ClassInstallAddReg

[ClassInstallAddReg]
HKR,,Security,,<security descriptor>

          -or-

[DriverInstall.hw]
AddReg=HwAddReg

[HwAddReg]
HKR,,Security,,<security descriptor>
```


IPR2024-00292
KASPERSKY EX 1008 Page 409

In each of these cases, *<security descriptor>* is a string that succinctly encodes a standard security descriptor. The security description language is documented in the Platform SDK in connection with *ConvertStringSecurity-DescriptorToSecurityDescriptor* and its complement. For example, the following security descriptor string grants all access to the system account, read/write/execute access to administrators, and read-only access to everyone else:

```
D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GR;;;WD)
(This is the SDDL DEVOBJ SYS ALL ADM RWX WORLD R string from WDMSEC.H, a header in
the .NET DDK.)
```

This string has a visual appearance that only a parser could love. Here's a guided tour:

- D:P introduces a discretionary access control list (ACL) with the protected attribute, which means it cannot be modified by access control entries inherited from parent objects.
- Each of the three parentheses-delimited strings represents one access control entry (ACE).
- (A;;GA;;;SY) specifies an access-allowed access control entry (A) with default flags (omitted second parameter). It gives GENERIC_ALL rights (GA) to the system account (SY). The objects for which you specify security descriptors in an INF file are implicit in the placement of the directive. Consequently, you would never specify an object GUID or an inherited object GUID in one of these entries (omitted fourth and fifth parameters).
- (A;;GRGWGX;;;BA) grants GENERIC_READ (GR), GENERIC_WRITE (GW), and GENERIC_EXECUTE (GX) rights to members of the built-in administrators group (BA).
- (A;;GR;;;WD) grants GENERIC_READ access to members of the Everyone (WD) group.

15.2.4 Strings and Localization

In all the examples I've shown so far, I've used literal string values in places where you might really want to present localized text or where you're going to be using the same string value over and over. You can create a table of named strings in an INF file and then use an escape convention to reference them as needed. Here's an example:

```
[DeviceList]
%DESCRIPTION%=DriverInstall,*WCO1503

[Strings]
DESCRIPTION="DEVPROP Sample"
You can provide a set of localized string sections too. For example:
[Manufacturer]
%PROSEWARE%=DeviceList
:
[Strings]
PROSEWARE=" Proseware, Inc. of North America"

[Strings.0407]
PROSEWARE=" Proseware, Inc. Deutschland GmbH"

[Strings.040C]
PROSEWARE=" Marchandise de Prose"
:
```

When installing your driver, the system will pick the correct localized versions of the strings. It will fall back on the nonlocalized [Strings] section when necessary.

15.2.5 Device Identifiers

For true Plug and Play devices, the device identifier that appears in a manufacturer's model section of an INF is very important. Plug and Play devices are those that can electronically announce their presence and identity. A bus enumerator can find these devices automatically, and it can read some sort of on-board information to find out what kind each device is. Universal serial bus (USB) devices, for example, include vendor and product identification codes in their device descriptors, and the configuration space of Peripheral Component Interconnect (PCI) devices includes vendor and product codes.

NOTE
The lists of device identifier strings that appear here now also appear in the DDK, even including the examples I used in these lists for the first edition. I nevertheless thought it worthwhile to keep this material in the second edition for ease of reference.

When an enumerator detects a device, it constructs a list of device identification strings. One entry in the list is a complete

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 410

identification of the device. This entry will end up naming the hardware key in the registry. Additional entries in the list are "compatible" identifiers. The PnP Manager uses all of the identifiers in the list when it tries to match a device with an INF file. Enumerators place more specific identifiers ahead of less specific identifiers so that vendors can supply specific drivers that will be found in preference to more general drivers. The algorithm for constructing the strings depends on the enumerator, as follows:

PCI Devices

The full device identifier has the form

```
PCI\VEN_vvvv&DEV_dddd&SUBSYS_ssssssss&REV_rr
```

where *vvvv* is the vendor identifier that the PCI Special Interest Group assigned to the manufacturer of the card, *dddd* is the device identifier that the manufacturer assigned to the card, *ssssssss* is the subsystem ID (often 0) reported by the card, and *rr* is the revision number.

For example, the display adapter on a now-discarded laptop computer (based on the Chips and Technologies 65550 chip) had this identifier:

```
PCI\VEN_102C&DEV_00E0&SUBSYS_00000000&REV_04
```

A device can also match an INF model with any of these identifiers:

```
PCI\VEN vvvv&DEV dddd&SUBSYS ssssssss
PCI\VEN vvvv&DEV dddd&REV rr
PCI\VEN vvvv&DEV dddd
PCI\VEN vvvv&DEV dddd&REV rr&CC ccss
PCI\VEN vvvv&DEV dddd&CC ccsspp
PCI\VEN vvvv&DEV dddd&CC ccss
PCI\VEN vvvv&CC ccsspp
PCI\VEN vvvv&CC ccss
PCI\VEN vvvv
PCI\CC ccsspp
PCI\CC_ccss
```

in which *cc* is the base class code from the configuration space, *ss* is the subclass code, and *pp* is the programming interface. For example, the following additional identifiers for the aforementioned discarded laptop's display adapter would have matched the information in an INF file:

```
PCI\VEN 102C&DEV 00E0&SUBSYS 00000000
PCI\VEN 102C&DEV 00E0&REV 04
PCI\VEN 102C&DEV 00E0
PCI\VEN 102C&DEV 00E0&REV 04&CC 0300
PCI\VEN 102C&DEV 00E0&CC 030000
PCI\VEN 102C&DEV 00E0&CC 0300
PCI\VEN 102C&CC 030000
PCI\VEN 102C&CC 0300
PCI\VEN 102C
PCI\CC 030000
PCI\CC_0300
```

The INF that the system actually used for driver installation was the third one, which includes just the vendor and device identifiers.

PCMCIA Devices

The device identifier for a simple device has the form

```
PCMCIA\Manufacturer-Product-Crc
```

For example, the device identifier for the 3Com network card on the same dead laptop computer is

```
PCMCIA\MEGAHERTZ-CC10BT/2-BF05
```

For an individual function on a multifunction device, the identifier has the form

IPR2024-00292
KASPERSKY EX 1008 Page 411

```
PCMCIA\Manufacturer-Product-DEVdddd-Crc
```

where *Manufacturer* is the name of the manufacturer and *Product* is the name of the product. The PCMCIA enumerator retrieves these strings directly from tuples on the card. *Crc* is the 4-digit hexadecimal cyclical redundancy check (CRC) checksum for the card. The child function number (*dddd* in the template) is a decimal number without leading zeros.

If the card doesn't have a manufacturer name, the identifier will have one of these three forms:

```
PCMCIA\UNKNOWN MANUFACTURER-Crc
PCMCIA\UNKNOWN MANUFACTURER-DEVdddd-Crc
PCMCIA\MTD-0000 or PCMCIA\MTD-0002
```

(The last of these three alternatives is for a flash memory card with no manufacturer identifier on the card. The identifier with *0000* is for an SRAM card, while the one with *0002* is for a ROM card.)

In addition to the device identifier just described, an INF file's model section can contain an identifier composed by replacing the four-digit hexadecimal CRC with a string containing the four-digit hexadecimal manufacturer code, a hyphen, and the four-digit hexadecimal manufacturer information code (both from on-board tuples). For example:

```
PCMCIA\MEGAHERTZ-CC10BT/2-0128-0103
```

SCSI Devices

The complete device identifier is

```
SCSI\ttttvvvvvvvvpppppppppppppppprrrr
```

where *tttt* is a device type code, *vvvvvvvv* is an 8-character vendor identifier, *pppppppppppppppp* is a 16-character product identifier, and *rrrr* is a 4-character revision-level value. The device type code is the only one of the identifier components that doesn't have a fixed length. The bus driver determines this portion of the device identifier by indexing an internal string table with the device type code from the device's inquiry data, as shown in Table 15-7. (This table includes only the SCSI standard type codes. The SCSI enumerator can return additional type names for other types of device—see the DDK documentation for full information.) The remaining components are just the strings that appear in the device's inquiry data but with special characters (including space, comma, and any nonprinting graphic) replaced with an underscore.

SCSI Type Code	Device Type	Generic Type
DIRECT_ACCESS_DEVICE (0)	Disk	GenDisk
SEQUENTIAL_ACCESS_DEVICE (1)	Sequential	
PRINTER_DEVICE (2)	Printer	GenPrinter
PROCESSOR_DEVICE (3)	Processor	
WRITE_ONCE_READ_MULTIPLE_DEVICE (4)	Worm	GenWorm
READ_ONLY_DIRECT_ACCESS_DEVICE (5)	CdRom	GenCdRom
SCANNER_DEVICE (6)	Scanner	GenScanner
OPTICAL_DEVICE (7)	Optical	GenOptical
MEDIUM_CHANGER (8)	Changer	ScsiChanger
COMMUNICATION_DEVICE (9)	Net	ScsiNet

Table 15-7. *Type Names for SCSI Devices*

For example, a disk drive on one of my workstations has this identifier:

```
SCSI\DiskSEAGATE_ST39102LW_______0004
```

The bus driver also creates these additional identifiers:

```
SCSI\ttttvvvvvvvvpppppppppppppppp
SCSI\ttttvvvvvvvv
SCSI\vvvvvvvvpppppppppppppppr
vvvvvvvvpppppppppppppppr
gggg
```

In the third and fourth of these additional identifiers, *r* represents just the first character of the revision identifier. In the last identifier, *gggg* is the generic type code from Table 15-7.

To carry forward the example of my disk drive, the bus driver generated these additional device identifiers:

IPR2024-00292
KASPERSKY EX 1008 Page 412

```
SCSI\DiskSEAGATE ST39102LW
SCSI\DiskSEAGATE
SCSI\DiskSEAGATE ST39102LW       0
SEAGATE ST39102LW       0
GenDisk
```

The last of these (*GenDisk*) is the one that appeared as the device identifier in the INF file that the PnP Manager actually used to install a driver for this disk. In fact, the generic identifier is *usually* the one that's in the INF file because SCSI drivers tend to be generic.

IDE Devices

IDE devices receive device identifiers that are similar to SCSI identifiers:

```
IDE\ttttvpvprrrrrrrr
IDE\vpvprrrrrrrr
IDE\ttttvpvp
vpvprrrrrrrr
gggg
```

Here *ttt* is a device type name (same as SCSI); *vpvp* is a string containing the vendor name, an underscore, the vendor's product name, and enough underscores to bring the total to 40 characters; *rrrrrrrr* is an 8-character revision number; and *gggg* is a generic type name (almost the same as SCSI type names in Table 15-7). For IDE changer devices, the generic type name is *GenChanger* instead of *ScsiChanger*; other IDE generic names are the same as SCSI names.

For example, here are the device identifiers generated for an IDE hard drive on one of my desktop systems:

```
IDE\DiskMaxtor 91000D8                          SASX1B18
IDE\Maxtor 91000D8                            SASX1B18
IDE\DiskMaxtor 91000D8
Maxtor 91000D8                              SASX1B18
GenDisk
```

ISAPNP Devices

The ISA Plug-And-Play (ISAPNP) enumerator constructs two hardware identifiers:

```
ISAPNP\id*altid
```

where *id* and *altid* are EISA-style identifiers for the device—three letters to identify the manufacturer and four hexadecimal digits to identify the particular device. If the device in question is one function of a multifunction card, the first identifier in the list takes this form:

```
ISAPNP\id_DEVnnnn
```

where *nnnn* is the decimal index (with leading zeros) of the function.

For example, the codec function of the Crystal Semiconductor audio card on one of my desktop machines has these two hardware identifiers:

```
ISAPNP\CSC6835 DEV0000
*CSC0000
```

The second of these identifiers is the one that matched the actual INF file.

USB Devices

The complete device identifier is

```
USB\VID_vvvv&PID_dddd&REV_rrrr
```

where *vvvv* is the four-digit hexadecimal vendor code assigned by the USB committee to the vendor, *dddd* is the four-digit hexadecimal product code assigned to the device by the vendor, and *rrrr* is the revision code. All three of these values appear in the device descriptor or interface descriptor for the device.

An INF model section can also specify these alternatives:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 413

```
USB\VID vvvv&PID dddd
USB\CLASS cc&SUBCLASS ss&PROT pp
USB\CLASS cc&SUBCLASS ss
USB\CLASS cc
USB\COMPOSITE
```

where *cc* is the class code from the device or interface descriptor, *ss* is the subclass code, and *pp* is the protocol code. These values are in two-digit hexadecimal format.

A *composite* USB device is one that has more than one interface descriptor (not counting alternate settings) in its only configuration. If no more specific device identifier for a composite device matches an INF file, the system will match the *USB\COMPOSITE* compatible identifier with the generic parent driver. The generic parent driver in turn creates a PDO for each interface. The device identifier for the PDO is of the following form:

```
USB\VID_vvvv&PID_dddd&MI_nn
```

where *vvvv* and *dddd* are as before and *nn* is the *bInterfaceNumber* from the interface descriptor. The generic parent driver also creates compatible identifiers based on the class, subclass, and protocol codes in the interface descriptor:

```
USB\CLASS_cc&SUBCLASS_ss&PROT_pp
USB\CLASS_cc&SUBCLASS_ss
USB\CLASS_cc
```

1394 Devices

The 1394 bus driver constructs these identifiers for a device:

```
1394\VendorName&ModelName1394\UnitSpecId&UnitSwVersion
```

where *VendorName* is the name of the hardware vendor, *ModelName* identifies the device, *UnitSpecId* identifies the software specification authority, and *UnitSwVersion* identifies the software specification. The information used to construct these identifiers comes from the device's configuration ROM.

If a device has vendor and model name strings, the 1394 bus driver uses the first identifier as the hardware ID and the second identifier as the one and only compatible ID. If a device lacks a vendor or model name string, the bus driver uses the second identifier as the hardware ID.

Since I don't have a 1394 bus on any of my computers, I relied on fellow driver writer Jeff Kellam to provide me with two examples. The first example is for a Sony camera, for which the device identifier is

```
1394\SONY&CCM-DS250_1.08
```

The second example is for the 1394 bus itself operating in diagnostic mode; this device identifier is

```
1394\031887&040892
```

Please refer to the DDK for information about device identifiers in composite 1394 devices.

Identifiers for Generic Devices

The PnP Manager also works with device identifiers for generic devices that can appear on many different buses. These identifiers are of the form

*PNP*dddd*

where *dddd* is a four-digit hexadecimal type identifier. Microsoft keeps moving the list of these identifiers, so every link I publish breaks within a few months. Good luck finding it!

15.2.6 Driver Ranking

The DDK describes the complete algorithm that the setup program uses to rank drivers when it finds several INF files that all describe a new piece of hardware with greater or lesser specificity. (See the entry entitled, "How Setup Selects Drivers.") The entire algorithm is very complex and depends on several factors:

- Whether a driver package is signed.
- Whether a device identifier or a compatible identifier generated by the bus driver matches the device identifier or a

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 414

compatible identifier in a model statement and, if so, how specific the match is. Look back a few pages, and notice how the bus drivers generate lists of identifiers that begin with a very specific identifier and then become progressively less specific and that they generate both device identifier lists and compatible identifier lists.

- What the *DriverVer* statement in a signed INF file happens to say.
- Which operating system platform you're installing the device for.

I won't repeat all the ranking rules because you and I are unlikely to be involved in situations in which very many of them matter. Three rough rules of thumb are these: First, signed drivers have absolute preference over unsigned drivers in Windows XP. Second, within a group of drivers that are all signed or all unsigned, a match involving more specific identifiers is preferred to a less specific match. Finally, all else being equal, setup will break a tie between signed drivers by comparing the *DriverVer* values. Thus, you'll want to get your driver signed and to specify a full device identifier in a model statement. When you distribute updates, you'll want to be sure you update the *DriverVer* value so that systems everywhere realize the new driver is really new.

Some Microsoft bus drivers generate compatible device identifiers that map to signed function drivers. This creates a horrible problem for a vendor who wants to provide an unsigned driver that matches a more specific device identifier. Here's an example of what can go wrong. Let's say you've created a wonderful new USB mouse. (This is a real example, by the way, based on something that happened to one of my consulting clients who agreed to let his story be told.) The end user plugs this mouse in to a Windows XP computer. The USB hub driver creates a set of specific device identifiers based on the vendor and product ID in the device descriptor. So far, so good. If matters rested here, the system would go on to query for a device driver, and your driver would be found. In the interval between releasing the mouse and getting the mouse and the driver certified by WHQL, the user would also see an unsigned driver warning dialog box, which he or she could choose to ignore or not.

But matters don't rest there. The USB hub driver also creates a compatible device identifier based on the USB class code in the mouse's descriptors. The compatible ID matches a Microsoft INF that specifies HIDUSB.SYS as the driver for a HID-class device. Since the Microsoft driver is signed, the system *always* prefers it to an unsigned driver, even when the unsigned driver is for the specific product that has been detected and even though the end user would be willing to accept the unsigned driver.

At this point, the end user has to go through exactly the right steps in the Update Driver dialog boxes to get Microsoft's signed but generic driver replaced by your unsigned but specific driver. These steps are easy to get wrong and inevitably lead to support calls that wipe out the profit from the hardware sale. Not only do your users have to do this, but so do *you* when you want to test your driver package in-house. Your users have to do this not only when they first install your mouse but also *every* time they plug it in to a different port on their USB bus.

I'm withholding editorial comment on the way the ranking algorithm works out in this kind of situation. You can do a couple of things to make sure something similar doesn't happen to you. You can try to avoid putting your device in a class for which Microsoft provides a generic driver based on a compatible ID. In the case of a HID device, this approach will render it unusable at boot time and might, for that reason, be unacceptable. Alternatively, you can make sure you have enough time and money to get WHQL certification for your device before you begin to sell it.

NOTE
Microsoft offers a test signature program that allows you to get what amounts to a dummy signature for a driver. This program is not practical for a solo consultant because it requires you to obtain a VeriSign certificate, which is assuredly not free and which in turn requires that you jump through various legal hoops that have nothing to do with your ability to write drivers. Microsoft designed the current test signature program to work well for large companies, from whom the great bulk of driver submissions ultimately come, and periodically reviews its plans to see whether smaller fry can be accommodated. I started down the path toward getting a test certificate just so I could try it out and tell you readers how it worked, but I gave up when it became apparent how inflexible the legal requirements were.

15.2.7 Tools for INF Files

If you look in the TOOLS subdirectory of the Windows XP DDK, you'll find two useful utilities for working with INF files. CHKINF will help you validate an INF file, and GENINF will help you build a new INF file. Tools for putting *Version* stamps in INF files that appeared in earlier DDKs appear to have been dropped in the .NET DDK.

CHKINF

CHKINF is actually a BAT file that runs a PERL script to examine and validate an INF file. You'll obviously need a PERL interpreter to use this tool. I got a copy once upon a time from http://www.perl.com. The Hardware Compatibility Tests also include a copy.

You can run CHKINF most easily from a command prompt. For example:

```
C:\winddk\3615\tools\chkinf>chkinf C:\newbook\chap15\devprop\sys\devprop.inf
```

CHKINF generates HTML output files in an HTM subdirectory. The output includes a summary of all the errors and warnings found by CHKINF, followed by an annotated version of the INF file itself. I decided to show the unvarnished truth about

IPR2024-00292
KASPERSKY EX 1008 Page 415

DEVPROP.INF so you can see for yourself how CHKINF will help you prepare your driver package. Parents be warned: Figure 15-11 contains graphic images that may be inappropriate for all viewers. Well, perhaps it's not as shocking in black and white.

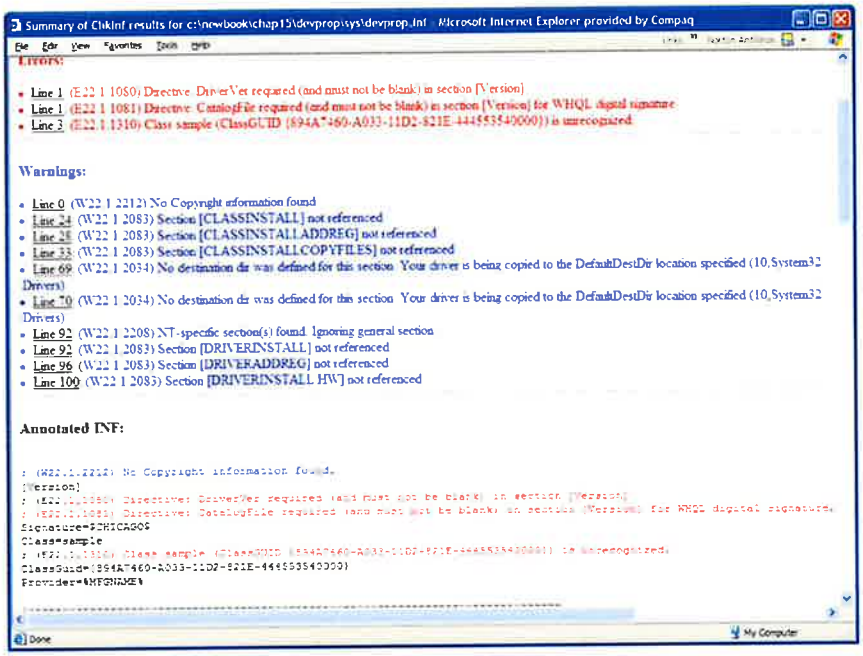


Figure 15-11. *Output from CHKINF utility.*

CHKINF shows me that I omitted the Version and CatalogFile sections that would be needed if I ever wanted to get WHQL certification for this "device." It also—incorrectly, in my view—tells me that the *SAMPLE* class GUID is undefined, even though the INF also contains a ClassInstall32 section to fully define the *SAMPLE* setup class. All of the warnings relate to things in the INF file that I might not have realized were going to happen. For example, telling me about unreferenced sections might alert me to a misspelling. As it happens, I'm content with the INF as it stands and don't plan to take any action based on the warnings.

You should know that CHKINF verifies only the basic syntax and structure of an INF file. It doesn't verify that you copy files to the right directories, or even that you copy all the files that you use. (The WHQL signability test will, however, check these items.) It doesn't understand the [ClassInstall] section needed for a new device class in Windows 98/Me. It doesn't understand that an undecorated section name like [DriverInstall] would be used by Windows 98/Me and so warns that the other Windows 98/Me sections in the INF aren't referenced. In other words, CHKINF isn't a perfect tool, but it will give you useful information. Plus, your INF file has to pass CHKINF muster for you to pass WHQL.

GENINF

GENINF is a graphical-based wizard that can help you construct INF files. This tool has come a long way from its beginnings in the early betas of the Windows 2000 DDK, and I think that many readers can profit from at least trying it out. It wouldn't have helped me with the samples in this book, though, because it doesn't support custom setup classes or cross-platform INF files. It's also a bit limited in terms of the device classes it does support.

GENINF also doesn't embody some of the class-specific knowledge that you need to build a working INF file. For example, I used it to build an INF file for a HID minidriver, and it didn't generate the syntax that would be needed to make sure HIDCLASS and HIDPARSE were installed on the end user system. It didn't ask me whether my device was a joystick, which would have required some additional input with regard to axis and button configuration.

I'm not carping here—I think GENINF is a step in the right direction, but you need to realize that it won't write your INF file for you.

Debugging Your INF File

One of the biggest problems with debugging an INF file is that it's so much like the old game of Adventure. Do something wrong, and you learn, "You can't do that." In other words, the system doesn't accept your INF file, and you have to try to figure out why, usually by removing more and more lines until you finally figure out what's causing the problem. I recall once spending several hours learning that Windows Me was unhappy with my INF file because one of the Windows 2000 sections had a name longer than 28 characters (a fact that was not then documented in the DDK). Once past that hurdle, I had similar

IPR2024-00292
KASPERSKY EX 1008 Page 416

pain learning that the disk-label field in a *SourceDisksNames* entry isn't optional in Windows Me.

In Windows XP, the device installer logs various information about the operations it performs in a disk file named SETUPAPI.LOG in the Windows directory. You can control the verbosity of the log and the name of the log file by manually changing entries in the registry key named *HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup*. Please consult the DDK documentation for detailed information about these settings. I just set the *LogLevel* parameter to 0xFFFFFFFF and get way more debugging output than I can usually use, but that's easier than trying to figure out the meanings of 32 bits in a mask.

Sometimes, a problem with your INF file or another aspect of setup shows up as a problem code in the Device Manager. (See Figure 15-12.) The DDK documents the problem codes under the heading "Device Manager Error Messages," and there are about 50 of them. This particular example is what happens after you install the STUPID sample from Chapter 2. Problem code 31 is *CM_PROB_FAILED_ADD*, and it means that the driver's *AddDevice* function failed. This accurately describes the way STUPID works, of course.

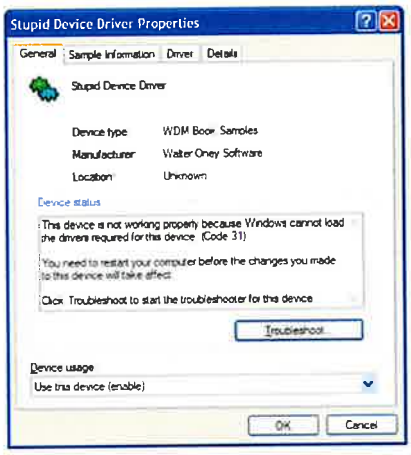


Figure 15-12. *A setup problem in the Device Manager.*

15.3 Defining a Device Class

Let's suppose you have a device that doesn't fit into one of the device classes Microsoft has already defined. When you're initially testing your device and your driver, you can get away with using the *Unknown* class in your INF file. Production devices are not supposed to be in the Unknown class, however. You should instead place your custom device in a new device class that you define in the INF file. I'll explain how to create a custom class in this section.

The INF example I showed you earlier relied on a custom device class:

```
[Version]
Signature=$CHICAGO$
Class=Sample
ClassGuid={894A7460-A033-11d2-821E-444553540000}
```

In fact, all of the samples in this book use the *Sample* class.

When you want to define a new class of device, you need to run GUIDGEN to create a unique GUID for the class. You can add polish to the user interface for your device class by writing a property page provider for use with the Device Manager and putting some special entries in the registry key your class uses. You can also provide filter drivers and parameter overrides that will be used for every device of your class. You control each of these additional features by statements in your INF file. For example, each of the INF files for the samples in the companion content includes these boilerplate entries:

```
1
[ClassInstall32]
   AddReg=ClassInstall32AddReg
   CopyFiles=ClassInstall32CopyFiles

   [ClassInstall32AddReg]
2
   HKR,,,,"WDM Book Samples"
3
```


IPR2024-00292
KASPERSKY EX 1008 Page 417

```
HKR,,Installer32,,"samclass.dll,SampleClassInstaller"
4
HKR,,EnumPropPages32,,samclass.dll
5
HKR,,Icon,,101

[ClassInstall32CopyFiles]
6
samclass.dll,,,2
```

1. Whenever you use a nonstandard setup class in your INF file, you should have a [ClassInstall32] section to define the class. Don't do what I did in the first edition and depend on some other setup procedure to define the class. Note that the system uses this section the very first time it installs a device belonging to your custom class.
2. In the AddReg section for a [ClassInstall32] section, HKR refers to the class key. The default (unnamed) value is the friendly name of the class that will appear in Device Manager and hardware install wizard dialog boxes.
3. *Installer32* names a class installer DLL. For the *SAMPLE* class, I combined the property page provider and the installer into one DLL named SAMCLASS.DLL. I included the installer so I could provide a custom icon for the class.
4. *EnumPropPages32* names a Device Manager property page provider.
5. Icon specifies the icon resource in the property page provider DLL.
6. This statement is how we make sure that the property page and installer DLL gets copied during installation of the first *SAMPLE* device. A statement in the [DestinationDirs] section of the INF file directs this file to the system directory.

15.3.1 A Property Page Provider

Way back in the introduction to this book, I showed you a screen shot of the property page I invented for use with the *Sample* device class. The SAMCLASS sample in the companion content is the source code for the property page provider that produced that page, and I'm now going to explain how it works.

TIP
Microsoft recommends that you use an installer or co-installer DLL as a vehicle for adding property pages to the Device Manager's property sheet. The property-page provider concept described here is still supported, and you can put the property-page function in the same DLL as an installer or a co-installer. In addition, you'll need a (16-bit) property page provider DLL to get the same functionality in Windows 98/Me, and the code in one of those is nearly identical to that in a 32-bit DLL for Windows 2000 and later systems. It's your choice how you proceed. If you want to follow the Microsoft recommendation, I suggest that you consult the CLASSINSTALLER portion of the TOASTER sample in the DDK or the COINSTALLER sample in the companion content for this book.

A property page provider for a device class is a 32-bit DLL with the following contents:

- An exported entry point for each class for which the DLL supplies property pages
- Dialog resources for each property page
- A dialog procedure for each property page

In general, a single DLL can provide property pages for several device classes. Microsoft supplies some DLLs with the operating system that do this, for example. SAMCLASS, however, provides only a single page for a single class of device. Its only exported entry point is the following function:

```
extern "C" BOOL CALLBACK EnumPropPages
  (PSP PROPSHEETPAGE REQUEST p,
  LPFNADDPROPSHEETPAGE AddPage, LPARAM lParam)
  {
  PROPSHEETPAGE page;
  HPROPSHEETPAGE hpage;
  memset(&page, 0, sizeof(page));
  page.dwSize = sizeof(PROPSHEETPAGE);
  page.hInstance = hInst;
  page.pszTemplate = MAKEINTRESOURCE(IDD SAMPAGE);
  page.pfnDlgProc = PageDlgProc;
  <some more stuff>
  hpage = CreatePropertySheetPage(&page);
  if (!hpage)
    return TRUE;
  if (!(*AddPage)(hpage, lParam))
```


IPR2024-00292
KASPERSKY EX 1008 Page 418

```
    DestroyPropertySheetPage(hpage);
  return TRUE;
  }
```

When the Device Manager is about to construct the property sheet for a device, it consults the class registry key to see whether there's a property page provider. It loads the DLL you specify (SAMCLASS.DLL in the case of *SAMPLE*) and calls the designated entry point (*EnumPropPages*). If the function returns *TRUE*, the Device Manager will display the property page; otherwise, it won't. The function can add zero or more pages by calling the *AddPage* function as shown in the preceding example.

Inside the *SP_PROPSHEETPAGE_REQUEST* structure your enumeration function receives as an argument, you'll find two useful pieces of information: a handle to a device information set and the address of an *SP_DEVINFO_DATA* structure that pertains to the device you're concerned with. These data items (but not, unfortunately, the *SP_PROPSHEETPAGE_REQUEST* structure that contains them) remain valid for as long as the property page is visible, and it would be useful for you to be able to access them inside the dialog procedure you write for your property page. Windows SDK Programming 101 (well, maybe 102 because this process is a little obscure) taught you how to do this. First create an auxiliary structure whose address you pass to *CreatePropertySheetPage* as the *lParam* member of the *PROPSHEETPAGE* structure:

```
struct SETUPSTUFF {
  HDEVINFO info;
  PSP DEVINFO DATA did;
  char infopath[MAX PATH];
  };

BOOL EnumPropPages(...)
  {
  PROPSHEETPAGE page;
  .
  .
  SETUPSTUFF* stuff = new SETUPSTUFF;
  stuff->info = p->DeviceInfoSet;
  stuff->did = p->DeviceInfoData;
  page.lParam = (LPARAM) stuff;
  page.pfnCallback = PageCallbackProc;
  page.dwFlags = PSP USECALLBACK;
  .
  .
  }

UINT CALLBACK PageCallbackProc(HWND junk, UINT msg, LPPROPSHEETPAGE p)
  {
  if (msg == PSPCB RELEASE && p->lParam)
    delete (SETUPSTUFF*) p->lParam;
  return TRUE;
  }
```

The *WM_INITDIALOG* message that Windows sends to your dialog procedure gets an *lParam* value that's a pointer to the same *PROPSHEETPAGE* structure, so you can retrieve the *stuff* pointer there. You can then use *SetWindowLong* and *GetWindowLong* to save the *stuff* pointer.

You also need to provide a way to delete the *SETUPSTUFF* structure when it's no longer needed. The easiest way, which works whether or not you ever get a *WM_INITDIALOG* message (you won't if there's an error constructing your property page), is to use a property page callback function as shown in the preceding fragment.

You can do all sorts of things in a custom property page. For the sample class, I wanted to provide a button that would bring up an explanation for each sample device. To keep things as general as possible, I decided to put a *SampleInfo* value naming the explanation file in the device's hardware registry key. To invoke a viewer for the explanation file, it suffices to call *ShellExecute*, which will interpret the file extension and locate an appropriate viewer application. For my book samples, the explanation files are HTML files, so the viewer in question will be your Web browser.

Most of the work in SAMCLASS occurs in the *WM_INITDIALOG* handler. (Error checking is again omitted.)

```
case WM INITDIALOG:
  {
  SETUPSTUFF* stuff = (SETUPSTUFF*) ((LPPROPSHEETPAGE) lParam)->lParam;
1
  SetWindowLong(hdlg, DWL USER, (LONG) stuff);

2
  TCHAR name[256];
  SetupDiGetDeviceRegistryProperty(stuff->info, stuff->did,
    SPDRP_FRIENDLYNAME, NULL, (PBYTE) name, sizeof(name), NULL);
```


IPR2024-00292
KASPERSKY EX 1008 Page 419

```
    SetDlgItemText(hdlg, IDC_SAMNAME, name);

3
    HWND hClassIcon = GetDlgItem(hdlg, IDC_CLASSICON);
    HICON hIcon;
    SetupDiLoadClassIcon(&stuff->did->ClassGuid, &hIcon, NULL);
    SendMessage(hClassIcon, STM_SETICON, (WPARAM) (HANDLE) hIcon, 0);

4
  HKEY hkey = SetupDiOpenDevRegKey(stuff->info, stuff->did,
      DICS_FLAG_GLOBAL, 0, DIREG_DEV, KEY_READ);
    DWORD length = sizeof(name);
    RegQueryValueEx(hkey, "SampleInfo", NULL, NULL,
      (LPBYTE) name, &length);
    DoEnvironmentSubst(name, sizeof(name));
    strcpy(stuff->infopath, name);
    RegCloseKey(hkey);
    break;
    }
```

1. This statement saves the *SETUPSTUFF* pointer where the dialog procedure can find it to handle later window messages.
2. Here we determine the friendly name for the device and put it in a static text control. The actual code sample obtains the device description if there's no friendly name. In the dialog template, I positioned the control quite carefully to match the position where the Device Manager puts a similar control on other pages of the property sheet. Taking this bit of extra care means that the text (which appears on every page) doesn't appear to hop around as you tab from one page to the next.
3. These statements determine the class icon. In this particular sample, I could have hard-coded the class icon in the dialog template since SAMCLASS is used only with *SAMPLE* class devices. I preferred to show you this more general way of getting the same icon that the Device Manager uses on other pages of the property sheet. Just as with the static control containing the friendly name, I had to position the icon control carefully in the dialog template.
4. The next few statements determine the *SampleInfo* filename from the hardware key's parameter subkey. The strings I put in the registry are of the form %wdmbook%\chap15\devprop\devprop.htm, in which %wdmbook% indicates substitution by the value of the WDMBOOK environment variable. The call to *DoEnvironmentSubst*, a standard Win32 API, expands the environment variable. (Trust me that I don't just do a blind *strcpy* in the actual code—I check the length first.)

When the end user—that would be you in this particular situation, I think—presses the More Information button on the property page, the dialog procedure receives a *WM_COMMAND* message, which it processes as shown here:

```
SETUPSTUFF* stuff = (SETUPSTUFF*) GetWindowLong(hdlg, DWL_USER);

case WM_COMMAND:
  switch (LOWORD(wParam))
    {
  case IDB_MOREINFO:
    {
    ShellExecute(hdlg, NULL, stuff->infopath,
      NULL, NULL, SW_SHOWNORMAL);
    return TRUE;
    }
    }
  break;
```

ShellExecute will launch the application associated with the SampleInfo file—namely, your Web browser—whereupon you can view the file and find all sorts of interesting information.

15.4 Customizing Setup

In this section, I'll discuss a number of ways you can tailor the setup process to the requirements of your device. First of all, if your device belongs to a new setup class, you can write an *installer DLL* to handle all the details of installing and managing a device of that class. I think that few readers of this book will actually need to do this. At most, you might need to provide a *class-specific co-installer DLL* to modify the default behavior of the system setup programs. If your device belongs to an existing setup class or to a new class that has no special requirements, you can provide a *device-specific co-installer DLL* that comes into play for just a few of the installation and management steps.

If your driver package has a digital signature (see the major section on WHQL at the end of this chapter), you can streamline the setup process by *preinstalling* your software on the end user's machine. By designing your package not to require interaction with the user, you will thereby permit an unattended *server-side* install that can occur even without an administrator

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 420

present at the computer.

Often, vendors ship special applications along with their hardware. Microsoft refers to software that's in addition to the signed driver package as *value-added software*. Using a co-installer DLL, you can provide for installation of the value-added software along with the drivers and other digitally signed components in your package. Sometimes, you might want to launch an application after setup completes, and the *RunOnce* key in the registry provides a vehicle for doing that. As a special case, you might want to launch an application each time the user plugs in your device. I'll briefly describe the scheme used by the AUTOLAUNCH sample to help you accomplish that as well.

NOTE
A so-called *server-side* install is possible with a signed package that requires no user interaction. A *client-side* install requires that an administrator be present and occurs whenever a package is unsigned or requires user interaction to locate the INF file, to locate one or more of the driver files, to initiate the add-hardware wizard in the first place, or because a component displays a property page. According to the DDK, "The term *server-side* is used because installation can be accomplished by the system's PnP Manager without a user-mode client making calls into the PnP Manager."

15.4.1 Installers and Co-installers

The system setup program gets into the act of installing, removing, and managing all the devices in the system. It works with a collection of installer and co-installer DLLs to perform the required actions. You can write such DLLs yourself, and you can install them along with a driver package by putting directives in your INF file.

NOTE
Strictly speaking, the setup program calls installer and co-installer *functions*, which the program finds in DLLs someplace on the system. One DLL can contain many entry points. I think that most readers of this book will tend to create only co-installer DLLs (and probably only device-specific ones at that) and to put just one entry point in each of those DLLs.

To speak generally, the setup program needs to identify the setup class for a device before the program can know which installer DLL and which class-specific co-installer DLLs it will call. Several pathways through the system setup functions lead up to identifying the setup class:

- When you install a PnP device, the setup program will match a device or a compatible identifier returned by the bus driver with a model statement in an INF file. The setup program learns the device's setup class by reading the [Version] section in the INF file. If necessary, the setup program will also process the directives in the [ClassInstall32] section to create a new class.
- When you install a non-PnP device, you use the Add Hardware wizard. The wizard allows you to designate a specific setup class as containing the device you want to install.
- When you install a non-PnP device belonging to a new class, you perforce have to tell the setup program which INF file you want to use. Setup determines the class from the [Version] section and will install a new class based on the [ClassInstall32] section.
- When you undertake to modify an existing device through the Device Manager, or by means of some other similar management utility, you make dialog choices that specify a particular device, whose setup class is recorded in the registry.

As installation of a new device or modification of an existing device proceeds, the setup program eventually knows which device you're working with. It then starts using any device-specific co-installer DLLs that exist. For many of the tasks the setup program performs, therefore, it uses an installer DLL, one or more class-specific co-installers, and one or more device-specific co-installers.

Setup Function Codes

The setup program communicates with the various DLLs by calling an exported function. Among the arguments to the function is a *DIF code* that indicates which function the DLL should perform. In this subsection, I'll present an overview of the DIF codes used for several common activities. Refer to the DDK (specifically the section entitled "Device Installation Function Codes") for a detailed explanation of the tasks to perform.

NOTE
I built the debug version of SAMCLASS.DLL and recorded the resulting trace information as I did various things with devices. Don't assume that these are the only steps that would be performed on all systems or that the listed steps wouldn't be performed in other situations too.

In the tables that follow, I indicate in the Co-installer? column whether the setup program sends a particular DIF code to a co-installer. It sends all the noted codes to the installer.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 421

DIF Code	Co-installer?	Summary of Operation
DIF_REGISTERDEVICE	No	Determine whether non-PnP device is a duplicate.
DIF_SELECTBESTCOMPATDRV	No	Modify list of possible drivers; possibly prune list or select the driver.
DIF_ALLOW_INSTALL	No	Determine whether setup should proceed to install the device.
DIF_INSTALLDEVICEFILES	No	Copy files; modify list of files to be copied later.
DIF_REGISTER_COINSTALLERS	No	Modify list of device-specific co-installers.
DIF_INSTALLINTERFACES	Yes	Register device interfaces.
DIF_INSTALLDEVICE	Yes	Do whatever is needed before setup loads the drivers for the device.
DIF_NEWDEVICEWIZARD_FINISHINSTALL	Yes	Create any desired additional wizard pages
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-8. *Installing a Non-PnP Device (for Example, IOCTL)*

DIF Code	Co-installer?	Summary of Operation
DIF_SELECTBESTCOMPATDRV	No	Modify list of possible drivers; possibly prune list or select the driver.
DIF_ALLOW_INSTALL	No	Determine whether setup should proceed to install the device.
DIF_INSTALLDEVICEFILES	No	Copy files; modify list of files to be copied later.
DIF_REGISTER_COINSTALLERS	No	Modify list of device-specific co-installers.
DIF_INSTALLINTERFACES	Yes	Register device interfaces.
DIF_INSTALLDEVICE	Yes	Do whatever is needed before setup loads the drivers for the device.
DIF_NEWDEVICEWIZARD_FINISHINSTALL	Yes	Create any desired additional wizard pages.
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-9. *Installing a PnP Device (for Example, USB42)*

The only difference between this scenario and the non-PnP installation scenario is the absence of the *DIF_REGISTERDEVICE* step.

Note that these actions occur only when you're initially installing the drivers for a PnP device. Thereafter, you can unplug and replug the device any number of times without involving the setup program.

DIF Code	Co-installer?	Summary of Operation
DIF_ADDPROPERTYPAGE_ADVANCED	Yes	Create any desired property pages.
DIF_POWERMESSAGEWAKE	Yes	Provide text for the Power Management tab.
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-10. *Examining Properties in the Device Manager*

DIF Code	Co-installer?	Summary of Operation
DIF_PROPERTYCHANGE	Yes	Alert the installer/co-installer to a change in device state (start/stop, enable/disable) or in a configuration parameter.
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-11. *Disabling a Device in the Device Manager*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 422

DIF Code	Co-installer?	Summary of Operation
DIF_PROPERTYCHANGE	Yes	Alert the installer/co-installer to a change in device state (start/stop, enable/disable) or in a configuration parameter.
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-12. *Enabling a Device in the Device Manager*

DIF Code	Co-installer?	Summary of Operation
DIF_REMOVE	Yes	Decide whether removal should proceed; clean up any persistent private data.
DIF_DESTROYPRIVATEDATA	Yes	Perform cleanup.

Table 15-13. *Removing a Device in the Device Manager*

Surprise-Removing a PnP device

No setup actions occur when you surprise-remove a PnP device.

Co-installer DLLs

A co-installer DLL has an exported entry point with the following prototype:

```
DWORD   stdcall CoinstallerProc(DI FUNCTION dif, HDEVINFO infoset,
  PSP_DEVINFO_DATA did, PCOINSTALLER_CONTEXT_DATA ctx);
```

The *dif* argument is one of the DIF codes discussed in the preceding subsection, *infoset* is a handle to a device information collection, *did* is the address of a structure containing information about a specific single device, and *ctx* is a pointer to a context structure (declared, along with the entire public interface to the setup program, in SETUPAPI.H):

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

The setup program calls the co-installer procedure either once or twice for each DIF code, as directed by the co-installer itself. The initial call is for preprocessing before an action is carried out. The *PostProcessing* member of the context structure will be *FALSE* in this initial call. The co-installer returns one of these values:

- *NO_ERROR*, which indicates that the co-installer has done everything it wants to do in regard to the DIF code.
- *ERROR_DI_POSTPROCESSING_REQUIRED*, which indicates that the co-installer wants to do additional work after the action associated with the DIF code completes.
- Any Win32 error except *ERROR_DI_DO_DEFAULT*, which indicates that some sort of error occurred. Co-installers should not return *ERROR_DI_DO_DEFAULT*, as this might cause the setup program to do the wrong thing.

Generally speaking, if the co-installer returns an error code, the setup program will abort some operation. You need to consult the detailed documentation for each DIF code to see exactly what will happen, though.

In the second of these return cases—returning *ERROR_DI_POSTPROCESSING_REQUIRED*—the co-installer can set the *PrivateData* member of the context structure to any desired value before returning. The setup program will call the co-installer again later with the same DIF code, *infoset*, and *did* values. The *PostProcessing* member of the context structure will be *TRUE*, and the *InstallResult* member will indicate the return code from the previous installer or co-installer DLL. *PrivateData* will equal whichever value the co-installer returned in the preprocessing phase. The co-installer can return *NO_ERROR* or a Win32 error code in response to a postprocessing call; the default return value should be *InstallResult*.

The DDK contains two examples of co-installer DLLs—COINST in TOASTER and TOASTCO in TOASTPKG. I know that I'd prefer to write user-mode code, especially code that includes user-interface elements, using Microsoft Foundation Classes. If you share this preference, you'll want to take a close look at the COINSTALLER sample in the companion content. This sample includes two general-purpose classes that you can copy to an MFC DLL project in order to very easily construct a co-installer DLL:

- *CCoinstaller*, derived from *CExternalDialogApp* (my class) and, ultimately, from *CWinApp* (a standard MFC class), encapsulates a co-installer DLL. The *CCoinstaller* class includes the co-installer procedure described earlier, virtual functions for the preprocessing and postprocessing variants of each relevant DIF code, and an *AddPropertyPage* function for easily populating either the Device Manager property sheet or the installation wizard with custom pages.
- *CCoinstallerDialog*, derived from *CExternalDialog* (my class) and, ultimately, from *CPropertyPage* (a standard MFC

IPR2024-00292
KASPERSKY EX 1008 Page 423

class), encapsulates a property page belonging to an external property sheet. *CCoinstallerDialog::OnInitDialog* also automatically initializes controls named IDC_CLASSICON and IDC_DEVNAME, if they exist, with the class icon and the device name. This initialization considerably reduces the burden of building a Device Manager property page.

Here is the entire declaration and implementation of the *CSampleCoinstaller* class. I removed some MFC commentary that just clutters the presentation.

```
class CSampleCoinstallerApp : public CCoinstaller
{
public:
  CSampleCoinstallerApp();
  virtual ~CSampleCoinstallerApp();

public:
  virtual DWORD AddPropertyPages(HDEVINFO infoset,
    PSP DEVINFO DATA did, PVOID& PostContext);
  virtual DWORD FinishInstall(HDEVINFO infoset,
    PSP DEVINFO DATA did, PVOID& PostContext);
  CShoeSize* m shoesize;
  CShoeSizeProperty* m shoesizeprop;
  DECLARE MESSAGE MAP()
};

BEGIN MESSAGE MAP(CSampleCoinstallerApp, CExternalDialogApp)
END MESSAGE MAP()

CSampleCoinstallerApp::CSampleCoinstallerApp()
  {
  m shoesize = NULL;
  m shoesizeprop = NULL;
  }

CSampleCoinstallerApp::~CSampleCoinstallerApp()
  {
  if (m shoesize)
    delete m shoesize;
  if (m shoesizeprop)
    delete m shoesizeprop;
  }

CSampleCoinstallerApp theApp;

DWORD CSampleCoinstallerApp::AddPropertyPages(HDEVINFO infoset,
  PSP DEVINFO DATA did, PVOID& PostContext)
  {
  m shoesizeprop = new CShoeSizeProperty;
  AddPropertyPage(infoset, did, m shoesizeprop);
  return NO ERROR;
  }

DWORD CSampleCoinstallerApp::FinishInstall(HDEVINFO infoset,
  PSP DEVINFO DATA did, PVOID& PostContext)
  {
  m shoesize = new CShoeSize;
  AddPropertyPage(infoset, did, m shoesize);
  return NO ERROR;
  }
```

CShoeSize and *CShoeSizeProperty* are standard MFC-generated dialog classes derived from *CCoinstallerDialog*. They implement the behavior of a wizard page and a Device Manager page, respectively.

AddPropertyPages overrides a virtual base function to handle the *DIF_ADDPROPERTYPAGE_ADVANCED* call. *FinishInstall* overrides a virtual base function to handle the *DIF_NEWDEVICEWIZARD_FINISHINSTALL* call.

Since this book is (by original intention, anyway) a treatise on driver programming, I won't go into the details of how I implemented the base classes. Figure 15-13 illustrates the wizard page that you'll see if you install the dummy device associated with the COINSTALLER sample.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 424

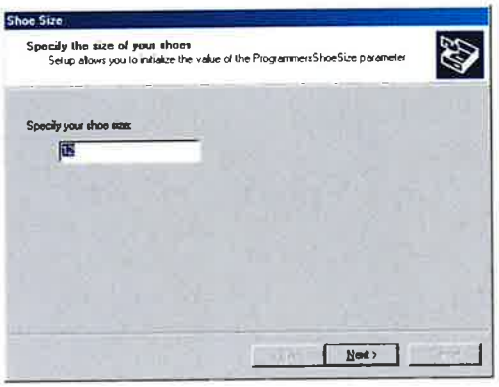


Figure 15-13. *Wizard page presented by a co-installer DLL.*

The wizard page in COINSTALLER has an important defect that you'd want to correct in a real situation. It purports to let you initialize a configuration parameter. The PnP Manager has, however, already loaded and initialized the driver by the time this page appears. The way the co-installer and the driver are coded, the driver won't pick up the changed parameter until the next time it starts. The co-installer might cause the device to restart automatically when the installation wizard completes. (In fact, the Device Manager property page contains a check box for doing just that.) Alternatively, you can provide a dynamic method, such as an I/O control (IOCTL) call or a Windows Management Instrumentation (WMI) call, to notify the driver that the parameter has changed. I elected not to burden the sample with this extra complication, though.

> **NOTE**
> One important use for a co-installer DLL is to assign a unique friendly name to a device when you have more than one device installed. Since, however, co-installer DLLs aren't supported in Windows 98/Me, you might need an alternative method. The MAKENAME sample in the companion content shows how to use RUNDLL32 with a simple DLL to create unique friendly names.

15.4.2 Preinstalling Driver Files

You achieve the most painless end user experience by preinstalling all the files needed for a server-side install. You'd follow these steps:

1. First get your driver installation package digitally signed. Preinstallation alone doesn't allow an unsigned package to be installed without user interaction. In other words, you don't gain much by preinstalling an unsigned package.

2. Write a setup program that will copy all needed driver files to a local directory on the end user system. People commonly use an InstallShield script for this purpose, but any solution that fulfills Microsoft's requirements for an application setup program will be OK.

3. As part of the setup program described in the preceding step, execute a call to *SetupCopyOEMInf* (documented as part of the platform SDK or in MSDN) with the *SPOST_PATH* option.

The TOASTPKG sample in the DDK contains a fully worked-out example of how to preinstall a signed driver package in Windows XP. The sample includes a CD-image directory with an AUTORUN.INF file that will autoplay a CD-ROM to run the preinstallation setup program.

A multifunction device often requires more than one INF file—typically, the parent and child devices belong to different setup classes. The main device's INF is the one that you should specify in a call to *SetupCopyOEMInf*, and it should itself contain *CopyInf* directives in the install section to copy the INF files to install child devices. The *CopyInf* directive was added to Windows XP. You can install COCPYINF.DLL as a device-specific co-installer. COCPYINF.DLL also implements the *CopyInf* directive. Full details about using this redistributable component of the DDK are in the file Tools\Coinstallers\x86\cocpyinf.htm.

15.4.3 Value-Added Software

You can save yourself some grief by not including any so-called value-added software (that is, applications and other user-mode components besides co-installers) in the *CopyFiles* directives within your INF file. The digital signature you obtain will cover any file that the INF file copies. Making any change at all to such a file invalidates your digital signature and requires you to obtain a new one from WHQL.

Microsoft recommends that you install valued-added software using a separate program, such as an InstallShield script. You can launch that separate program in a variety of ways, I suppose, but probably the most robust way is the one illustrated in the DDK's TOASTPKG sample. In the sample, a registry entry indicates whether the user has been given an opportunity to install

IPR2024-00292
KASPERSKY EX 1008 Page 425

the value-added package on a given system. If the flag is clear, a co-installer DLL presents an install wizard page to inquire whether the end user wants to install the value-added stuff. (Note that this dialog page will require a client-side install.) If the answer to *that* question is yes, the co-installer launches the value-added setup program from a subdirectory below the directory containing the INF file.

In TOASTPKG, the co-installer comes into play to install the value-added software only when the end user happens to install the hardware before running the preinstallation program that comes with the package. (The preinstallation program will do the value-added install or not, as specified by the user, and set the registry to prevent the co-installer from later presenting the wizard page. A server-side install is therefore possible after the preinstallation program has run.)

Envisioning and preparing for the various installation scenarios that might occur is a complex task. It's because of this complexity that I advised you in the "Management Overview and Checklist" section of Chapter 1 to plan ahead for the work required to prepare your installation package.

15.4.4 Installing a Driver Programmatically

If you have a non-PnP device, or if you have a PnP device for which you want to supply an updated driver, you might want to write a program to install the driver software without involving the user any more than necessary. The DEVCON sample in the DDK illustrates how to programmatically install or update driver software and to perform many other device-manager tasks in Windows 2000 and Windows XP. The FASTINST sample in the companion content to this book illustrates just the programmatic installation of driver software, but it applies to any WDM platform. I hope that you've been using FASTINST to install the sample drivers throughout this book.

The Windows 2000 and Windows XP version of FASTINST performs these steps:

1. Parses the INF file whose name you supply as a command-line argument to locate the first model statement, which determines the device identifier to be used later on. (You can run FASTINST from a command prompt and specify any device identifier you want, thereby overriding the default choice of the identifier in the first model statement.)
2. Constructs an empty device node having the device identifier determined by the first step.
3. Calls *UpdateDriverForPlugAndPlayDevices* to "replace" the driver for the empty device node using the INF file.

15.4.5 The *RunOnce* Key

You can place values in the *HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce* registry key to execute commands after setup finishes installing a device. Each value in *RunOnce* contains a command string. When setup finishes, it executes those commands and deletes the entries.

> **NOTE**
> The system also executes any commands that might be in the *RunOnce* key each time the system boots. The DDK indicates that *RunOnce* commands can be processed at other, unspecified times in addition. It adds, "If you place a command under the *RunOnce* key, you cannot predict when it will execute."

Here's an example, drawn from the AUTOLAUNCH sample in the companion content, of INF-file syntax for creating a *RunOnce* command:

```
[DriverInstall.ntx86]
CopyFiles=DriverCopyFiles,AutoLaunchCopyFiles
AddReg=DriverAddReg.ntx86

[DriverAddReg.ntx86]
HKLM,%RUNONCEKEYNAME%,AutoLaunchStart,, \
  "rundll32 StartService,StartService AutoLaunch"

[Strings]
RUNONCEKEYNAME="Software\Microsoft\Windows\CurrentVersion\RunOnce"
```

RUNDLL32 is a standard system utility that invokes a function within a DLL. If all the *RunOnce* commands you create involve RUNDLL32, a server-side install will still be possible using your INF. In fact, using any other verb in a *RunOnce* command causes your INF to flunk the WHQL tests.

The syntax of a RUNDLL32 command is this:

```
rundll32 dllname,entryname [cmdline]
```

where *dllname* is the name of a DLL (with or without the .DLL file extension), *entryname* is the name of an exported entry point, and *cmdline* is an optional command string to be processed by the DLL. The *entryname* function should be defined like this:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 426

```
[extern "C"] void CALLBACK entryname(HWND hwnd, HINSTANCE hInst,
  LPSTR cmdline, int nshow)
  {
  :
  }
```

Note that *CALLBACK* includes the *__stdcall* directive. The *hwnd* argument is the handle to a window that should be the parent of any user-interface elements the function creates, *hInst* is the instance handle of the DLL, *cmdline* is an exact copy of the like-named argument in the RUNDLL32 command, and *nshow* is one of the standard show-state values such as *SW_SHOWNORMAL*.

15.4.6 Launching an Application

As a special case of using *RunOnce*, I designed a scheme for the first edition to allow you to arrange to execute a value-added application each time (including the first) that the system starts your device. This scheme is shown in the AUTOLAUNCH sample in the companion content. In outline, the scheme is this:

- The INF file installs an AutoLaunch service that runs in user mode looking for arrival notifications on a particular interface GUID.
- When AutoLaunch receives an arrival notification, it consults the registry to obtain a command string, which it then executes on the interactive desktop.
- In order to get the application launched the very first time you install the device, the INF uses a *RunOnce* command to start the AutoLaunch service.

I've gotten very few questions from readers about AutoLaunch, so I judge that it's not important enough to describe here in detail.

15.5 The Windows Hardware Quality Lab

Microsoft really wants hardware devices and their associated drivers to meet certain minimum standards for quality, interoperability, and ease of use for consumers. To that end, Microsoft established the Windows Hardware Quality Lab (WHQL) in 1996. WHQL's basic mandate is to publish and administer an evolving set of Hardware Compatibility Tests for systems and peripherals. Successfully passing these tests confers three basic benefits:

- Entrée to various marketing programs, such as the "Designed for Windows" logo and various lists maintained by Microsoft.
- A digital signature for your driver package, which greatly eases installation on end user machines.
- Free distribution of your driver through Windows Update and other means.

Your starting point for working with WHQL is http://www.microsoft.com/hwdq/hwtest. As I remarked in Chapter 1, it's important to get started early in a development project because there are a number of legal and business hurdles to surmount before you even get to the point of asking WHQL to test your hardware and software. Once past those hurdles, you will need to acquire systems and hardware that you wouldn't necessarily need to own except that you need them for some of the prescribed tests for your class of device.

15.5.1 Running the Hardware Compatibility Tests

When you're ready to begin the WHQL certification process, you'll start by running the relevant Hardware Compatibility Tests. Microsoft distributes the HCT as part of certain MSDN subscriptions and beta programs. You can also download the tests over the Internet. The installation wizard allows you to pick one or more categories of test to perform. If you're just testing the driver for one device, I recommend that you select just the one test category that includes your device in order to minimize the number of meaningless choices that you might have to make later. For example, if you install tests for Pointing And Drawing Devices and Smart Card Readers, you'll have to pick one device in each category before the Test Manager will let you begin testing in either category.

To provide a concrete example, I decided to run the tests for a USB gaming mouse for which I wrote the driver. Figure 15-14 shows how I selected the relevant test category while installing the HCT.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 427

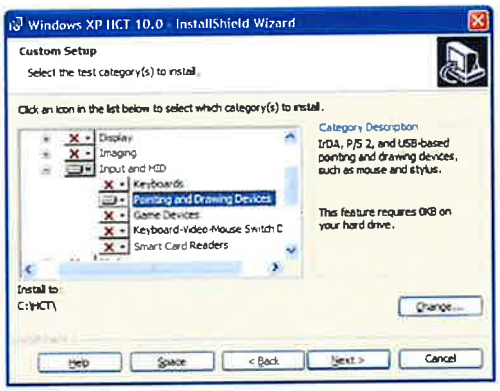


Figure 15-14. *Selecting a test category.*

The HCT setup program automatically kicks off a wizard that allows you to select a device for test. A dialog box reminds you that you need to have the hardware and software installed at this point. For each of the categories you installed, you'll fill in a dialog box like the one shown in Figure 15-15. My device appears as a "HID-compliant mouse." Microsoft is listed as the manufacturer because HCT thinks that my device uses the standard MOUHID.SYS driver. In fact, my mouse is a nonstandard HIDCLASS minidriver with many features that need to be tested beyond the basic things that HCT will test.

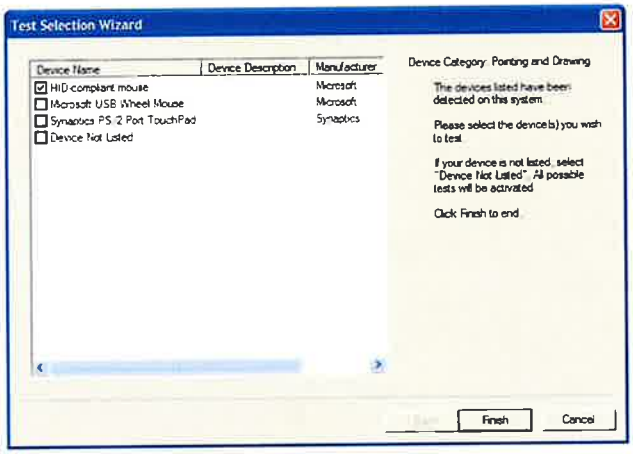


Figure 15-15. *Selecting a device for testing.*

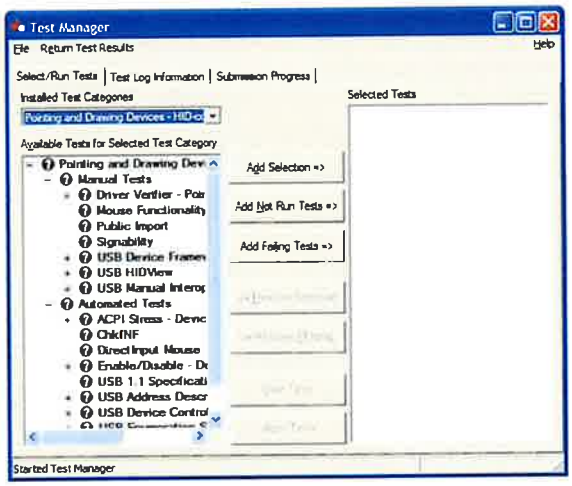


Figure 15-16. *The Test Manager dialog box.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 428

HCT presents several additional dialog boxes before it gets to the point where testing can begin. Figure 15-16 illustrates the basic test manager dialog box. Ideally, you would just press the button labeled "Add Not Run Tests," which would populate the right-hand pane with all of the tests. A bit of circumspection is called for here, however.

One of the tests—the ACPI stress test—runs for many hours, if it runs at all. Many computers can't run this test, and the laptop on which I was doing this testing is one of them. To run this test, you need XP *Professional* on a desktop system that supports the S1 *and* S3 states or a notebook that supports S1 *or* S3. (I was using Windows XP Home Edition because USB wake-up stopped working on the notebook if I upgraded, and USB wake-up testing was the only reason I bought that particular notebook.) I suspect that I'll never own a computer that can run this test because I tend to buy computers with the operating system preinstalled and then upgrade the operating system as part of a beta program, whereupon power management stops working.

The USB Manual Interoperability test requires several hundred dollars' worth of multimedia hardware that I would have no use for beyond running this one test suite. (Figure 15-17 is a screen shot from a test run when I made the mistake of allowing this test to commence.) This test is pretty important from the hardware point of view because it verifies that commonly used USB devices will continue to work with your device plugged in and vice versa.

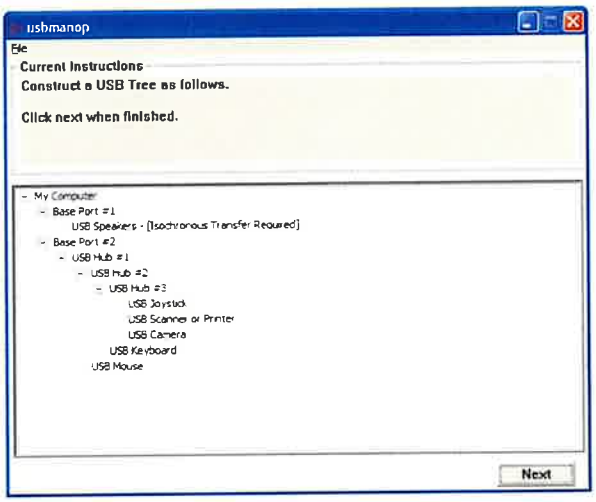


Figure 15-17. *Required hardware topology for the USB Manual Interoperability test.*

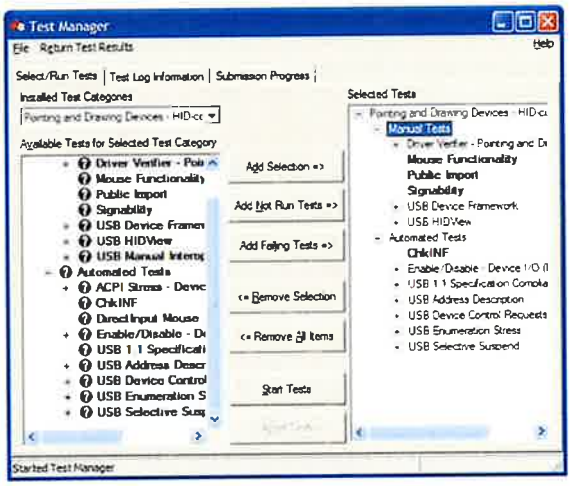


Figure 15-18. *I'm ready to start testing...*

Others of the tests are actually useless for telling me anything about the quality of my own driver. The DirectInput Mouse test verifies that Microsoft's drivers interact correctly with DirectInput, a fact I never doubted. The USB Selective Suspend test isn't currently very important for a HID device because HIDCLASS never suspends a device in the first place: most devices can't wake up without losing an input event. In fact, all of the automated USB tests relate to hardware issues. I decided to let them run in this particular example because I was working closely at the time with a leading firmware engineer in getting this product to market. When I was done selecting the tests that I expected to be able to perform—whether or not they would succeed was a different question, to which I actually wanted the answer—my Test Manager dialog box looked as shown in Figure 15-18.

IPR2024-00292
KASPERSKY EX 1008 Page 429

The very first thing the test engine does is engage the driver verifier—on the wrong drivers—and reboot the computer. Remember that HCT thinks MOUHID.SYS is the driver for my mouse. In reality, the verifier should be getting turned on for my minidriver instead. Attempting to do that by hand would invalidate the test run, though, so I allowed the test run to continue. I'm told that newer versions of the HCT will do a better job of identifying which driver needs to be tested. I later ran tests with the verifier turned on for my driver. It was a good thing I did because I caught a rookie mistake in the way my HID minidriver was forwarding a device *IRP_MJ_POWER* with the minor function code *IRP_MN_SET_POWER* and the power type *SystemPowerState* after waiting for its interrupt-endpoint polling interrupt request packet (IRP) to finish.

The Mouse Functionality test (see Figure 15-19) is the one most relevant to the quality of my driver in that it verifies whether I am actually delivering mouse reports in the format expected by the system. Because my mouse lacks an actual wheel (users can program some of its buttons to act as a wheel), I had to fudge part of the functionality test with another mouse attached to the same system.

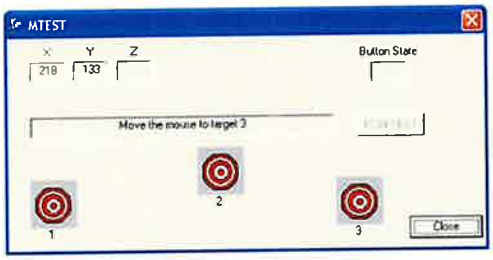


Figure 15-19. *The Mouse Functionality test.*

The Public Import and Signability tests both asked whether my product "installs it's *[sic]* own driver." I answered that it does and pointed the test engine to a directory where I had placed my INF and all the other files that get installed on any platform. The import test verified that my driver wasn't calling any *verboten* kernel-mode functions. The signability test verified, among other things, that all files copied by my INF file were in fact present. (Recall that the CHKINF doesn't do this.)

The CHKINF test ran CHKINF on the wrong INF file, namely the Microsoft-supplied INPUT.INF. Being a good citizen, I ran CHKINF myself. The PERL test script initially failed because it lacked a copy of STRICT.PM, which I found in the HCT directory and copied by hand. The test report told me that a *RunOnce* entry running CONTROL.EXE (my solution to a client request to automatically launch their control panel) was not allowed because it didn't involve RUNDLL32. Since I had always regarded that particular client request as a bad idea, I resolved to use the test failure as a lever to get my client to change his mind. Mind you, I'm sure I could have thought of a way to use RUNDLL32 to launch a control panel applet, but doing that would defeat the real but unstated goal of the test, which is to make sure that a server-side install can proceed without the intrusion of user-interface elements.

The remainder of the tests I scheduled happened without my needing to intervene, which is why I guess they're called automated tests. In the end, I got the test log shown in Figure 15-20.

The reason that the Enable/Disable test failed to generate a log is that it generated an exception in user mode. Some part of the test engine caught the exception and silently terminated that test.

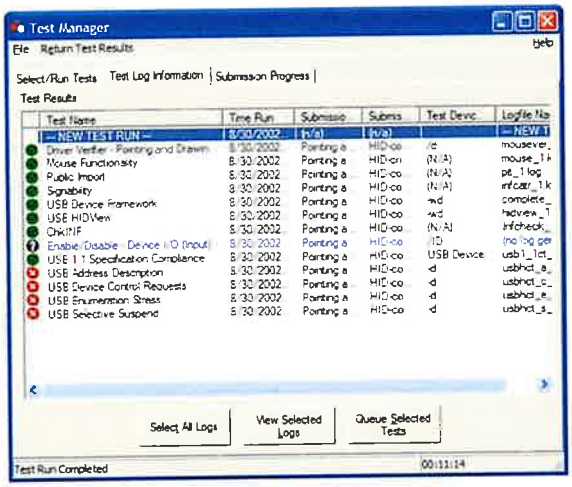


Figure 15-20. *Test results after running selected tests.*

I worked with my firmware engineer colleague to iron out the failures in the various USB tests. In doing this, it would have been very helpful to correlate test failures with the HCT documentation entries for the same tests. For example, the USB

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 430

Address Description test log referred to a test assertion numbered 9.22.6. After opening the HCT 10.0 documentation from the Start menu, I browsed to the section labeled Resources/WHQL Test Specification/Chapter 9 USB Test Specification/USB Test Assertions/Address Test Assertions, where I found the information shown in Figure 15-21. Test assertion number 9.22.6 is, uh, well, something important, probably.

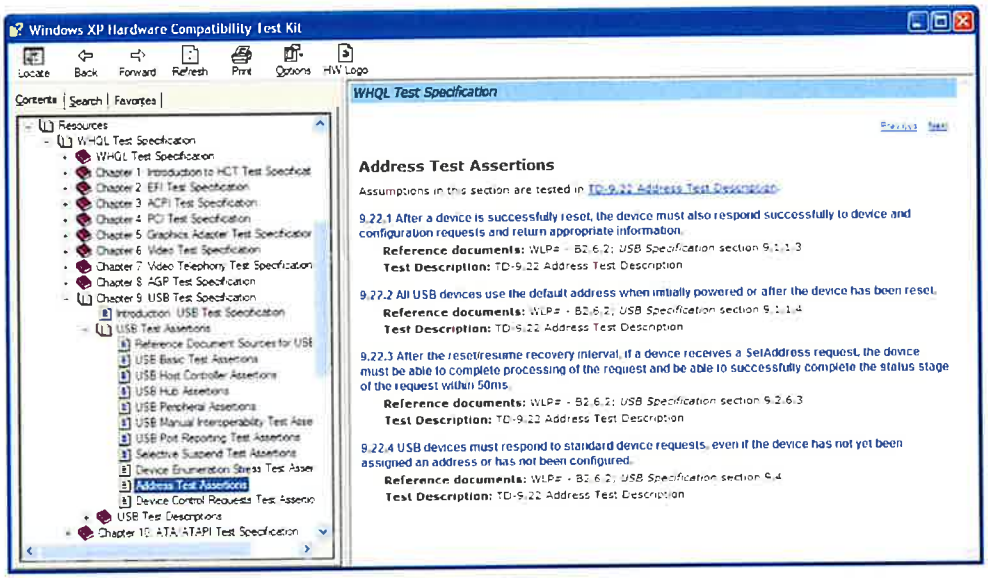


Figure 15-21. *Documentation for test assertions.*

You'll notice that many things went wrong in the testing process. To summarize:

- I couldn't run some of the tests because of hardware or budget limitations. I wouldn't be able to put together a WHQL submission for my client. As it turns out, he doesn't have the resources either and will have to hire an outside contractor who specializes in WHQL testing. He doesn't actually want a logo, though. In fact, the counterculture he sells into would prefer that his mouse not have a logo. He needs a digital signature, though, because of the driver ranking problem discussed earlier in this chapter.
- One of the tests failed on its own for unguessable reasons.
- A few of the tests were testing the wrong thing.
- A few of the failed test assertions I encountered weren't documented.

What you would do in a similar situation is ask for help. WHQL personnel monitor several newsgroups on the msnews.microsoft.com news server, including microsoft.public.development.device.drivers and microsoft.public.windowsxp.winlogo. WHQL also responds to e-mail requests for assistance at addresses accessible from the WHQL home page, http://www.microsoft.com/hwdq/hwtest.

15.5.2 Submitting a Driver Package

The last step in running the Hardware Compatibility Tests would be to create a WHQL submission package. You'll want to do this separately for each operating system that your driver supports and then gather together the resulting CAB files in one convenient place. Your next step, which I think you should actually have performed months prior, would be to visit http://winqual.microsoft.com and get yourself signed up as a WHQL client company.

Given a login ID and a password, you can log on through the winqual page to do any of several things:

- You can create a new submission package.
- You can review the status of a previous submission.
- You can retrieve error reports that users worldwide have submitted that apparently arise from your product.

For this chapter, I wanted to create a new submission for a new hardware device. Figure 15-22 is a screen shot showing the starting point for a brand-new submission.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 431

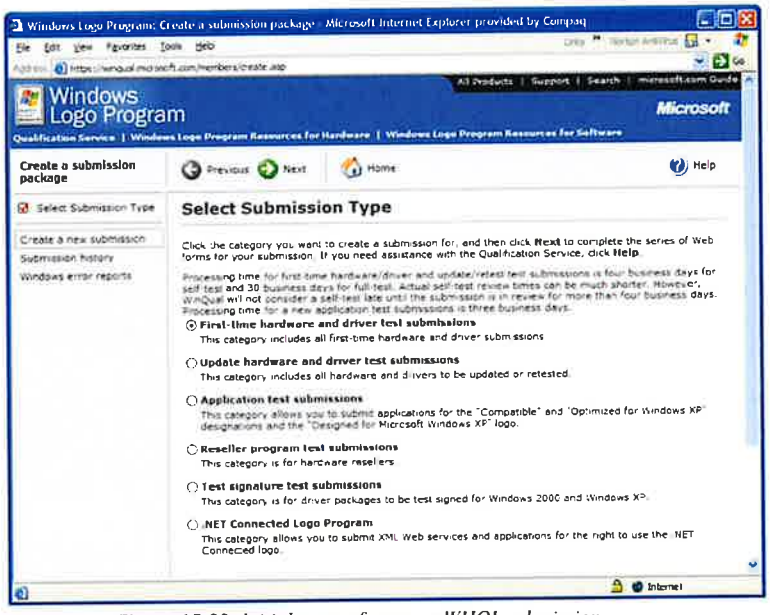


Figure 15-22. *Initial screen for a new WHQL submission.*

From the point shown in Figure 15-22, Web forms lead through the process of characterizing your submission in a relatively painless way. You'll answer questions such as these:

- What kind of product is it? I said my product was in the Input/Pointing Drawing class, which is the same as the test category I used when I was running the HCT.
- With which operating systems will the product be used? You want to be sure you've run the relevant HCT on all the platforms you select because you'll later have to identify the test results for each of them.
- What are two e-mail contacts (including yourself) for communications related to the submission? I'm not sure what you do if you're a one-person company. (I was doing my testing as a nominal member of a dummy company that the WHQL folks use for their own internal testing, so I didn't have a chance to see how this particular problem would be resolved.)
- What, exactly, is your product? (See Figure 15-23.)

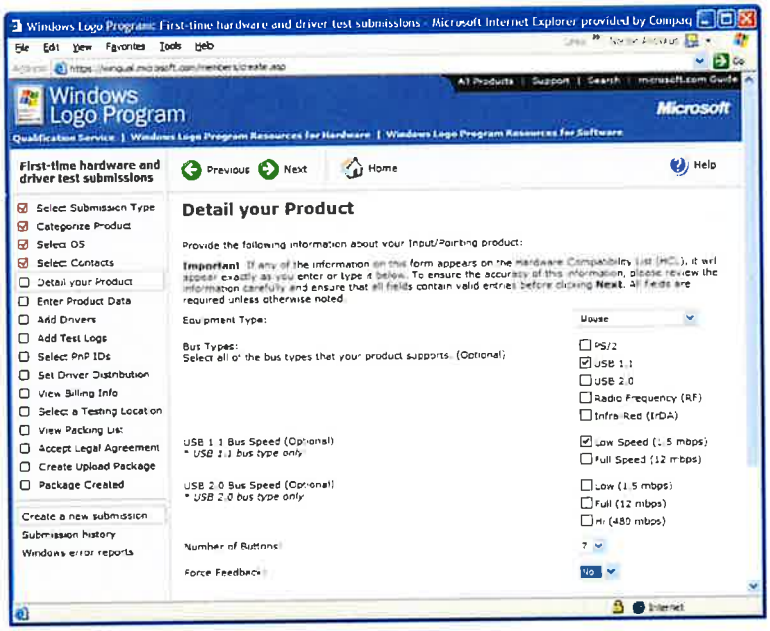


Figure 15-23. *Detailing the product.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 432

- What is the name of your product, when will it be released, and which platforms are supported? There are rules about what constitutes an acceptable product name too. I could not have said just "mouse." I entered a description that included the manufacturer's model number and generic description ("rotary gaming mouse"). The question about platform support is different from the earlier question about operating systems in that it includes several varieties of each system. For example, you can specify that the product support Windows XP Home Edition but not Windows XP Professional, and so on.

- Where are the driver packages for each operating system? In answering this question, you supply, for each operating system, the name of a directory tree that contains all the files that will be covered by the eventual signature file. That is, the directory tree includes the INF file or files and all files installed by those INF files. The easiest and best way to perform this step is to create a directory tree that mirrors your distribution media layout and contains all the files, and nothing but the files, that are destined for the end user's machine. As part of this step, you also get to specify the languages supported by each driver package.

- Where are the test logs for each operating system? You can't have more than one in the same directory because of filename conflicts, by the way.

- For which hardware (PnP) IDs do you want Microsoft to distribute drivers via Windows Update? There are additional requirements to using Windows Update, by the way, but this step is one not to miss. See Figure 15-24.

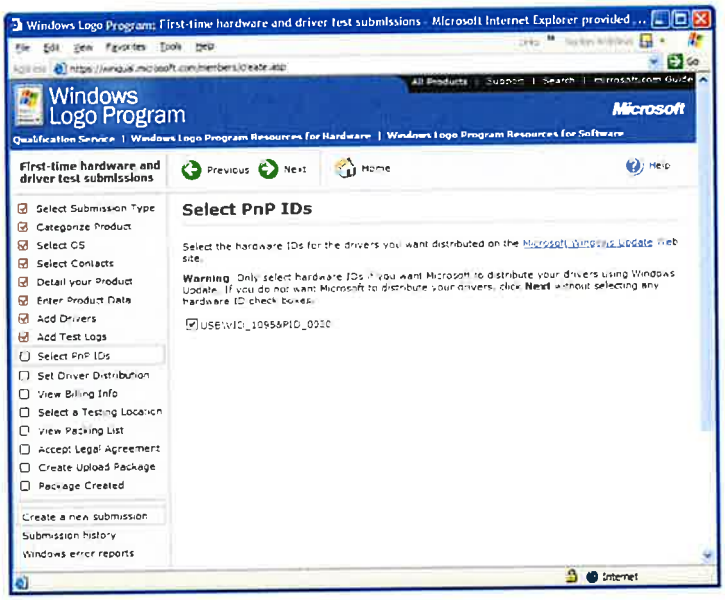


Figure 15-24. *Specifying PnP IDs for Windows Update.*

- How will the driver be distributed? There are many choices, all of which are contingent on you having the right to distribute the driver. See Figure 15-25.

IPR2024-00292
KASPERSKY EX 1008 Page 433

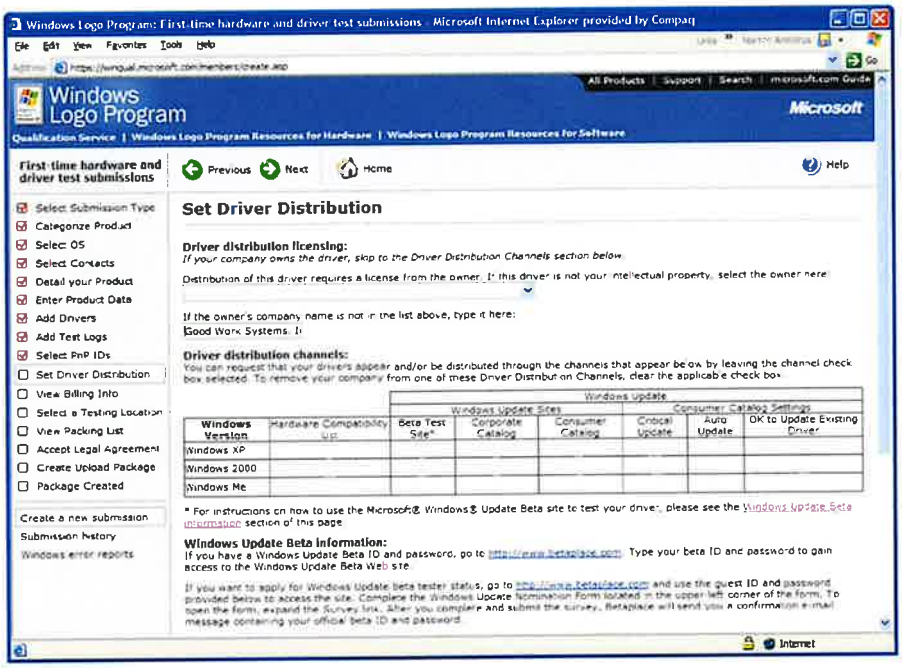


Figure 15-25. *Specifying driver distribution channels.*

- How do you intend to pay for the testing? Say what? You mean this isn't free? At the time I'm writing this, my test submission would have cost $250 for each of the operating systems (Windows XP, Windows 2000, and Windows Me) that I claimed to support, for a total of $750. The cost for retest submissions is the same, so you don't want to submit obviously flawed packages.

- Where do you want the testing performed? There are several WQHL testing sites around the world. In the United States, you'd pick the one in Redmond. This question is actually relevant only if your submission requires hardware. Mine didn't. (See Figure 15-26.) In fact, most WHQL tests at the time I'm writing this are self-test programs and don't require you to submit hardware.

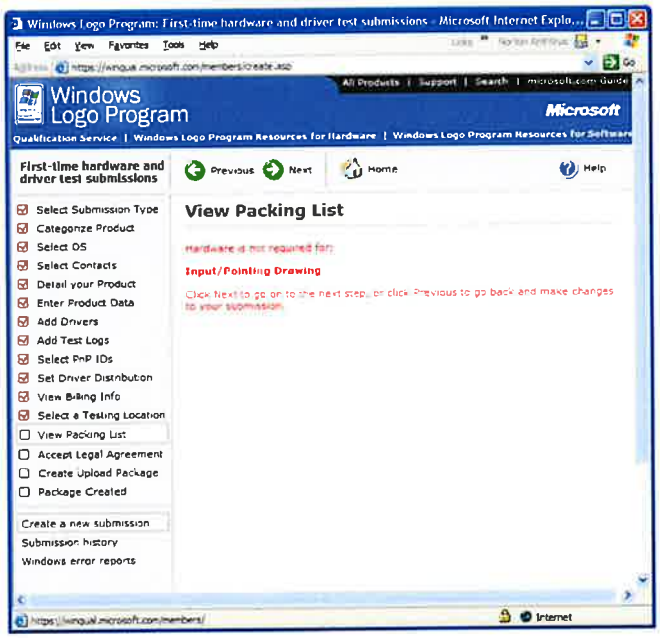


Figure 15-26. *The packing list.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 434

- Does the party of the first part (hereinafter known as the party of the first part) agree that, and so on? Yes, there is a legal agreement that you have to sign.
- Is this your final answer? That is, do you need to correct the driver and test log locations you specified earlier? Speak now, or forever hold your peace….
- Ta-da! (See Figure 15-27). You're done. You can digitally sign your submission package and upload it to WHQL. This is where I had to stop. Not only did I have a submission package with fatal omissions in it, but also I didn't have (and didn't want to go to the considerable trouble of obtaining) a VeriSign ID. If I get much more stubborn and independent, I'll have to move to a cabin in Idaho and use the Internet with a dial-up modem the way our pioneer forebeings did.

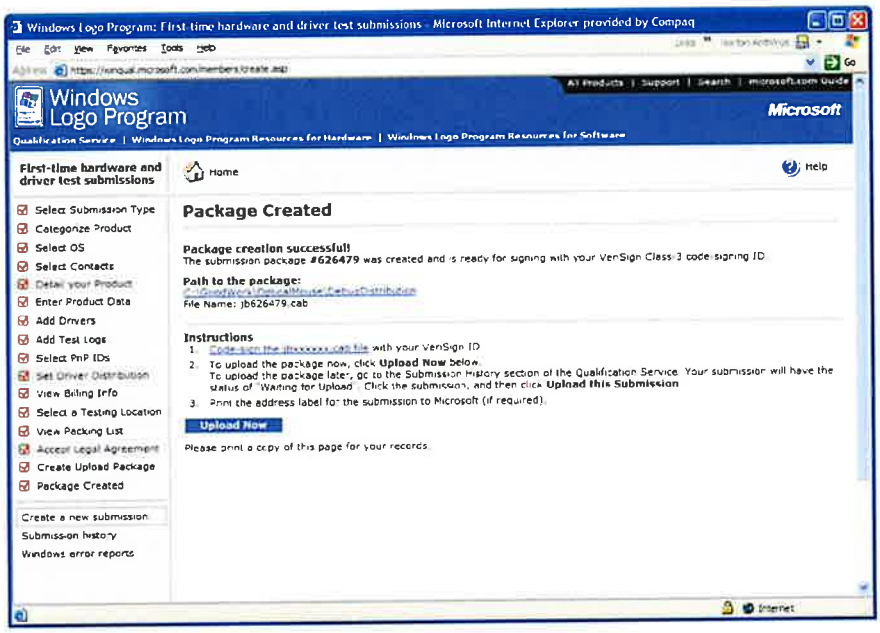


Figure 15-27. *Ready to sign and submit.*

I learned a few tricks in the process of running through the Web forms for the first time. As I mentioned, you want to be sure to have all the distribution packages and test results handy. You have plenty of time to finish the process, but the Web application will time out after about an hour—so don't plan on having a power lunch in between steps. Some of the choices you make can't be undone except by backing up. Choosing the wrong directories for certain options can add hours to the process if it forces the application to navigate large directory trees in its search for files. The forms warn you about the last two of these gotchas, so I don't think you're likely to go wrong.

15.6 Windows 98/Me Compatibility Notes

Windows 98/Me uses completely different technology for installing and maintaining devices than does Windows XP. In this section, I'll describe some of the ways this might affect you.

15.6.1 Property Page Providers

A property page provider for a new device class must be a 16-bit DLL. Look at SAMCLS16 in the companion content if you want to see an example, and don't discard your 16-bit compiler just yet!

15.6.2 Installers and Co-installers

A class installer for Windows 98/Me is a 16-bit DLL and uses functions from SETUPX.DLL, which are documented in MSDN rather than in the DDK. I know of no samples to give guidance in writing one. Windows 98/Me doesn't support co-installer DLLs.

15.6.3 Preinstalled Driver Packages

SetupCopyOEMInf isn't implemented in Windows 98 or Windows Me. To preinstall a driver package on these platforms, simply copy the files to the correct locations and design the INF file not to require any file copies.

IPR2024-00292
KASPERSKY EX 1008 Page 435

15.6.4 Digital Signatures

Windows 98 doesn't use digital signatures. Windows Me won't install an unsigned driver to replace a signed driver for audio and certain other multimedia devices or for a display adapter. The DDK describes other rules for when Windows Me uses digital signatures. I confess I couldn't understand all the qualifications and special cases, so I can't explain them to you.

15.6.5 Installing Drivers Programmatically

Because the Windows 98/Me setup program is 16-bit, and because *UpdateDriverForPlugAndPlayDevices* isn't implemented in those systems, relatively heroic means are required to programmatically install a driver for a non-PnP device. I cobbled together the Windows 98/Me version of FASTINST in the companion content by trial and error because the documentation of the 16-bit setup functions is pretty sparse. The basic trick is to construct a Device Information structure for the INF file, restrict it to the driver for the decided-upon device identifier, and call *DiInstallDevice*. Needless to say, you'll need to get a bunch of details just right to make this process work.

15.6.6 CONFIGMG API

In Windows 98/Me, you often need to call entry points in the protected-mode API exported by CONFIGMG.VXD. The functions you call are documented in the Windows 95 DDK as ring-0 service calls whose names begin with *CONFIGMG_*. The API functions you call from ring 3 have the same names and arguments except with the prefix *CM_*. The SAMCLS16.DLL sample in the companion content contains a few CONFIGMG calls that illustrate the mechanics of making these calls.

15.6.7 INF Incompatibilities

Windows 98/Me uses parsing technology and setup libraries that are completely different from those of Windows 2000 and later systems. Consequently, there are many restrictions on how you write an INF file that will be used in both environments. Here's a partial list of them:

- Section names are limited to 28 characters.
- Windows 98/Me ignores Unicode INF files and decorated [Strings] sections. Thus, to localize an installation for Windows 98/Me, you have to provide a completely new INF file (which requires a new WHQL signature). These facts provide additional reasons for not marketing your driver for Windows 98 or Windows Me on the planet Chronos.
- Windows 98/Me doesn't append platform or operating system decorations to the names of sections. Generally, this behavior means that you use undecorated section names for Windows 98/Me and decorated names for the later systems.
- Windows 98/Me doesn't combine identically named sections into one section as do Windows 2000 and later systems. Instead, it picks the first one.
- Windows 98/Me requires a [ClassInstall] section to define a new setup class.
- The Windows 98/Me setup program doesn't handle long filenames (that is, names with components longer than 8 characters) in the first instance. For example, if you specify mydriverfile.sys in a CopyFiles section, setup will present a dialog box saying it can't find the file, even when the file exists. You can then press the OK button, whereupon setup will happily copy the file. A similar issue arises with the directory path in a [SourceDisksFiles] directive. You might notice that all the sample INF files specify objchk~1\i386 instead of objchk_wxp_x86\i386, so you won't be discommoded by this quirk.

15.6.8 Registry Usage

Windows 98/Me uses a registry structure that's completely different from that of Windows XP:

- Hardware keys are found in *HKLM\Enum*.
- There is no separate hardware parameters subkey of the hardware key. Instead, standard and nonstandard properties are mixed together in the hardware key.
- Class keys are found in *HKLM\System\CurrentControlSet\Services\Class*. They're identified merely by the class name. (There is a dummy key under *Class* that has the class GUID as its name, but it's not important.)
- Driver keys are numbered subkeys of the class key, as in Windows XP.
- Although there can be a service key, it's not very important in Windows 98/Me. Instead, the Configuration Manager relies on entries in the driver key to load the right driver.

To initialize the driver key, your Windows 98/Me install section should have an *AddReg* directive similar to this example:

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 436

```
[DriverInstall]
AddReg=DriverAddReg
<other install directives>
[DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,pktdma.sys
```

That is, you designate NTKERN.VXD as the device loader for your device, and you designate your WDM driver as the *NTMPDriver* for which NTKERN looks.

15.7 Getting Device Properties

Windows 98/Me incorrectly implements *IoGetDeviceProperty* for the *FriendlyName* property. To retrieve the friendly name in a WDM driver, you should use *IoOpenDeviceRegistryKey* and interrogate the property by name. The DEVPROP sample illustrates how to do this.

IPR2024-00292
KASPERSKY EX 1008 Page 437

Chapter 16

Filter Drivers

The Windows Driver Model assumes that a hardware device can have several drivers, each of which contributes in some way to the successful management of the device. WDM accomplishes the layering of drivers by means of a stack of device objects. I discussed this concept in Chapter 2. Up until now, I've been talking exclusively about the *function driver* that manages the main functionality of a device. To round out this book, I'll describe how you write a *filter driver* that resides above or below the function driver and modifies the behavior of the device in some way by filtering the I/O request packets (IRPs) that flow through it.

16.1 The Role of a Filter Driver

A filter driver that's above the function driver is called an *upper filter driver*; a filter driver that's below the function driver (but still above the bus driver) is called a *lower filter driver*. Please refer to Figure 2-2 for an illustration of this layering. The mechanics of building either type of filter are exactly the same, even though the drivers themselves serve different purposes. In fact, you build a filter driver just as you build any other WDM driver—with a *DriverEntry* routine, an *AddDevice* routine, a bunch of dispatch functions, and so on.

16.1.1 Upper Filter Drivers

Recall that the I/O Manager sends IRPs for a device to the topmost filter device object (FiDO) in that device's PnP stack. Upper filter drivers therefore see all requests before the function driver does and can edit the stream of requests in any desired way. Among the things you can accomplish with an upper filter are these:

- Implement some consistent upper-edge interface for wildly different underlying hardware. I'll say more about this concept in the following paragraphs.
- Gather and report metering information, say, in response to Windows Management Instrumentation (WMI) requests.
- Work around a bug in a function driver.

Examples of upper filter drivers abound in Windows XP. I'll describe a few of them. I suggest you use the DevView utility mentioned in Chapter 2 in the companion content to examine some of the device stacks to better visualize the examples.

Disk and Tape Storage Driver Stacks

Several completely different technologies are in common use for disk and tape devices, including SCSI, IDE, USB, and 1394. Function drivers for these devices are therefore going to be very different at their lower edge, where they talk to their hardware. To simplify the overall system architecture, each disk or tape function driver exports a SCSI upper edge. That is, drivers higher in the stack send SCSI Request Blocks (SRBs) to the function driver. The driver for an actual SCSI device simply extracts standardized Command Description Blocks (CDBs) from the SRBs and sends them to the hardware more or less directly. Drivers for other types of device translate the CDBs into their own hardware protocol.

NOTE
Strictly speaking, there is no "function" driver in a storage stack because none of the drivers in the stack is so designated in the registry. The bus driver at the bottom of the stack acts as a function driver. All the other drivers in the stack are technically upper filter drivers.

Microsoft has implemented three upper filter drivers—DISK.SYS, CDROM.SYS, and TAPE.SYS—that implement the "diskness" or "tapeness" of hardware. Microsoft refers to these as class drivers, but here the word *class* is used differently than we've been considering in this book. Elsewhere, we've talked about class/minidriver pairs that add up to a function driver. Here the word just refers to a filter driver that implements an upper-edge interface appropriate to a class of devices.

The upper edge of the storage class drivers presents a unified disk, CD-ROM, or tape interface consisting of certain I/O control (IOCTL) requests and a defined behavior for read and write requests. At their lower edge, these drivers talk SCSI to whoever is underneath them in the stack.

Figure 16-1 shows the driver stack for a SCSI hard disk on one of my systems. At the bottom of the stack is AIC78U2.SYS, a SCSI miniport driver that works with a class driver named SCSIPORT.SYS to manage the particular SCSI adapter that came with my computer. DISK.SYS makes the hard disk look like a disk to everyone above. PARTMGR.SYS manages the two partitions that the manufacturer created on the disk.

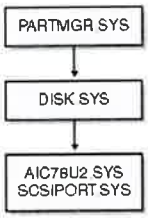


Figure 16-1. *Driver stack for a hard drive.*

Figure 16-2 shows the driver stack for an IDE-based DVD-ROM drive on a different computer. Here ATAPI.SYS is at the bottom of the stack. At its upper edge, it looks like a SCSI port driver. At its lower edge, it acts like an IDE controller. Layered on top of ATAPI.SYS are IMAPI.SYS, CDROM.SYS (presents an ISO-standard CD-ROM image at its upper edge), and REDBOOK.SYS (implements the Redbook audio standard).

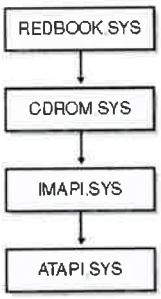


Figure 16-2. *Driver stack for a DVD-ROM drive.*

The PnP Manager constructs these driver stacks based on the type of device reported by the bus driver. In the hard-disk example (Figure 16-1), the device is class Disk. DISK.SYS and PARTMGR.SYS are listed as upper filters in the Disk class key. In the DVD-ROM example (Figure 16-2), the device is class Cdrom. REDBOOK.SYS and IMAPI.SYS are listed as upper and lower filters, respectively, in the hardware key; CDROM.SYS is listed as an upper filter in the Cdrom class key.

Mouse and Keyboard Drivers

Two connection technologies—universal serial bus (USB) and 8042—are commonly used nowadays for keyboards and mice in Windows systems. Figure 16-3 illustrates the driver stack for a USB mouse. HIDUSB.SYS (a HIDCLASS minidriver) plays the role of a bus driver, while MOUHID.SYS acts as the function driver. MOUCLASS.SYS is a class upper filter for the Mouse class.

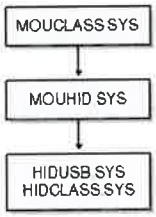


Figure 16-3. *Driver stack for a USB mouse.*

Figure 16-4 is a parallel diagram for a PS/2 mouse on a different system. I8042.SYS is the function driver for the PS/2 mouse port. On this particular system, ACPI.SYS (the driver with overall responsibility for power management) acts as the bus driver.

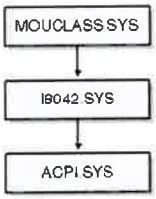


Figure 16-4. *Driver stack for a PS/2 mouse.*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 440

Evidently, both driver stacks have MOUCLASS.SYS in common. MOUCLASS presents a consistent mouse interface to the rest of the system, which is how it comes to pass that you can use either type of mouse (or both at the same time) on a Windows computer.

For a keyboard, the only important difference in the driver stacks would be that KBDCLASS is at the top.

Serial Enumerator

You often plug other things in to a serial port. The most common things, of course, are modems and mice. Until the advent of USB, the serial port was also a popular place to attach a variety of other peripherals. Microsoft long ago defined a protocol whereby serial-attached devices can provide a Plug and Play identifier. The document "Plug and Play External COM Device Specification" describes the protocol but (because of indexing difficulties posed by the fact that the word "serial" doesn't appear in the title) is nearly impossible to find. I found a copy at http://www.microsoft.com/hwdev/resources/specs/pnpcom.asp, but it will probably have moved by the time you go looking for it.

Microsoft implements the serial PnP protocol in a device upper filter named SERENUM.SYS. SERENUM uses the standard serial port interface to talk downward to a serial port driver. If it detects a PnP identifier string, it acts as a bus driver by creating a physical device object (PDO). Thereafter, the PnP Manager loads function and filter drivers in the ordinary way.

Figure 16-5 shows the driver stack for a serial mouse. SERENUM.SYS is in the figure twice, once as a device upper filter for the serial port (whose function driver is SERIAL.SYS) and once as the bus driver at the bottom of the SERMOUSE stack. Note once again that MOUCLASS.SYS gets into the act to present that consistent mouse interface to the rest of the system.

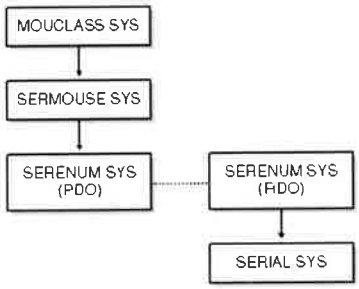


Figure 16-5. *Driver stack for a serial mouse.*

The DISKPERF Driver

The DISKPERF driver in the DDK illustrates another use for an upper filter. DISKPERF is an optional driver that collects statistics about the disk I/O requests that pass through it. DISKPERF reports these statistics by means of WMI requests that any WMI-based performance monitor can use.

16.1.2 Lower Filter Drivers

Lower filter drivers are much less common than upper filters. Since (by definition) a lower filter driver is below the function driver in the PnP stack, it receives only those IRPs that the function driver chooses to send down. For a device attached to a traditional bus such as Peripheral Component Interconnect (PCI), the function driver will consume all the interesting IRPs by making hardware abstraction layer (HAL) calls to do actual I/O. The only IRPs that are likely to flow down from the function driver are power and PnP. Generally speaking, there's not much a filter driver can or should do about those.

You might design a system of drivers as shown in Figure 16-6. The idea is to create a single function driver that uses lower filter drivers to communicate to hardware on various bus architectures. The drivers for the built-in modem on one of my notebook computers use this idea. The function driver for the modem is the Microsoft-supplied MODEM.SYS. The modem vendor supplies a lower filter driver that supports the standard serial port interface and two additional lower filters for reasons that don't immediately meet the eye. This idea is really the same as the concept underlying DISK.SYS and MOUCLASS.SYS—only the technical names assigned to the drivers are different. The really important thing is the way the drivers are layered, of course, and not their names.

I sometimes use a lower filter driver to "snoop" on the USB traffic generated by a function driver. Later in this chapter, I'll describe a lower filter I built to help me debug power-management problems in my own drivers; its job in life is to print debugging messages about all the *IRP_MJ_POWER* requests that the function driver generates or passes down.

Appendix A describes a lower filter driver named WDMSTUB.SYS. WDMSTUB defines a bunch of kernel functions that Windows 98 and Windows Me don't export. Making it a lower filter means that those functions get defined before the system tries to load a function driver that calls those functions. This fact in turn makes it possible to have true binary portability between Windows 2000, Windows XP, and Windows 98/Me.

IPR2024-00292
KASPERSKY EX 1008 Page 441

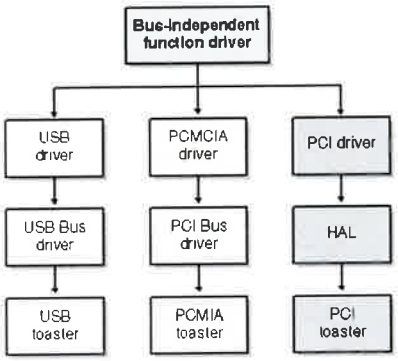


Figure 16-6. *Using lower filter drivers to achieve bus independence.*

16.2 Mechanics of a Filter Driver

In this section, I'll describe the mechanics of building a filter driver. As I've said several times, a filter driver is just another kind of WDM driver that has a *DriverEntry* routine, an *AddDevice* routine, dispatch functions for PnP and Power IRPs, and so on. The devil, as usual, is in the details.

On the CD The FILTER sample in the companion content illustrates the points discussed in this section.

16.2.1 The *DriverEntry* Routine

The *DriverEntry* routine for a filter driver is similar to that for a function driver. The major difference is that a filter driver must install dispatch routines for every type of IRP, not just for the types of IRP it expects to handle:

```
extern "C" NTSTATUS DriverEntry(PDRIVER OBJECT DriverObject,
  PUNICODE STRING RegistryPath)
  {
  DriverObject->DriverUnload = DriverUnload;
  DriverObject->DriverExtension->AddDevice = AddDevice;
  for (int i = 0; i < arraysize(DriverObject->MajorFunction); ++i)
    DriverObject->MajorFunction[i] = DispatchAny;
  DriverObject->MajorFunction[IRP MJ POWER] = DispatchPower;
  DriverObject->MajorFunction[IRP MJ PNP] = DispatchPnp;
  return STATUS SUCCESS;
  }
```

A filter driver has a *DriverUnload* and an *AddDevice* function just as any other driver does. I filled the major function table with the address of a routine named *DispatchAny* that would pass any random request down the stack. I specified dispatch routines for power and PnP requests.

The reason that a filter driver has to handle every conceivable type of IRP has to do with the order in which driver *AddDevice* functions get called vis-à-vis *DriverEntry*. In general, a filter driver has to support all the same IRP types that the driver immediately underneath it supports. If a filter were to leave a particular *MajorFunction* table entry in its default state, IRPs of that type would get failed with *STATUS_INVALID_DEVICE_REQUEST*. (The I/O Manager includes a default dispatch function that simply completes a request with this status. The driver object initially comes to you with all the *MajorFunction* table entries pointing to that default routine.) But you won't know until *AddDevice* time which device object or objects are underneath you. You can investigate the dispatch table for each lower device driver inside *AddDevice* and plug in the needed dispatch pointers in your own *MajorFunction* table, but remember that you might be in multiple device stacks, so you might get multiple *AddDevice* calls. It's easier just to declare support for all IRPs at *DriverEntry* time.

16.2.2 The *AddDevice* Routine

Filter drivers have *AddDevice* functions that get called for each appropriate piece of hardware. You'll be calling *IoCreateDevice* to create an unnamed device object and *IoAttachDeviceToDeviceStack* to plug in to the driver stack. In addition, you'll need to copy a few settings from the device object underneath you:

```
NTSTATUS AddDevice(PDRIVER OBJECT DriverObject, PDEVICE OBJECT pdo)
  {
  PDEVICE_OBJECT fido;
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 442

```
NTSTATUS status = IoCreateDevice(DriverObject,
  sizeof(DEVICE EXTENSION), NULL, GetDeviceTypeToUse(pdo),
  0, FALSE, &fido);
if (!NT SUCCESS(status))
  return status;
PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
do
  {
  pdx->DeviceObject = fido;
  pdx->Pdo = pdo;
  PDEVICE OBJECT fdo = IoAttachDeviceToDeviceStack(fido, pdo);
  pdx->LowerDeviceObject = fdo;
   fido->Flags |= fdo->Flags &
    (DO_DIRECT_IO | DO_BUFFERED_IO | DO_POWER_PAGABLE);
  fido->Flags &= ~DO DEVICE INITIALIZING;
  }
while (FALSE);
if (!NT SUCCESS(status))
  IoDeleteDevice(fido);
return status;
}
```

The parts that are different from a function driver are shown in boldface. Basically, we're using a peculiar method to determine the device type, and we're propagating a few flag bits from the next device object beneath us.

GetDeviceTypeToUse is a local function that determines the device type of the device object immediately under ours. We haven't yet called *IoAttachDeviceToDeviceStack*, so we don't have our regular *LowerDeviceObject* pointer. *GetDeviceTypeToUse* uses *IoGetAttachedDeviceReference* to get a pointer to the device object that's currently at the top of the stack rooted in our PDO, and it returns that device object's type. The reason we do this in the first place is that if we happen to be filtering a disk storage device object, we must have the correct type code in our call to *IoCreateDevice* so that the I/O Manager will create an auxiliary data structure known as a Volume Parameters Block (VPB). Without a VPB on every device object in the stack, some Windows 2000 file system drivers might crash later on.

We specify 0 for the device object characteristics. The PnP Manager will propagate any crucial characteristics flags up and down the device stack automatically. It would be wrong for a filter driver to force the *FILE_FLAG_SECURE_OPEN* flag, which applies to the whole driver stack, except for the purpose of fixing a bug in a function driver that forgets to set this flag.

We copy the buffering flags from the lower device object because the I/O Manager bases some of its decisions on what it sees in the topmost device object. In particular, whether a read or write IRP gets a memory descriptor list or a system buffer depends on what the top object's *DO_DIRECT_IO* and *DO_BUFFERED_IO* flags are. The reason a function driver must set one or the other of these flags at *AddDevice* time and can't change its mind later should now be clear: a filter driver will copy the flags at *AddDevice* time and won't have any way to know that a lower driver has changed them.

We copy the *DO_POWER_PAGABLE* flag from the lower device object to satisfy an obscure restriction imposed by the Power Manager. Refer to the sidebar for an explanation of the restriction. We will deal with another aspect of the same problem in our *IRP_MJ_PNP* dispatch routine. We don't need to propagate the *DO_POWER_INRUSH* flag because the Power Manager needs that flag set in only one device object.

The *DO_POWER_PAGABLE* Flag

Drivers must actively manage the *DO_POWER_PAGABLE* flag to accommodate some quirks in the Windows 2000 Power Manager. If this flag is set in a device object, the Power Manager will send *IRP_MN_SET_POWER* and *IRP_MN_QUERY_POWER* requests to the corresponding driver at *PASSIVE_LEVEL*. If the flag is clear, the Power Manager sends those IRPs at *DISPATCH_LEVEL*. (*IRP_MN_WAIT_WAKE* and *IRP_MN_POWER_SEQUENCE* requests are always sent at *PASSIVE_LEVEL*.)

PoCallDriver acts as a sort of interrupt request level (IRQL) transformer for *SET_POWER* and *QUERY_POWER* requests. A driver might forward these IRPs at either *PASSIVE_LEVEL* or *DISPATCH_LEVEL*, depending on the IRQL at which the driver itself received the IRP and on whether it's forwarding the IRP as part of an I/O completion routine. If necessary, *PoCallDriver* will raise the IRQL or schedule a work item to call the next driver at the correct IRQL.

In Windows 2000, however, the Power Manager nevertheless objects (by bugchecking) if it finds a nonpaged device object layered on top of a paged device object when it's building internal lists in preparation for power operations. The Driver Verifier in all systems after and including Windows 2000 also checks for this condition. Because of the rule, you need to make sure at all times that your device object has *DO_POWER_PAGABLE* set if the driver under you does, and you need to help the driver above you obey the rule too. The first aspect of obeying the rule is to set the flag the same as the lower device object at *AddDevice* time.

We don't need to copy the *SectorSize* or *AlignmentRequirement* members of the lower device object—*IoAttachDeviceToDeviceStack* will do that automatically. We don't need to copy the *Characteristics* flags because the PnP Manager does that automatically after the device stack is completely built and after it applies overrides from the registry.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 443

There's ordinarily no need for a FiDO to have its own name. If the function driver names its device object and creates a symbolic link, or if the function driver registers a device interface for its device object, an application will be able to open a handle for the device. Every IRP sent to the device gets sent first to the topmost FiDO driver, regardless of whether that FiDO has its own name. Further on in this chapter, I'll discuss how to create an extra named device object to allow applications to access your filter in the middle of a driver stack.



The Driver Verifier ensures that you set the device object flags and type as described here.

16.2.3 The *DispatchAny* Function

You write a filter driver in the first place because you want to modify the behavior of a device in some way. Therefore, you'll have dispatch functions that do something with some of the IRPs that come your way. But you'll be passing most of the IRPs down the stack, and you pretty much know how to do this already:

```
NTSTATUS DispatchAny(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
  IoSkipCurrentIrpStackLocation(Irp);
  status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
  }
```

NOTE
You should follow this guideline when you program a filter driver: *First, do no harm.* In other words, don't cause drivers above or below you to fail because you perturbed anything at all in their environment or in the flow of IRPs.



We need to revisit the Chapter 6 discussion about the remove lock at this point. Recall that we acquire this lock for each IRP that we pass down the PnP stack and release it when that IRP finishes. Our *IRP_MN_REMOVE_DEVICE* dispatch routine will call *IoReleaseRemoveLockAndWait* to make sure that all such IRPs have drained through the lower driver before we call *IoDetachDevice* and before we return to the PnP Manager. These steps prevent the lower driver from being removed while it's processing an IRP that we've sent it.

DispatchAny uses the remove lock in an attempt to partially fulfill our responsibility to keep the lower driver in memory. As discussed in Chapter 6, however, there's a small hole in the protection we're trying to provide. Our protection of the lower driver will expire as soon as we release the lock at the end of *DispatchAny*. If the lower driver, or any driver underneath it, returns *STATUS_PENDING* from the dispatch routine, we're going to release the lock too soon. To provide totally bulletproof protection, we would need to install a completion routine (using *IoSetCompletionRoutineEx* if it's available) that would release the remove lock.

Installing a completion routine for every IRP in every filter driver isn't an acceptable solution to the early-unload problem, however, because doing so would greatly increase the cost of handling every IRP just to guard against a low-probability race condition. Furthermore, many thousands of filter drivers already in the field don't go to these heroic lengths. Consequently, Microsoft is going to have to find a more general solution to the problem. You could even make the case that a *DispatchAny* routine in a filter driver might as well not bother with the remove lock at all since it provides only limited protection at some slight cost and since most IRPs that flow through a filter driver are handle based in the first place. As discussed in Chapter 6, handle-based IRPs are inherently safe because Windows XP won't even send an *IRP_MN_REMOVE_DEVICE* to a device stack while anyone has an open handle.

16.2.4 The *DispatchPower* Routine

The dispatch routine for *IRP_MJ_POWER* in a filter driver is straightforward and (nearly) trivial:

```
NTSTATUS DispatchPower(PDEVICE OBJECT fido, PIRP Irp)
  {
  PDEVICE EXTENSION pdx = (PDEVICE EXTENSION) fido->DeviceObject;
  PoStartNextPowerIrp(Irp);
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT SUCCESS(status))
    return CompleteRequest(Irp, status, 0);
  IoSkipCurrentIrpStackLocation(Irp);
  status = PoCallDriver(pdx->LowerDeviceObject, Irp);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
```


IPR2024-00292
KASPERSKY EX 1008 Page 444

```
    return status;
    }
```

The only remarkable thing about this routine is that in contrast with every other *DispatchPower* routine you've ever seen, this one is actually simple to code.

16.2.5 The *DispatchPnp* Routine

The dispatch routine for *IRP_MJ_PNP* in a filter driver has several special cases:

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fido, PIRP Irp)
  {
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG fcn = stack->MinorFunction;

  NTSTATUS status;
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) fido->DeviceExtension;
  status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status, 0);

  if (fcn == IRP_MN_DEVICE_USAGE_NOTIFICATION)
    {
    if (!fido->AttachedDevice
      || (fido->AttachedDevice->Flags & DO_POWER_PAGABLE))
      fido->Flags |= DO_POWER_PAGABLE;
    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
      UsageNotificationCompletionRoutine,
      (PVOID) pdx, TRUE, TRUE, TRUE);
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
    }

  if (fcn == IRP_MN_START_DEVICE)
    {
    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
      StartDeviceCompletionRoutine,
      (PVOID) pdx, TRUE, TRUE, TRUE);
    return IoCallDriver(pdx->LowerDeviceObject, Irp);
    }

  if (fcn == IRP_MN_REMOVE_DEVICE)
    {
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(pdx->LowerDeviceObject, Irp);
    IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
    RemoveDevice(fido);
    return status;
    }

  IoSkipCurrentIrpStackLocation(Irp);
  status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
  }
```

IRP_MN_DEVICE_USAGE_NOTIFICATION

Recall from Chapter 6 that the usage notification informs a function driver that a disk device contains or doesn't contain a paging file, a dump file, or a hibernation file. In response to a usage notification, the function driver might change the setting of its *DO_POWER_PAGABLE* flag. We may need to alter our setting of that flag in sympathy.

As the IRP travels down the stack, we set *DO_POWER_PAGABLE* if we're at the top of the PnP stack or if the driver above us has set this flag. It's not normally safe to reference the *AttachedDevice* field because we don't have access to the internal spin lock that protects the device object stack. The reference is safe in this context, though, because the PnP Manager won't be changing the stack while the usage notification IRP is outstanding.

IPR2024-00292
KASPERSKY EX 1008 Page 445

NOTE
If another driver were to attach to our device stack while a usage notification was traveling down the stack, there is some slight possibility of allowing a nonpaged driver to be layered above a paged driver. Microsoft's own filter drivers don't worry about this possibility, so you and I can probably follow suit.

As the IRP travels back up the stack, our completion routine propagates the flag setting from the lower driver so as to obey the rule that we don't have a nonpaged handler layered on top of a paged handler.

```
NTSTATUS UsageNotificationCompletionRoutine(
  PDEVICE OBJECT fido, PIRP Irp, PDEVICE EXTENSION pdx)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
  if (!(pdx->LowerDeviceObject->Flags & DO POWER PAGABLE))
    fido->Flags &= ~DO POWER PAGABLE;
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return STATUS SUCCESS;
  }
```

For this code to work right, the driver at the top of the stack must set *DO_POWER_PAGABLE* on the way down so that we'll have set it ourselves in the dispatch routine. We'll leave it set if the lower driver sets it. We'll clear it if the lower driver clears it.

IRP_MN_START_DEVICE

We hook *IRP_MN_START_DEVICE* so we can propagate the *FILE_REMOVABLE_MEDIA* flag. This flag doesn't get set in *AddDevice* (because the function driver can't talk to the device then) but must be set correctly in the topmost device object of a storage stack. The completion routine is as follows:

```
NTSTATUS StartDeviceCompletionRoutine(PDEVICE OBJECT fdo,
  PIRP Irp, PDEVICE EXTENSION pdx)
  {
  if (Irp->PendingReturned)
    IoMarkIrpPending(Irp);
  if (pdx->LowerDeviceObject->Characteristics & FILE REMOVABLE MEDIA)
    fido->Characteristics |= FILE REMOVABLE MEDIA;
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return STATUS SUCCESS;
  }
```

IRP_MN_REMOVE_DEVICE

We handle *IRP_MN_REMOVE_DEVICE* specially because this is where we do the *RemoveDevice* processing that calls *IoDetachDevice* and *IoDeleteDevice*. The reason this is so much simpler than in a function driver is that we don't have queues to abort or I/O resources to release.

16.3 Installing a Filter Driver

There are only two truly easy ways to install a filter driver: when the filter is for an entire class of device, or when it's part of a package that includes the function driver. To install a filter driver for an existing device, you need to locate and modify the right registry entries more or less by hand. I'll discuss these installation issues in this section.

Figure 2-7 illustrates the layering of device and class filters. To recapitulate what Chapter 2 says about the order of driver loading, the PnP Manager first loads device and class lower filters, then the function driver, and finally device and class upper filters. The lowest driver in the stack (apart from the bus driver, that is) will be the first driver mentioned in the hardware key's *LowerFilters* value. The highest driver in the stack will be the last driver mentioned in the class key's *UpperFilters* value.

16.3.1 Installing a Class Filter

To install a class filter along with the class, add some syntax to the [ClassInstall32] section:

```
[ClassInstall32]
AddReg=ClassInstall32AddReg
CopyFiles=CopyClassFilters
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 446

```
[ClassInstall32AddReg]
HKR,,UpperFilters,0x00010000,foo,bar

[ClassInstall32.Services]
AddService=FOO,,FooAddService
AddService=BAR,,BarAddService

[CopyClassFilters]
foo.sys,,,2
bar.sys,,,2
```

The AddReg section defines a *REG_MULTI_SZ* value containing two strings: foo and bar, which specify the service names of two filter drivers. The [ClassInstall32.Services] section defines these services using the same service-definition syntax that I showed in Chapter 15 for a function driver. The [CopyClassFilters] section relies on a [DestinationDirs] directive (not shown) to put the driver files into the drivers directory.

To define class lower filters, simply define a value named *LowerFilters* using similar syntax.

Once the class has been defined, you can add filter drivers to the end of the list of filters in Windows 2000 and later systems by using an INF like this one. (Compare the CLASFILT sample in the DDK, which has a bit more syntax than you actually need.)

```
[Version]
Signature=$CHICAGO$

[DefaultInstall]
CopyFiles=CopyClassFilters
AddReg=FilterAddReg

[DefaultInstall.Services]
AddService=JUNKOLA,,FilterAddService

[SourceDisksFiles]
junkola.sys=1

[SourceDisksNames]
1="Filter install disk"

[DestinationDirs]
DefaultDestDir=12

[CopyClassFilters]
junkola.sys,,,2

[FilterAddService]
ServiceType=1
ErrorControl=1
StartType=3
ServiceBinary=%12%\junkola.sys

[FilterAddReg]
HKLM,%CLASSKEY%,UpperFilters,0x00010008,junkola
[Strings]
CLASSKEY=System\CurrentControlSet\Control\Class\{<class-GUID>}
```

The only interesting thing in this sample INF is the [FilterAddReg] section, which uses the *FLG_ADDREG_APPEND* flag to cause the new filter service to be appended to the existing *UpperFilters* value unless it's already in the list of upper filters. Note that you need to spell out the setup class globally unique identifier (GUID) for the class you're overriding in this way.

NOTE
The DDK documentation is a bit sparse when it comes to the *REG_MULTI_SZ* value syntax. If you want to completely overwrite a *REG_MULTI_SZ* value with one or more string values, specify the flag 0x00010000 and specify the strings with commas between them, as in the first example in this subsection. If you want to append a value, specify the flag 0x00010008, as in the second example. You can append just one value with one directive in the AddReg section. You can, however, have more than one directive that applies to the same value, and they'll be executed in the order they appear. The comparison to see whether a given string is already in the value is not case sensitive. There is no syntax for putting a new string elsewhere than at the end of the value.

To actually install the filter by hand, right-click on the INF file and select the INSTALL choice. Alternatively, use RUNDLL32 to execute the *InstallHinfSection* function in setupapi.dll with an argument string composed by concatenating "DefaultInstall "

IPR2024-00292
KASPERSKY EX 1008 Page 447

(notice the trailing space) with the name of the INF file.

Microsoft hasn't provided a way to extend the list of class filters in Windows 98/Me. You need to write an installation program to directly modify the class key.

16.3.2 Installing a Device Filter with the Function Driver

To install a device upper or lower filter along with the function driver in Windows 2000 and later systems, you need merely to define an *UpperFilters* or *LowerFilters* value in the hardware key, copy the filter driver files to the driver directory, and define the filter services. Here's an example of the additional syntax you need:

```
[DriverCopyFiles]
foo.sys,,,2
:

[DriverInstall.ntx86.Services]
AddService=whatever,2,DriverService  ; <== for the function driver
AddService=foo,,FooService
:

[FooService]
ServiceType=1
ErrorControl=1
StartType=3
ServiceBinary=%12%\foo.sys

[DriverInstall.ntx86.hw]
AddReg=FilterAddReg.ntx86

[FilterAddReg.ntx86]
HKR,,UpperFilters,0x00010000,foo
```

To specify a lower filter, define a *LowerFilters* value in the hardware key. To specify more than one filter, use commas to separate the service names in the directive or directives in the AddReg section.

16.3.3 Installing a Device Filter for an Existing Device

Microsoft hasn't provided a way to install a filter driver for an existing device. The problem you face when you try to do this is determining the name of the hardware key in the registry, and there's no general way to do that.

The FILTJECT.DLL code in the companion content illustrates a scheme for installing a device filter after the fact. You invoke FILTJECT by means of the system RUNDLL32 utility, which makes it usable in a RunOnce key installed by an INF file. (See the FILTER sample's INF file for an example.) FILTJECT parses command-line options that include the exact device description or friendly name string of the device you want to filter. Then it enumerates all installed devices to locate all instances of that device, and it modifies the *UpperFilters* or *LowerFilters* values as specified by other command-line options.

NOTE
The FILTER.HTM file describing the FILTER sample also describes the syntax for FILTJECT.DLL.

The POWTRACE sample shows another scheme for after-the-fact filtering. POWTRACE's INF file merely defines the POWTRACE service. You can then hand-edit the registry to add a *LowerFilters* value to any device you want to filter.

16.4 Case Studies

The preceding theory lays out a framework for writing any sort of WDM filter driver. To write a filter driver for any particular purpose, you will undoubtedly encounter many pitfalls based on peculiarities in device stacks and your position in the device stack. In this section, I'll present some additional examples of filter drivers to help you get started.

16.4.1 Lower Filter for Traffic Monitoring

I'll tell you a secret. I have a hard time getting power management to work right in my drivers. It's such a recurring theme that I finally wrote the POWTRACE filter driver. You install POWTRACE as a lower filter for any driver you're trying to debug. It then logs all the power-related IRPs that your function driver generates. There's actually nothing at all remarkable about how POWTRACE works on the inside, so I'll commend you to the companion content.

Let's say you want to investigate how Microsoft's HIDUSB and HIDCLASS drivers handle wake-up for a USB mouse. After making sure that the POWTRACE service entries are defined, you can modify the hardware key for the mouse in question

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 448

(Figure 16-7). Then unplug and replug the mouse, and start watching the debug trace with DbgView. Figure 16-8 is the trace I generated by enabling wake-up in the Device Manager (line 58), putting the machine in standby (lines 60 through 68), and then waking the system with a mouse click (lines 70 through 89).

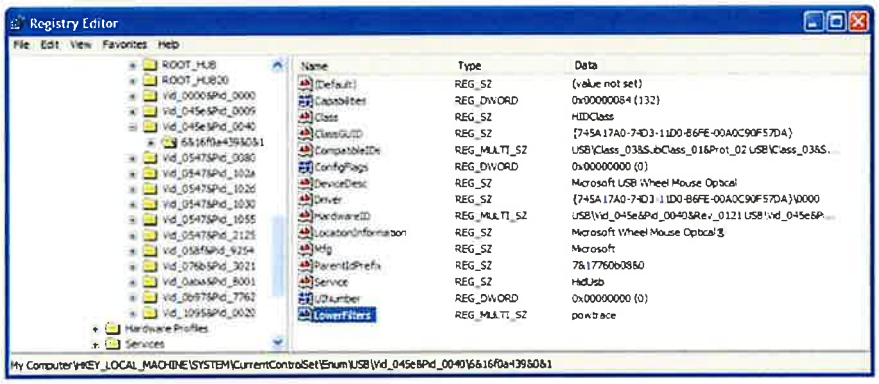


Figure 16-7. *Preparing to use POWTRACE.*

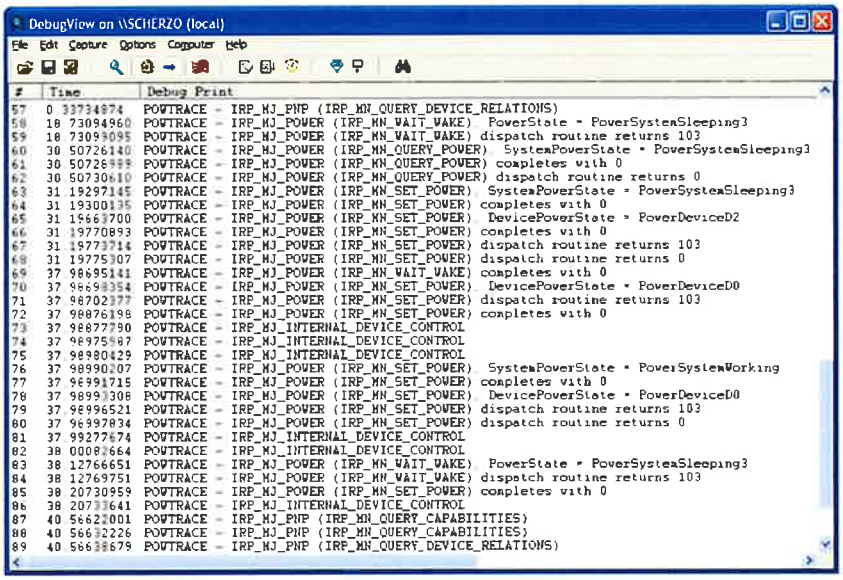


Figure 16-8. *POWTRACE log of wake-up/resume cycle.*

16.4.2 Named Filters

Sometimes you want to have an application talk directly to a filter driver. The straightforward way to arrange this sort of communication is for the application to open a handle to the device and then use some private IOCTL operations that the filter consumes. Unfortunately, this approach isn't always feasible:

- Some devices won't let your application open a handle. A mouse or a keyboard, for example, will already be open for the raw input thread, and you can't open a second handle. A serial port is usually an exclusive device and likewise won't let you open a second handle if someone else is already using the port.
- You're at the mercy of all the drivers above you as to whether you'll see any IOCTL traffic. MOUCLASS and KBDCLASS, for example, block private IOCTLs. Even if your application could open a handle, therefore, it still couldn't talk to a filter in the mouse or keyboard stack.

The standard solution to these kinds of problems is to create an Extra Device Object (EDO) that shadows the FiDO you put in the PnP stack. Figure 16-9 illustrates the concept.

IPR2024-00292
KASPERSKY EX 1008 Page 449

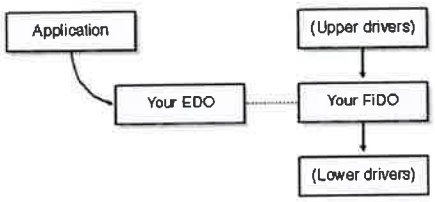


Figure 16-9. *An Extra Device Object for a filter driver.*

To bring this off, you need to define two different device extension structures that have at least one initial member in common. For example:

```
typedef struct  COMMON DEVICE EXTENSION {
  ULONG flag;
  } COMMON DEVICE EXTENSION, *PCOMMON DEVICE EXTENSION;

typedef struct  DEVICE EXTENSION : public  COMMON DEVICE EXTENSION {
:
  struct  EXTRA DEVICE EXTENSION* edx;
:
  } DEVICE EXTENSION, *PDEVICE EXTENSION;

typedef struct  EXTRA DEVICE EXTENSION : public
   COMMON DEVICE EXTENSION {
:
  PDEVICE EXTENSION pdx;
:
  } EXTRA DEVICE EXTENSION, *PEXTRA DEVICE EXTENSION;

#define FIDO EXTENSION 0
#define EDO_EXTENSION 1
```

In your *AddDevice* function, you create two device objects: a FiDO that you link into the PnP stack and an EDO that you don't. You give the EDO a unique name too. For example:

```
NTSTATUS AddDevice(PDRIVER OBJECT Driver Object, PDEVICE OBJECT pdo)
  {
  PDEVICE OBJECT fido;
  IoCreateDevice(...);
  PDEVICE EXTENSION pdx =
    (PDEVICE EXTENSION) fido->DeviceExtension;
  <etc. -- all the stuff shown earlier in the chapter>
  pdx->flag = FIDO EXTENSION;

  WCHAR namebuf[64];
  static LONG numextra = -1;
   snwprintf(namebuf, arraysize(namebuf), L"\\Device\\MyExtra%d",
    InterlockedIncrement(&numextra));
  UNICODE STRING edoname;
  RtlInitUnicodeString(&edoname, namebuf);
  IoCreateDevice(DriverObject, sizeof(EXTRA DEVICE EXTENSION),
    &edoname, FILE DEVICE UNKNOWN, 0, FALSE, &edo);

  PEXTRA DEVICE EXTENSION edx =
    (PEXTRA DEVICE EXTENSION) edo->DeviceExtension;
  edx->flag = EDO EXTENSION;
  edx->pdx = pdx;
  pdx->edx = edx;
:
  edo->flags &= ~DO DEVICE INITIALIZING;  //see Updates&errata
  }
```

You'll also want to create a uniquely named symbolic link to point to the EDO. That's the name your application will use when it wants to open a handle to your filter.

In each dispatch routine, you first cast the device object's *DeviceExtension* pointer to *PCOMMON_DEVICE_EXTENSION*,

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 450

and you inspect the flag member to see whether the IRP is aimed at the FiDO or the EDO. You handle FiDO IRPs as shown earlier for a generic filter driver. You handle EDO IRPs however you please. At a minimum, you'll succeed *IRP_MJ_CREATE* and *IRP_MJ_CLOSE* requests for the EDO, and you'll respond to whichever set of IOCTLs you define.

The foregoing are the basics for making your filter driver accessible no matter where it is in the driver stack and no matter which policies the drivers above you may be enforcing on file opens and private IOCTLs. There are a few other details to attend to if you want to put the idea to real use:

- Pay attention to the security attributes on the EDO. This would be a good time to use *IoCreateDeviceSecure* when it becomes available in the DDK. (This function, which was brand-new at press time, allows you to specify a security descriptor for a new device object. This is one of the situations it was invented to handle.)
- Use instructions you receive over the EDO pathway to modulate how you handle IRPs in the FiDO pathway.
- You need to delete the EDO at the same time you delete the FiDO. Don't forget to drain IRPs through the EDO too.
- The EDO is not part of the PnP stack. Consequently, it doesn't receive PnP, power, or WMI requests. Moreover, you can't use *IoRegisterDeviceInterface* to create a symbolic link to it—you have to use the older method of naming the EDO and creating a symbol link that points to it.

16.4.3 Bus Filters

A bus filter is a special kind of upper filter that attaches just above a bus driver. Recall that bus drivers wear two hats: a function device object (FDO) hat and a PDO hat. Creating an upper filter for the FDO personality of a bus driver is completely trivial: just make yourself a device *UpperFilter* in the hardware key for the bus driver. The tricky part about a bus filter is inserting yourself into each of the child device stacks right above the PDOs that the bus driver creates. Figure 16-10 illustrates the topology you're aiming for.

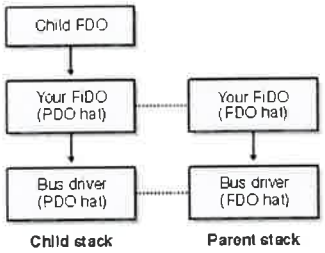


Figure 16-10. *Bus filter topology.*

Managing the FiDOs in each child device stack isn't too hard. In your parent stack filter role (FDO hat, in other words), be on the lookout for *IRP_MN_QUERY_DEVICE_RELATIONS* asking for *BusRelations*. This query is how the PnP Manager asks the bus driver for a list of all the child devices. Review Chapter 11 if you're a bit unclear on how this protocol works. You'll want to pass this query down synchronously (the *ForwardAndWait* scenario from the end of Chapter 5) and then inspect the list of PDOs that the bus driver returns. Each time you see a child PDO for the first time, you'll create a child-stack FiDO and attach it to the PDO. Each time you fail to see a PDO that you had seen earlier, you'll detach and delete the now-obsolete child-stack FiDO.

Creating and destroying child-stack FiDOs may not be all you need to do to create a working bus filter, though. The USB driver stack, for example, uses a back door to let USBHUB communicate efficiently with the host controller driver without sending IRPs through all the intermediate hub drivers that might be present. A USB bus filter uses *USBD_RegisterHcFilter* in the parent device stack for the host controller to make sure that it sees this backdoor traffic. Other buses might have similar registration requirements.

16.4.4 Keyboard and Mouse Filters

- A common use of filtering is for keyboard and mouse input. Among the applications I've seen for such filters are
- Computer-aided training, especially when it involves raw motion reports or keystrokes that can't be hooked in user mode.
- Accessibility applications.
- Automated testing.

Earlier in this chapter, I showed example driver stacks for keyboards and mice. I think it's easiest to plan on building a keyboard or mouse filter as a class upper filter that sits just below KBDCLASS or MOUCLASS, as the case may be. Your *DriverEntry* and *AddDevice* routines will be as shown earlier for a standard WDM filter driver. You'll probably want to create an EDO for out-of-band communication with your own user-mode application code too.

IPR2024-00292
KASPERSKY EX 1008 Page 451

NOTE
The DDK samples KBFILTR and MOUFILTR illustrate the basics of writing a keyboard or mouse filter.

KBDCLASS and MOUCLASS use a direct-call interface to receive keyboard and mouse reports from whichever port driver is actually handling the hardware. (These two drivers are samples in the DDK, so you can see for yourself exactly what's going on.) To effectively filter the reports, you need to hook into this direct-call mechanism by handling either *IOCTL_INTERNAL_KEYBOARD_CONNECT* or *IOCTL_INTERNAL_MOUSE_CONNECT*, which are internal device control requests. These IOCTLs use a parameter structure declared in KBDMOU.H:

```
typedef struct  CONNECT DATA {
  IN PDEVICE OBJECT ClassDeviceObject;
  IN PVOID ClassService;
  } CONNECT_DATA, *PCONNECT_DATA;
```

Here *ClassDeviceObject* is the address of a device object belonging to KBDCLASS or MOUCLASS, and *ClassService* is the address of a function with the following abstract prototype:

```
typedef VOID (*PSERVICE CALLBACK ROUTINE) (PVOID NormalContext,
  PVOID SystemArgument1, PVOID SystemArgument2,
  PVOID SystemArgument3);
```

Your filter driver's dispatch routine for *IRP_MJ_INTERNAL_DEVICE_CONTROL* would process the *CONNECT* request by saving the *ClassDeviceObject* and *ClassService* values in your own device extension and then substituting your own device object and callback routine addresses before passing the IRP down the stack to the port driver.

Once the direct-call connection is made, the port driver calls the *ClassService* callback routine each time an input event occurs. The callback routine is expected to consume a certain number of reports from any array provided by the port driver and return with an indication of how many reports were consumed.

In the case of a keyboard filter, the callback routine has this actual prototype:

```
VOID KeyboardCallback(PDEVICE OBJECT fido,
  PKEYBOARD INPUT DATA start,
  PKEYBOARD_INPUT_DATA end, PULONG consumed);
```

The *start* and *end* parameters delimit an array of *KEYBOARD_INPUT_DATA* structures, each of which relates to a single key press or release event (see NTDDKKB.H):

```
typedef struct  KEYBOARD INPUT DATA {
  USHORT UnitId;
  USHORT MakeCode;
  USHORT Flags;
  USHORT Reserved;
  ULONG ExtraInformation;
  } KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

The so-called *MakeCode* is a raw scan code generated by the keyboard. This scan code describes the physical position of the key on the keyboard and has no necessary relationship to the graphic on the keycap. The *Flags* member indicates whether the key has been pressed (*KEY_MAKE*) or released (*KEY_BREAK*). There are also flag bits (*KEY_E0* and *KEY_E1*) to indicate extended shift states that modulate certain special keys like SysRq and Pause.

In the case of a mouse filter, the callback has this actual prototype:

```
VOID MouseCallback(PDEVICE OBJECT fido, PMOUSE INPUT DATA start,
  PMOUSE_INPUT_DATA end, PULONG consumed);
```

In this function, start and end refer to instances of the following structure (see NTDDMOU.H):

```
typedef struct  MOUSE INPUT DATA {
  USHORT UnitId;
  USHORT Flags;
  union {
    ULONG Buttons;
    struct {
      USHORT ButtonFlags;
      USHORT ButtonData;
      };
```

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 452

```
    };
ULONG RawButtons;
LONG LastX;
LONG LastY;
ULONG ExtraInformation;
} MOUSE_INPUT_DATA, *PMOUSE_INPUT_DATA;
```

The mouse report information is in *LastX* and *LastY* (distances moved or absolute position) and *ButtonFlags* (bits such as *MOUSE_LEFT_BUTTON_DOWN* to indicate button events).

In either case, you return (in **consumed*) the number of events you've actually removed from the array bounded by *start* and *end*. The function driver will report any unconsumed events in a subsequent callback. When you're testing a filter, you can easily be misled into thinking that you never get more than one report at a time, but you really do have to program the driver to accept an array of reports.

Within your callback routine, you can do any of the following:

- Forward events by passing them to the higher-level driver's callback routine. Don't forget to take into account that the higher-level driver might not consume all the events that you try to forward.
- Remove events by not passing them up. You can, for example, make several calls to the higher-level driver's callback routines for portions of the array not including the events you want to elide.

You can also insert events into the stream by making your own call to the higher-level driver's callback routine.

16.4.5 Filtering Other HID Devices

It's very difficult to provide a useful filter driver for a HID device. To understand why, we need to look closely at the device stack for a HID device such as a joystick. (See Figure 16-11.) The function driver for the physical device is HIDUSB, a HIDCLASS minidriver. It does you no particular good to attach an upper filter to this device (which you could easily do using techniques already discussed in this chapter) because HIDCLASS will fail any *IRP_MJ_CREATE* directed to the FDO. There is, in fact, no simple way for a filter driver in the parent driver stack to receive *any* IRPs except by creating an Extra Device Object.

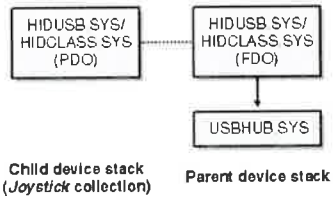


Figure 16-11. *Driver stack for a joystick.*

Often, you're not interested in filtering the real device anyway. Instead, you want to filter the reports flowing upward in the child driver stack that HIDCLASS creates for each top-level collection exported by the real device. The problem is, you can't know in advance which device identifier HIDCLASS will use for the collection PDOs, and you can't modify the Microsoft-provided INF files for the collection device class without breaking Microsoft's digital signature for its own drivers. If you can get by with a class filter, as you can for keyboards and mice, well and good—just install your filter as a class filter. But if you need a device-specific filter, it looks as though you're out of luck.

This is where the concept of a bus filter might be useful. Go ahead and install your filter as an upper filter in the parent stack. Then have it monitor the *BusRelations* queries as described earlier for bus filter drivers. At the appropriate time, you can instantiate a FiDO in the collection driver stack. If you can get by with having your filter be the bottommost in the collection stack, you're done. Otherwise, you may need to do something relatively heroic such as alter the device identifiers reported by *IRP_MN_QUERY_ID* in order to force use of your own INF file.

16.5 Windows 98/Me Compatibility Notes

There are some minor differences between Windows 98/Me and Windows XP insofar as the material discussed in this chapter goes.

16.5.1 WDM Filters for VxD Drivers

People often want to take WDM filter drivers for serial or disk devices and just port them to Windows 98/Me. This process won't work. Windows 98/Me uses VxD drivers for serial ports and block storage devices, and there's no simple way to insert a WDM driver into the flow of I/O requests.

IPR2024-00292
KASPERSKY EX 1008 Page 453

16.5.2 INF Shortcut

Rather than trying to figure out how to store a *REG_MULTI_SZ* value in Windows 98/Me (which doesn't support this registry data type in the first place), I find it easier to just specify multiple drivers in the driver key:

```
[DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,, \
  "wdmstub.sys,powtrace.sys,whatever.sys,filter.sys"
```

The reason this works is that Windows 98/Me device loaders, including NTKERN, invariably call a CONFIGMG routine that loads all the drivers specified in a comma-delimited string.

16.5.3 Class Filter Drivers

Since Windows 98/Me doesn't support the *REG_MULTI_SZ* value type, you use a binary value when you want to specify class filters. (You can use the shortcut just described for device filters.) For example:

```
[ClassInstall]
AddReg=ClassInstallAddReg
CopyFiles=CopyClassFilters

[ClassInstallAddReg]
HKR,,UpperFilters,1, 66, 6f, 6f, 2e, 73, 79, 73, 00, \
62, 61, 72, 2e, 73, 79, 73, 00, 00
```

This syntax defines a *REG_BINARYUpperFilters* value containing foo.sys\0bar.sys\0\0. Note that you need to specify the filename rather than a service name.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 454

Appendix A
Coping with Cross-Platform Incompatibilities

I closed many of the chapters in this book with a series of MicrosoftWindows 98/Me compatibility notes. Microsoft originally planned that you'd be able to ship a single driver binary file for all WDM platforms, including Windows 98, Windows 98 Second Edition, Windows Me, Windows 2000, Windows XP, and later systems, but the sad fact is that so lofty a goal has proven elusive in practice. Not surprisingly, systems continued to evolve long after Windows 98 was up and running on millions of PCs, and Microsoft has added numerous kernel-mode service functions that the earlier systems don't support. If a WDM driver calls one of these functions, the system simply won't load the driver because it can't resolve the reference to the symbol. In this appendix, I'll discuss methods of coping with the incompatibilities so as to use a single binary anyway.

Determining the Operating System Version

Windows 2000 and later systems provide a routine, *PsGetVersion*, which seems ideally suited for determining which platform you're running on. Unfortunately, Windows 98/Me doesn't support this function. All platforms do, however, have in common this function:

BOOLEAN IoIsWdmVersionAvailable(major, minor);

This function returns *TRUE* if the platform supports the WDM driver interfaces at the specified level. The WHICHOS sample in the companion content illustrates one way you might use *IoIsWdmVersionAvailable* to distinguish among platforms, based on the information in Table A-1.

Platform	WDM Version Supported
Windows 98 "Gold" and Second Edition	1.0
Windows Me	1.05
Windows 2000	1.10
Windows XP	1.20
Windows .NET	1.30

Table A-1. *WDM Versions by Platform*

For example, you can determine whether you're running under Windows XP by making this function call:

```
BOOLEAN isXP = IoIsWdmVersionAvailable(1, 0x20);
```

Windows 98 was issued in two versions: the original, or "gold," version, and the Second Edition. *IoIsWdmVersionAvailable* will indicate version 1.0 on both systems. If you need to distinguish between the two, you can use the trick illustrated in WHICHOS of checking the *ServiceKeyName* member of the *DriverExtension* structure:

```
BOOLEAN Win98SE =
  DriverObject->DriverExtension->ServiceKeyName.Length != 0;
```

I've used this trick only in my *DriverEntry* function, by the way. For all I know, the *ServiceKeyName* member might change afterward.

The biggest distinction between platforms is, of course, between the Windows 2000 line of systems and Windows 98/Me. All of the sample drivers in the companion content define a global variable named *win98*, which *DriverEntry* initializes based on the result of calling *IoIsWdmVersionAvailable*:

```
win98 = !IoIsWdmVersionAvailable(1, 0x10);
```

Run-Time Dynamic Linking

As you know, Windows systems rely extensively on dynamic linking to connect applications and drivers to system libraries. In

IPR2024-00292
KASPERSKY EX 1008 Page 455

both kernel mode and user mode, if an executable module references a library or an entry point that doesn't exist, the system will refuse to load the module. If you've done extensive application programming for Windows platforms, you may be familiar with the Win32 API function *GetProcAddress*, which allows you to obtain a pointer to an exported function at run time. Programmers commonly use *GetProcAddress* to resolve symbol imports dynamically in a way that doesn't preclude the application from being loaded.

In Windows 2000 and later systems, you can call *MmGetSystemRoutineAddress* to locate an entry point in the kernel or in the hardware abstraction layer (HAL). In effect, this function is a kernel-mode version of *GetProcAddress*. For example (see the SPINLOCK sample from Chapter 4):

```
typedef VOID (FASTCALL *KEACQUIREINSTACKQUEUEDSPINLOCK)
  (PKSPIN LOCK, PKLOCK QUEUE HANDLE);
KEACQUIREINSTACKQUEUEDSPINLOCK pKeAcquireInStackQueuedSpinLock;
UNICODE STRING us;
RtlInitUnicodeString(&us, L"KeAcquireInStackQueuedSpinLock");
pKeAcquireInStackQueuedSpinLock = (KEACQUIREINSTACKQUEUEDSPINLOCK)
  MmGetSystemRoutineAddress(&us);
```

To use *MmGetSystemRoutineAddress* effectively, note the following fine points:

- The function only searches NTOSKRNL.EXE (the kernel) and HAL.DLL (the hardware abstraction layer) for the symbol you specify. You can't use this function to dynamically resolve entry points into other drivers or system components.
- Windows 98/Me doesn't support the function (but read on to learn about WDMSTUB, which *does* support it).
- You need to use the same name that the kernel uses for a given symbol. You may need to consult WDM.H or NTDDK.H to follow a chain of macro definitions.

Checking Platform Compatibility

The WDMCHECK utility in the companion content evaluates a WDM driver for import compatibility with a Windows 98 or Windows Me platform. To use the tool, follow these steps:

1. Develop your driver under Windows XP.
2. Copy the driver binary to the %windir%\system32\drivers directory on a Windows 98/Me system.
3. Make the drivers directory current.
4. Invoke WDMCHECK with the name of your driver as the only command-line argument.

WDMCHECK gives you a report about any symbols you're importing that aren't exported by the operating system or any installed WDM library. Figure A-1 illustrates the result of running WDMCHECK on a driver that has no unresolved imports.



Figure A-1. *Clean bill of health from WDMCHECK.*

Contrast Figure A-1 with Figure A-2, which shows the results for a driver that calls three functions that Windows Me doesn't natively support.



Figure A-2. *WDMCHECK results for a driver with missing imports.*

WDMCHECK scans the import sections in your driver and attempts to resolve each imported symbol. For symbols imported from NTOSKRNL.EXE, HAL.DLL, NDIS.SYS, and SCSI.SYS, the utility calls a helper VxD. The VxD in turn uses the *_PELDR_GetProcAddress* service to resolve the symbol. (That's the same service that the system loader will be using when it tries to load your driver, by the way.) For symbols imported from other modules, WDMCHECK opens the target module and scans its table of exported symbols. This algorithm essentially duplicates how the system would resolve references from your driver and therefore constitutes a fair test.

IPR2024-00292
KASPERSKY EX 1008 Page 456

If you want to check a driver for compatibility with Windows 2000 or later, copy it by hand to the drivers directory on the target system and run the DEPENDS utility from the Platform SDK. DEPENDS will flag any undefined imports. Figure A-3 is an example of using DEPENDS on Windows 2000 with a module that uses two functions added to Windows XP.

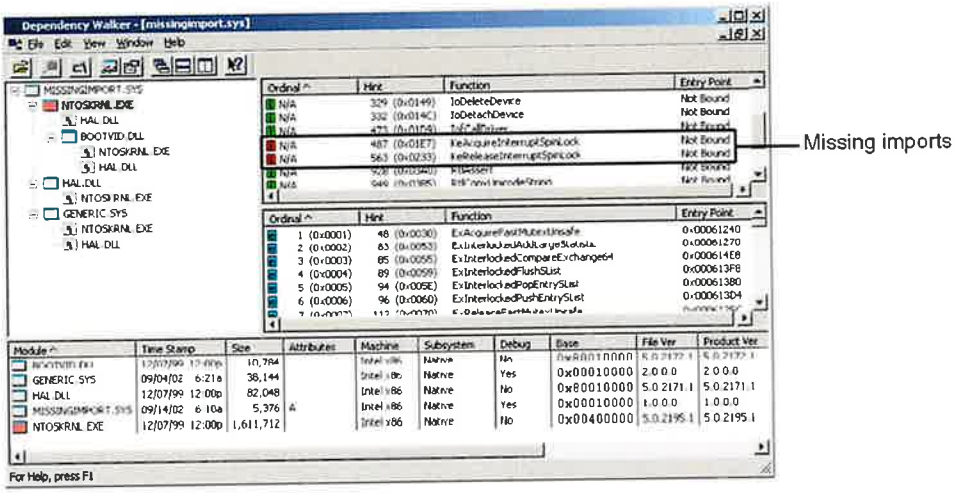


Figure A-3. *DEPENDS shows missing imports.*

Defining Win98/Me Stubs for Kernel-Mode Routines

The WDMSTUB sample in the companion content is a lower-filter driver that defines a number of kernel routines that Windows 98/Me omits. It relies on the same basic trick that Microsoft crafted to port several hundred kernel-mode support functions from Microsoft Windows NT to Windows 98/Me—that is, extending the symbol tables that the run-time loader uses when it resolves import references. To extend the symbol tables, you first define three data tables that will persist in memory:

- A name table that gives the names of the functions you're defining
- An address table that gives the addresses of the functions
- An ordinal table that correlates the name and address tables

Here are some of the table entries from WDMSTUB:

```
static char* names[] = {
  "PoRegisterSystemState",

  "ExSystemTimeToLocalTime",

  };

static WORD ordinals[] = {
  0,

  6,

  };

static PFN addresses[] = {
  (PFN) PoRegisterSystemState,

  (PFN) ExSystemTimeToLocalTime,

  };
```

The purpose of the ordinal table is to provide the index within *addresses* of the entry for a given *names* entry. That is, the function named by *names[i]* is *address[ordinals[i]]*.

If it weren't for a version compatibility problem I'll describe in a moment, you could call *_PELDR_AddExportTable* as follows:

IPR2024-00292
KASPERSKY EX 1008 Page 457

```
HPEEXPORTTABLE hExportTable = 0;

extern "C" BOOL OnDeviceInit(DWORD dwRefData)
  {
   PELDR AddExportTable(&hExportTable,
    "ntoskrnl.exe",
    arraysize(addresses), // <== don't do it this way!
    arraysize(names), 0,
    (PVOID*) names,
    ordinals, addresses, NULL);
  return TRUE;
  }
```

The call to *_PELDR_AddExportTable* extends the table of symbols that the loader uses when it tries to resolve import references from NTOSKRNL.EXE, which is of course the Windows XP kernel. NTKERN.VXD, the main support module for WDM drivers in Windows 98/Me, initializes this table with the addresses of the several hundred functions it supports. In effect, then, WDMSTUB is an extension of NTKERN.

Version Compatibility

The version compatibility problem to which I just alluded is this: Windows 98 supported a particular subset of the Windows 2000 functions used by WDM drivers. Windows 98 Second Edition supported a larger subset. The last version of Windows, Windows Me, supports a still larger subset. You wouldn't want your stub driver to duplicate one of the functions that the operating system supports. What WDMSTUB actually does during initialization, therefore, is dynamically construct the tables that it passes to *_PELDR_AddExportTable*:

```
HPEEXPORTTABLE hExportTable = 0;

extern "C" BOOL OnDeviceInit(DWORD dwRefData)
  {
  char** stubnames = (char**) HeapAllocate(sizeof(names), HEAPZEROINIT);
  PFN* stubaddresses = (PFN*) HeapAllocate(sizeof(addresses), HEAPZEROINIT);
  WORD* ordinals = (WORD*) HeapAllocate(arraysize(names) *
    sizeof(WORD), HEAPZEROINIT);
  int i, istub;
  for (i = 0, istub = 0; i < arraysize(names); ++i)
    {
    if ( PELDR GetProcAddress((HPEMODULE) "ntoskrnl.exe",
      names[i], NULL) == 0)
      {
      stubnames[istub] = names[i];
      ordinals[istub] = istub;
      stubaddresses[istub] = addresses[i];
      ++istub;
      }
    }
   PELDR AddExportTable(&hExportTable, "ntoskrnl.exe", istub,
    istub, 0, (PVOID*) stubnames, ordinals, stubaddresses, NULL);
  return TRUE;
  }
```

The line appearing in boldface is the crucial step here—it makes sure that we don't inadvertently replace a function that already exists in NTKERN or another system VxD.

There's one annoying glitch in the version compatibility solution I just outlined. Windows 98 Second Edition and Windows Me export just three of the four support functions for managing the IO_REMOVE_LOCK object. The missing function is *IoRemoveLockAndWaitEx*, if you care. My WDMSTUB.SYS driver compensates for this omission by stubbing either all or none of the remove lock functions based on whether or not this function is missing.

Stub Functions

The main purpose of WDMSTUB.SYS is to resolve symbols that your driver might reference but not actually call. For some functions, such as *PoRegisterSystemState*, WDMSTUB.SYS simply contains a stub that will return an error indication if it is ever called:

```
PVOID PoRegisterSystemState(PVOID hstate, ULONG flags)
  {
```


IPR2024-00292
KASPERSKY EX 1008 Page 458

```
    ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
    return NULL;
    }
```

Sometimes, though, you don't need to write a stub that fails the function call—you can actually implement the function, as in this example:

```
VOID ExLocalTimeToSystemTime(PLARGE_INTEGER localtime, PLARGE_INTEGER systime)
    {
    systime->QuadPart = localtime->QuadPart + GetZoneBias();
    }
```

where *GetZoneBias* is a helper routine that determines the time zone *bias*—that is, the number of units by which local time differs from Greenwich mean time—by interrogating the *ActiveTimeBias* value in the *TimeZoneInformation* registry key.

Support Function	Remarks
ExFreePoolWithTag	Stubbed
ExLocalTimeToSystemTime	Implemented
ExSystemTimeToLocalTime	Implemented
HalTranslateBusAddress	Subset implemented
IoAcquireRemoveLockEx	Implemented
IoAllocateWorkItem	Implemented
IoCreateNotificationEvent	Stub—always fails
IoCreateSynchronizationEvent	Stub—always fails
IoFreeWorkItem	Implemented
IoInitializeRemoveLockEx	Implemented
IoQueueWorkItem	Implemented
IoRaiseInformationalHardError	Implemented
IoReleaseRemoveLockEx	Implemented
IoReleaseRemoveLockAndWaitEx	Implemented
IoReuseIrp	Implemented
IoReportTargetDeviceChangeAsynchronous	Stub—always fails
IoSetCompletionRoutineEx	Implemented
KdDebuggerEnabled	Implemented
KeEnterCriticalRegion	Implemented
KeLeaveCriticalRegion	Implemented
KeNumberProcessors	Always returns 1
KeSetTargetProcessorDpc	Implemented
PoCancelDeviceNotify	Stub—always fails
PoRegisterDeviceNotify	Stub—always fails
PoRegisterSystemState	Stub—always fails
PoSetSystemState	Stub—always fails
PoUnregisterSystemState	Stub—always fails
PsGetVersion	Implemented
RtlInt64ToUnicodeString	Stub—always fails
RtlUlongByteSwap	Implemented
RtlUlonglongByteSwap	Implemented
RtlUshortByteSwap	Implemented
SeSinglePrivilegeCheck	Always returns TRUE
ExIsProcessorFeaturePresent	Implemented
MmGetSystemRoutineAddress	Implemented
ZwLoadDriver	Implemented
ZwQueryDefaultLocale	Implemented
ZwQueryInformationFile	Implemented
ZwSetInformationFile	Implemented
ZwUnloadDriver	Implemented

Table A-2. *Functions Stubbed in WDMSTUB.SYS*

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 459

Table A-2 lists the kernel-mode support functions that WDMSTUB.SYS exports.

Using WDMSTUB

To use WDMSTUB, include WDMSTUB.SYS in your driver package and specify it as a lower filter for your function driver. Here's an example from the INF file for one of the sample drivers in the companion content:

```
[DriverInstall]
AddReg=DriverAddReg
CopyFiles=DriverCopyFiles,StubCopyFiles

[StubCopyFiles]
wdmstub.sys,,,2

[DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,"wdmstub.sys,workitem.sys"
```

The indicated syntax for the NTMPDriver value in the driver key causes the system to load WDMSTUB.SYS before attempting to load the function driver WORKITEM.SYS. By the time the system gets around to loading WORKITEM.SYS, the *DriverEntry* routine in WDMSTUB has already executed and defined the functions listed in Table A-2.

Note that using WDMSTUB as a lower-filter driver means that the install package won't require a reboot.

Interaction Between WDMSTUB and WDMCHECK

WDMCHECK will tell you when certain symbols are defined only because you happen to have WDMSTUB loaded. See Figure A-4 for an example.



Figure A-4. *WDMCHECK results involving WDMSTUB.*

Special Licensing Note

WDMSTUB.SYS is an exception to the licensing requirements for the samples in the companion content. To avoid problems on end user machines caused by inconsistent versions of WDMSTUB, I ask that you not redistribute WDMSTUB.SYS except under license from me. I will grant a royalty-free license to distribute WDMSTUB to anyone who asks. Just send an e-mail to waltoney@oneysoft.com, explain that you want to redistribute WDMSTUB, and provide contact details so I can fax a license agreement to you.

IPR2024-00292
KASPERSKY EX 1008 Page 460

Appendix B

Using WDMWIZ.AWX

This appendix describes how to use the WDMWIZ.AWX custom wizard to build a driver project for use with MicrosoftVisual C++ version 6.0 and the Windows XP or .NET DDK. I built this wizard because I wanted an easy and reproducible way to generate the sample drivers for this book. I've included it in the companion content because I knew you'd want an easy way to generate drivers as you read through the book.

The WDMBOOK.HTM file in the companion content tells you how to install this wizard on your system. Once you've installed the wizard, you'll find a WDM Driver Wizard item on the Projects tab of the New dialog box that Visual C++ presents when you create a new project. WDMBOOK.HTM also contains detailed instructions for setting up your build environment to use the wizard. I'm not providing those instructions here in the book because they will undoubtedly change from time to time as Microsoft releases new Driver Development Kits.

WDMWIZ.AWX is not a product and never will be. I would like to know about situations in which it generates incorrect code, but I'm not planning to make any changes to the admittedly clunky user interface. Furthermore, you're on your own as far as quality assurance for your finished driver goes.

Basic Driver Information

The initial page (shown in Figure B-1) asks you for basic information about the driver you want to build.

For Type Of Driver, you can specify these choices:

- **Generic Function Driver**
  Builds a function driver for a generic device. (Note that the use of the word *generic* here is unfortunate because it has nothing to do with GENERIC.SYS.)
- **Generic Filter Driver**
  Builds a filter driver with default handling for all types of I/O request packet (IRP).
- **USB Function Driver**
  Builds a function driver for a universal serial bus (USB) device.
- **Empty Driver Project**
  Builds a project with no files but with options set up for building a WDM driver.
- **Convert existing SOURCES file**
  Creates a project according to the definition in a standard DDK SOURCES file. Use this option, for example, to convert a DDK sample to a Visual Studio project.



Figure B-1. *Page for entering basic driver information.*

You can select the following options:

- **Verbose Debugging Trace**
  If you check this option, the driver project files will include many *KdPrint* macro calls to trace important operations in

IPR2024-00292
KASPERSKY EX 1008 Page 461

the driver.

- **Use Buffered Method For Reads And Writes**
  Set this option if you want to use the *DO_BUFFERED_IO* method for read and write operations. Clear this option if you want to use *DO_DIRECT_IO* instead.

- **Use Old-Style For Device Naming**
  Set this option to generate named device objects. Clear this option to generate a driver that uses a device interface instead. The second choice (device interface) is the one Microsoft prefers for WDM drivers.

- **Replace ASSERT For i86 Platforms**
  The DDK's *ASSERT* macro calls a kernel-mode support routine (*RtlAssert*) that's a no-operation in the free build of Microsoft Windows 2000. The checked build of your driver will therefore not stop in the free build of the operating system. Set this option to redefine *ASSERT* so that the checked build of your driver halts even in the free build of the operating system.

- **Use GENERIC.SYS Library**
  Set this option to make use of the standardized driver code in GENERIC.SYS. Clear this option to put all that standardized code in your own driver.

- **Windows 98 Detection**
  Set this option to include a run-time check for whether your driver is running under Windows 98/Me or Windows 2000/XP. Clear this option to omit the check.

You can also specify the base pathname where you've installed the Windows .NET DDK and the samples for this book. The default values—$(DDKPATH) and $(WDMBOOK)—rely on the environment variables that the sample setup program creates.

Finally you can click the Dispatch Functions button to specify the types of IRP your driver will handle, as Figure B-2 shows. The dialog box embodies some design decisions that you can't override. Your driver will include support for *IRP_MJ_PNP* and *IRP_MJ_POWER*. If you specify handling for *IRP_MJ_CREATE*, you'll get support for *IRP_MJ_CLOSE*. If you specify handling for *IRP_MJ_READ*, *IRP_MJ_WRITE*, or *IRP_MJ_DEVICE_CONTROL*, you'll get support for *IRP_MJ_CREATE* (and therefore *IRP_MJ_CLOSE*). WDMWIZ.AWX doesn't generate skeleton dispatch functions for many types of IRP that are used only by file system drivers.

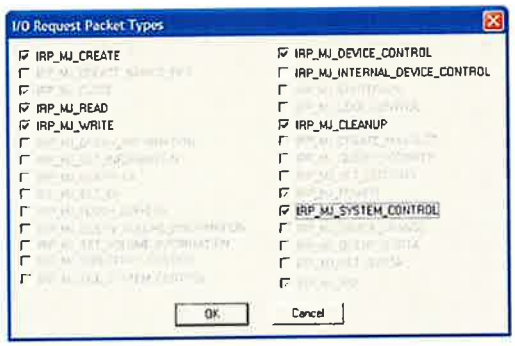


Figure B-2. *Dialog box for specifying the IRP major function codes for which you want dispatch functions.*

DeviceIoControl Codes

If you specified handling for *IRP_MJ_DEVICE_CONTROL*, the wizard will present a page (depicted in Figure B-3) to allow you to specify information about the control operations you support.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 462

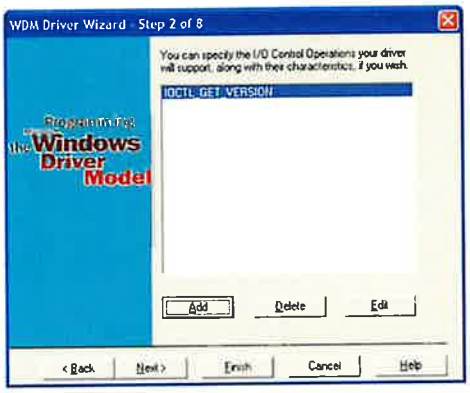


Figure B-3. *Page for specifying supported I/O control operations.*

Figure B-4 is an example of how you specify information about a particular *DeviceIoControl* operation. Most of the fields correspond directly to parameters in the *CTL_CODE* preprocessor macro and should therefore require no explanation. Setting the Asynchronous option generates support for an operation that you complete asynchronously after the dispatch function returns *STATUS_PENDING*.

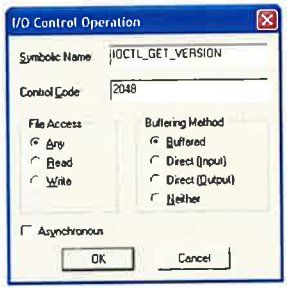


Figure B-4. *Dialog box for adding and editing an I/O control operation.*

I/O Resources

If your device uses any I/O resources, you can fill in the third page with information about them, as Figure B-5 shows.

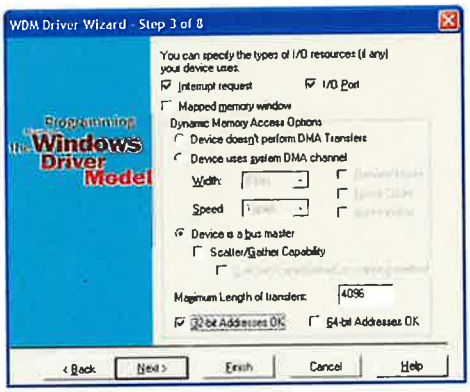


Figure B-5. *Page for specifying I/O resources.*

USB Endpoints

If you selected USB Function Driver on the first page, the wizard will present a page that allows you to describe the endpoints of your device, as Figure B-6 shows. This page lists the names of variables in your device extension that will hold pipe handles. The order of names corresponds to the order of endpoint descriptors on your device.

IPR2024-00292
KASPERSKY EX 1008 Page 463

NOTE
This page isn't sufficiently complex to let you describe a device with multiple interfaces or with alternate settings for interfaces.

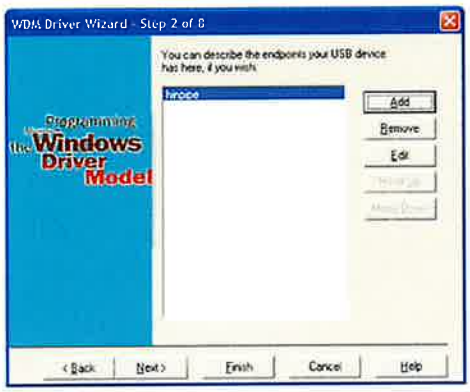


Figure B-6. *Page for defining USB endpoints.*

Refer to Figure B-7 for an illustration of the dialog box you can use to describe a single endpoint. The Description Of Endpoint group relates to the description of the endpoint in your device firmware and should be self-explanatory. Within the Resources In The Driver group, complete the fields as follows:

- **Name Of Pipe Handle In Device Extension**
  Supply the name of a *DEVICE_EXTENSION* member to hold the pipe handle you'll use for operations on this endpoint.
- **Maximum Transfer Per URB**
  Specify here the maximum number of bytes you'll transfer in a single URB. In general, this value is much larger than the endpoint maximum.

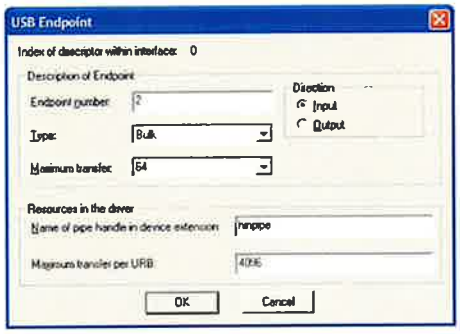


Figure B-7. *Dialog box for adding and editing a USB endpoint.*

WMI Support

If you've specified that you want to handle *IRP_MJ_SYSTEM_CONTROL* requests, the wizard will present the page shown in Figure B-8 to allow you to specify the elements of your custom Windows Management Instrumentation (WMI) schema or to specify any Microsoft-standard classes you will support.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 464

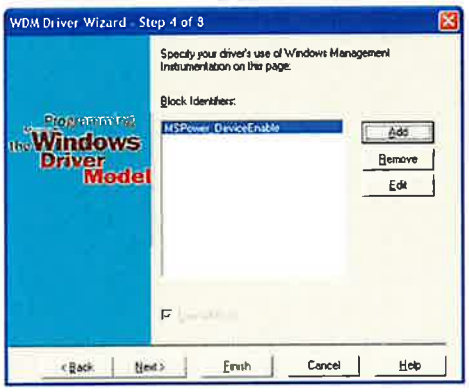


Figure B-8. *Page for specifying WMI options.*

The Block Identifiers list names the class globally unique identifiers (GUIDs) in the order they'll appear in the GUID list for WMILIB.

Figure B-9 illustrates how you can describe one of the standard Microsoft classes. The topmost (unlabeled) control is the symbolic name of the GUID. By typing in a name, you can specify a class in your custom schema. You can specify the following attributes of a WMI class:

- **Number Of Instances**
  Indicates how many instances of the class your driver will create.
- **Expensive**
  Indicates an expensive class that must be specifically enabled.
- **Event Only**
  Indicates that the class is used only to fire an event.
- **Traced**
  Corresponds to a WMI option that I don't currently understand. But if I ever do understand it, I'll be able to use this check box to influence its state.

You can choose between physical device object (PDO)-based instance naming and instance naming using a base name. Microsoft recommends you use PDO-based naming.

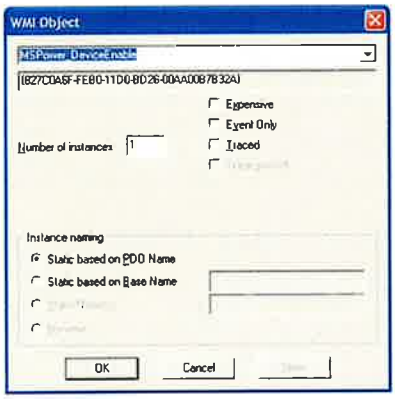


Figure B-9. *Dialog box for specifying a WMI class.*

Parameters for the INF File

The last page in the wizard (shown in Figure B-10) lets you specify information for the INF file that becomes part of your driver project.

The fields in this page are as follows:

- **Manufacturer Name**
  Name of the hardware manufacturer.

IPR2024-00292
KASPERSKY EX 1008 Page 465

- **Device Class**
  The standard device class to which your device belongs. Sample is my own class for the driver samples in this book; you shouldn't use this class for a production device.

- **Hardware ID**
  The hardware identifier for this device. I made up *WCO0B01 for this example. You should specify the identifier that will match one of the identifiers that the relevant bus driver will create. Refer to the section titled "Device Identifiers" in Chapter 15 for more information.

- **Friendly Name For Device**
  If you want to have a *FriendlyName* value inserted into the device's hardware key, specify that name here.

- Auto-Launch Command
  If you want the AutoLaunch service to automatically start an application when your device starts, specify the command line here. For example, when I built the AutoLaunch sample for Chapter 15, I specified %windir%\altest.exe %s %s in this field.

- **Device Description**
  Insert the description of your device here.

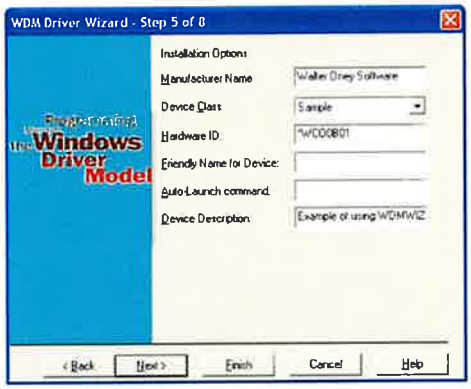


Figure B-10. *Page for specifying INF file options.*

Now What?

After you run through all the pages of the wizard, you'll have a project that you can use to finish crafting your driver. Because of limitations on the custom wizard support in Visual C++, you'll need to modify the project settings by hand. Please refer to WDMBOOK.HTM in the companion content for a description of these settings.

The generated code will contain a number of TODO comments that highlight areas where you need to write some code. I suggest you use the Find In Files command to locate these items.

WDMWIZ also generates a standard DDK SOURCES file that you can use with the BUILD utility. Many people prefer to use BUILD for driver builds, and this feature will make your life easier if you're one of them.

 Copyright © 2003 by Walter Oney

IPR2024-00292
KASPERSKY EX 1008 Page 466

Copyrighted Material

PROGRAMMING THE MICROSOFT WINDOWS DRIVER MODEL 2

Second Edition



Covers Windows 98, Windows Me, Windows 2000, and Windows XP

Learn to write Windows drivers the easy way—with expert help from the Windows driver authorities.

The Microsoft Windows driver model (WDM) supports Plug and Play, provides power management capabilities, and expands on the driver/minidriver approach. Written by long time device driver expert Walter Oney in cooperation with the Windows kernel team, this book provides extensive practical examples, illustrations, advice, and line-by-line analysis of code samples to clarify real world driver programming issues. It's also been updated with the latest details about the driver technologies in Windows XP and Windows 2000, plus more information about how to debug drivers.

Topics covered include:

- Beginning a driver project, and the structure of a WDM driver. **NEW:** Minidrivers and class drivers, driver taxonomy, the WDM development environment and tools, management checklist, driver selection and loading, approved API calls, and driver stacks
- Basic programming techniques. **NEW:** Safe string functions, memory tools, the Driver Verifier scheme and flags, the kernel handle flag, and the Windows 98 floating point problem
- Synchronization. **NEW:** Details about the interrupt request level (IRQL) scheme, along with Windows 98 and Windows Me compatibility
- The I/O request packet (IRP), and I/O control operations. **NEW:** How to send control operations to other drivers, custom queue implementations, and how to handle and safely cancel IRPs
- Plug and Play for function drivers. **NEW:** Monitoring device removal in user mode, and controller and multifunction devices. Human Interface Devices (HID), joysticks and other game controllers, minidrivers for non-HID devices, and feature reports
- Reading and writing data, power management, and Windows Management Instrumentation (WMI). **NEW:** System wakeup, the WMI control for idle detection, and using WMIMOFCK
- Specialized topics and distributing drivers. **NEW:** USB 2.0, selective suspend, Windows Hardware Quality Lab (WHQL) certification, driver selection and loading, approved API calls, and driver stacks

CD-ROM features:

- A fully searchable electronic copy of the book
- Sample code in Microsoft Visual C++®

*For **System Requirements**, please see the book's Introduction.*

About the Author:

Walter Oney has 35 years of experience in systems-level programming and has been teaching Windows device driver classes for 10 years. A Microsoft MVP based in Boston, MA, he has authored numerous magazine articles and was a contributing editor of *Microsoft Systems Journal* during its heyday. He has written several books, including *Systems Programming for Windows 95* and the first edition of *Programming the Microsoft Windows Driver Model*.

To see the full line of Microsoft Press® developer resources, go to:
microsoft.com/mspress/developer



ISBN 0-7356-1803-8
90000
90145 18058
9 780735 618039

U.S.A. $59.99
Canada $86.99

Copyrighted Material

Microsoft®