

WinDriver PCI/ISA/CardBus v7.02 User's Guide

Jungo Ltd

COPYRIGHT

Copyright ©1997 - 2005 Jungo Ltd. All Rights Reserved

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement. The software may be used, copied or distributed only in accordance with that agreement. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means, electronically or mechanically, including photocopying and recording for any purpose without the written permission of Jungo Ltd.

Windows, Win32, Windows 98, Windows Me, Windows CE, Windows NT, Windows 2000, Windows XP and Windows Server 2003 are trademarks of Microsoft Corp. WinDriver and KernelDriver are trademarks of Jungo. Other brand and product names are trademarks or registered trademarks of their respective holders.

Contents

Table of Contents	2
List of Figures	13
1 WinDriver Overview	14
1.1 Introduction to WinDriver	14
1.2 Background	15
1.2.1 The Challenge	15
1.2.2 The WinDriver Solution	16
1.3 How Fast Can WinDriver Go?	17
1.4 Conclusion	17
1.5 WinDriver Benefits	17
1.6 WinDriver Architecture	19
1.7 What Platforms Does WinDriver Support?	20
1.8 Limitations of the Different Evaluation Versions	20
1.9 How Do I Develop My Driver with WinDriver?	21
1.9.1 On Windows Windows 98/Me/NT/2000/XP/Server 2003, Linux and Solaris	21
1.9.2 On Windows CE	21
1.9.3 On VxWorks	22
1.10 What Does the WinDriver Toolkit Include?	22
1.10.1 WinDriver Modules	23
1.10.2 Utilities	24
1.10.3 WinDriver's Specific Chipset Support	24
1.10.4 Samples	25
1.11 Can I Distribute the Driver Created with WinDriver?	25
1.12 Identifying the Right Tool for Your Development	25
2 Understanding Device Drivers	27
2.1 Device Driver Overview	27
2.2 Classification of Drivers According to Functionality	28

2.2.1	Monolithic Drivers	28
2.2.2	Layered Drivers	29
2.2.3	Miniport Drivers	29
2.3	Classification of Drivers According to Operating Systems	30
2.3.1	WDM Drivers	30
2.3.2	VxD Drivers	31
2.3.3	Unix Device Drivers	31
2.3.4	Linux Device Drivers	31
2.3.5	Solaris Device Drivers	32
2.4	The Entry Point of the Driver	32
2.5	Associating the Hardware to the Driver	32
2.6	Communicating with Drivers	33
3	Installing WinDriver	34
3.1	System Requirements	34
3.1.1	For Windows 98/Me	34
3.1.2	For Windows NT/2000/XP/Server 2003	34
3.1.3	For Windows CE	34
3.1.4	For Linux	35
3.1.5	For Solaris	35
3.1.6	For VxWorks	36
3.2	WinDriver Installation Process	36
3.2.1	Windows 98/Me/NT/2000/XP/Server 2003 WinDriver Installation Instructions	37
3.2.2	Windows CE WinDriver Installation Instructions	39
3.2.2.1	Installing WinDriver CE when Building New CE-based Platforms	39
3.2.2.2	Installing WinDriver CE when Developing Applications for CE Computers	40
3.2.2.3	Windows CE Installation Note	40
3.2.3	Linux WinDriver Installation Instructions	41
3.2.3.1	Preparing the System for Installation	41
3.2.3.2	Installation	42
3.2.4	Solaris WinDriver Installation Instructions	44
3.2.4.1	Restricting Hardware Access on Solaris	46
3.2.5	VxWorks DriverBuilder Installation Instructions	46
3.3	Upgrading Your Installation	47
3.4	Checking Your Installation	48
3.4.1	On Your Windows, Linux and Solaris Machines	48
3.4.2	On Your Windows CE Machine	48
3.4.3	On VxWorks	49
3.5	Uninstalling WinDriver	50
3.5.1	On Windows 98/Me/NT/2000/XP/Server 2003	50

3.5.2	On Linux	53
3.5.3	On Solaris	54
3.5.4	On VxWorks	55
4	Using DriverWizard	56
4.1	An Overview	56
4.2	DriverWizard Walkthrough	57
4.3	DriverWizard Notes	64
4.3.1	Sharing a Resource	64
4.3.2	Disabling a Resource	64
4.3.3	Logging WinDriver API Calls	64
4.3.4	DriverWizard Logger	64
4.3.5	Automatic Code Generation	65
4.3.5.1	Generating the Code	65
4.3.5.2	Generated PCI/PCMCIA/ISA Code	65
4.3.5.3	Compiling the Generated Code	66
4.3.5.4	Visual Basic or Delphi Code Generation	66
4.3.5.5	For Linux and Solaris:	66
4.3.5.6	For Other OSs or IDEs:	66
5	Developing a Driver	67
5.1	Using the DriverWizard to Build a Device Driver	67
5.2	Writing the Device Driver Without the DriverWizard	68
5.2.1	Include the Required WinDriver Files	68
5.2.2	Write Your Code	69
5.3	Developing Your Driver on Windows CE Platforms	70
5.4	Developing in Visual Basic and Delphi	71
5.4.1	Using DriverWizard	71
5.4.2	Samples	71
5.4.3	Kernel PlugIn	71
5.4.4	Creating your Driver	71
6	Debugging Drivers	72
6.1	User-Mode Debugging	72
6.2	Debug Monitor	72
6.2.1	Using Debug Monitor in Graphical Mode	73
6.2.2	Using Debug Monitor in Console Mode	75
6.2.2.1	Using Debug Monitor on Windows CE	75
6.2.2.2	Using Debug Monitor on VxWorks	75
7	Enhanced Support for Specific Chipsets	77
7.1	Overview	77
7.2	Developing a Driver Using the Enhanced Chipset Support	78

8	PCI Express	79
8.1	PCI Express Overview	79
8.2	WinDriver for PCI Express	80
9	Advanced Issues	82
9.1	Performing Direct Memory Access (DMA)	82
9.1.1	Scatter/Gather DMA	83
9.1.1.1	Sample Scatter/Gather DMA Implementation	84
9.1.1.2	What Should You Implement?	86
9.1.2	Contiguous Buffer DMA	87
9.1.2.1	Sample Contiguous Buffer DMA Implementation	87
9.1.2.2	What Should You Implement?	89
9.1.3	Performing DMA on SPARC	90
9.2	Handling Interrupts	90
9.2.1	General – Handling an Interrupt	91
9.2.1.1	Basic Interrupt Handler Code Sequence	91
9.2.1.2	Simplified Interrupt Handling Using windrvr_int_thread.c	92
9.2.1.3	Handling Interrupts with the WDC Library	94
9.2.2	ISA/EISA and PCI Interrupts	96
9.2.2.1	Transfer Commands at Kernel Level (Acknowledging the Interrupt)	96
9.2.3	Improving the Interrupt Handling Rate on VxWorks	99
9.2.4	Interrupts in Windows CE	99
9.2.4.1	Improving Interrupt Latency in Windows CE	101
9.3	Support for 64-bit Operating Systems	102
9.4	Byte Ordering	102
9.4.1	Introduction to Endianness	102
9.4.2	WinDriver Byte Ordering Macros	103
9.4.3	Macros for PCI Target Access	103
9.4.4	Macros for PCI Master Access	104
10	Improving Performance	106
10.1	Overview	106
10.1.1	Performance Improvement Checklist	107
10.2	Improving the Performance of a User-Mode Driver	108
10.2.1	Using Direct Access to Memory-Mapped Regions	108
10.2.2	Accessing I/O-Mapped Regions	108
10.2.3	Performing 64-bit Data Transfers	109
11	Understanding the Kernel PlugIn	111
11.1	Background	111

11.2	Do I Need to Write a Kernel PlugIn Driver?	112
11.3	What Kind of Performance Can I Expect?	112
11.4	Overview of the Development Process	112
11.5	The Kernel PlugIn Architecture	113
11.5.1	Architecture Overview	113
11.5.2	WinDriver's Kernel and Kernel PlugIn Interaction	113
11.5.3	Kernel PlugIn Components	114
11.5.4	Kernel PlugIn Event Sequence	114
11.5.4.1	Opening Handle from the User Mode to a Kernel PlugIn Driver	115
11.5.4.2	Handling User-Mode Requests from the Kernel PlugIn	116
11.5.4.3	Interrupt Handling – Enable/Disable and High Interrupt Request Level Processing	116
11.5.4.4	Interrupt Handling – Deferred Procedure Calls	117
11.5.4.5	Plug and Play and Power Management Events	117
11.6	How Does Kernel PlugIn Work?	118
11.6.1	Minimal Requirements for Creating a Kernel PlugIn Driver	118
11.6.2	Kernel PlugIn Implementation	119
11.6.2.1	Before You Begin	119
11.6.2.2	Write Your KP_Init() Function	119
11.6.2.3	Write Your KP_Open() Function	121
11.6.2.4	Write the Remaining PlugIn Callbacks	125
11.6.3	Sample/Generated Kernel PlugIn Driver Code Overview	125
11.6.4	Kernel PlugIn Sample/Generated Code Directory Structure	126
11.6.4.1	pci_diag and kp_pci Sample Directories	126
11.6.4.2	The Generated DriverWizard Kernel PlugIn Directory	127
11.6.5	Handling Interrupts in the Kernel PlugIn	128
11.6.5.1	Interrupt Handling in User Mode (Without Kernel PlugIn)	129
11.6.5.2	Interrupt Handling in the Kernel (Using a Kernel PlugIn)	130
11.6.6	Message Passing	131
12	Writing a Kernel PlugIn	133
12.1	Determine Whether a Kernel PlugIn is Needed	133
12.2	Prepare the User-Mode Source Code	134
12.3	Create a New Kernel PlugIn Project	134
12.4	Create a Handle to the Kernel PlugIn	135
12.5	Set Interrupt Handling in the Kernel PlugIn	136
12.6	Set I/O Handling in the Kernel PlugIn	137
12.7	Compile Your Kernel PlugIn Driver	137

12.7.1	On Windows	137
12.7.2	On Linux	139
12.7.3	On Solaris	140
12.8	Install Your Kernel PlugIn Driver	141
12.8.1	On Win32 Platforms	141
12.8.2	On Linux	142
12.8.3	On Solaris	142
13	Dynamically Loading Your Driver	144
13.1	Why Do You Need a Dynamically Loadable Driver?	144
13.2	Windows NT/2000/XP/Server 2003 and 98/Me	145
13.2.1	Windows Driver Types	145
13.2.2	The WDREG Utility	145
13.2.2.1	WDM Drivers	146
13.2.2.2	Non-WDM Drivers	147
13.2.3	Dynamically Loading/Unloading windrvr6.sys INF Files	149
13.2.4	Dynamically Loading/Unloading Your Kernel PlugIn Driver	150
13.3	Linux	151
13.4	Solaris	151
14	Distributing Your Driver	152
14.1	Getting a Valid License for WinDriver	152
14.2	Windows 98/Me and Windows 2000/XP/Server 2003	153
14.2.1	Preparing the Distribution Package	153
14.2.2	Installing Your Driver on the Target Computer	153
14.2.3	Installing Your Kernel PlugIn on the Target Computer	156
14.3	Windows NT 4.0	157
14.3.1	Preparing the Distribution Package	157
14.3.2	Installing Your Driver on the Target Computer	157
14.3.3	Installing Your Kernel PlugIn on the Target Computer	158
14.4	Creating an INF File	159
14.4.1	Why Should I Create an INF File?	159
14.4.2	How Do I Install an INF File When No Driver Exists?	159
14.4.3	How Do I Replace an Existing Driver Using the INF File?	161
14.5	Windows CE	163
14.6	Linux	164
14.6.1	WinDriver Kernel Module	164
14.6.2	User-Mode Hardware Control Application/Shared Objects	165
14.6.3	Kernel PlugIn Modules	166
14.6.4	Installation Script	166
14.7	Solaris	166
14.8	VxWorks	167

A	API Reference	168
A.1	PCI/ISA/PCMCIA/CardBus – WDC Library Overview	168
A.2	PCI/PCMCIA/ISA – WDC High Level API	170
A.2.1	Structures, Types and General Definitions	170
A.2.1.1	WDC_DEVICE_HANDLE	170
A.2.1.2	WDC_DRV_OPEN_OPTIONS Definitions	170
A.2.1.3	WDC_DIRECTION Enumeration	171
A.2.1.4	WDC_ADDR_MODE Enumeration	171
A.2.1.5	WDC_ADDR_RW_OPTIONS Enumeration	172
A.2.1.6	WDC_ADDR_SIZE Definitions	173
A.2.1.7	WDC_SLEEP_OPTIONS Definitions	173
A.2.1.8	WDC_DBG_OPTIONS Definitions	174
A.2.1.9	WDC_SLOT_U Union	176
A.2.1.10	WDC_PCI_SCAN_RESULT	176
A.2.1.11	WDC_PCMCIA_SCAN_RESULT	177
A.2.2	WDC_DriverOpen()	178
A.2.3	WDC_DriverClose()	180
A.2.4	WDC_PciScanDevices()	181
A.2.5	WDC_PcmciaScanDevices()	183
A.2.6	WDC_PciGetDeviceInfo()	185
A.2.7	WDC_PcmciaGetDeviceInfo()	188
A.2.8	WDC_PciDeviceOpen()	191
A.2.9	WDC_PcmciaDeviceOpen()	193
A.2.10	WDC_IsaDeviceOpen()	195
A.2.11	WDC_PciDeviceClose()	199
A.2.12	WDC_PcmciaDeviceClose()	200
A.2.13	WDC_IsaDeviceClose()	201
A.2.14	WDC_CardCleanupSetup()	202
A.2.15	WDC_CallKerPlug()	204
A.2.16	WDC_ReadMemXXX()	206
A.2.17	WDC_WriteMemXXX()	207
A.2.18	WDC_ReadAddrXXX()	208
A.2.19	WDC_WriteAddrXXX()	210
A.2.20	WDC_ReadAddrBlock()	212
A.2.21	WDC_WriteAddrBlock()	214
A.2.22	WDC_MultiTransfer()	216
A.2.23	WDC_AddrSpaceIsActive()	219
A.2.24	WDC_PciReadCfgBySlot()	220
A.2.25	WDC_PciWriteCfgBySlot()	222
A.2.26	WDC_PciReadCfgBySlotXXX()	224
A.2.27	WDC_PciWriteCfgBySlotXXX()	226
A.2.28	WDC_PciReadCfg()	228

A.2.29	WDC_PciWriteCfg()	229
A.2.30	WDC_PciReadCfgXXX()	230
A.2.31	WDC_PciWriteCfgXXX()	232
A.2.32	WDC_PcmciaReadAttribSpace()	234
A.2.33	WDC_PcmciaWriteAttribSpace()	235
A.2.34	WDC_PcmciaSetWindow()	236
A.2.35	WDC_PcmciaSetVpp()	238
A.2.36	WDC_DMASContigBufLock()	239
A.2.37	WDC_DMASGBufLock()	242
A.2.38	WDC_DMABufUnlock()	245
A.2.39	WDC_DMASyncCpu()	246
A.2.40	WDC_DMASyncIo()	248
A.2.41	WDC_IntEnable()	250
A.2.42	WDC_IntDisable()	255
A.2.43	WDC_IntIsEnabled()	256
A.2.44	WDC_EventRegister()	257
A.2.45	WDC_EventUnregister()	260
A.2.46	WDC_EventIsRegistered()	261
A.2.47	WDC_SetDebugOptions()	262
A.2.48	WDC_Err()	264
A.2.49	WDC_Trace()	265
A.2.50	WDC_GetWDHandle()	266
A.2.51	WDC_GetDevContext()	267
A.2.52	WDC_GetBusType()	268
A.2.53	WDC_Sleep()	269
A.3	PCI/PCMCIA/ISA – WDC Low Level API	270
A.3.1	WDC_ID_U Union	270
A.3.2	WDC_ADDR_DESC	270
A.3.3	WDC_DEVICE	271
A.3.4	PWDC_DEVICE	272
A.3.5	WDC_MEM_DIRECT_ADDR Macro	272
A.3.6	WDC_ADDR_IS_MEM Macro	274
A.3.7	WDC_ADDR_SPACE_IS_ACTIVE Macro	275
A.3.8	WDC_GET_ADDR_DESC Macro	276
A.3.9	WDC_IS_KP Macro	277
A.4	PCI/PCMCIA/ISA - WD_xxx Functions	278
A.4.1	Calling Sequence WinDriver - PCI/PCMCIA/ISA	278
A.4.2	WD_PciScanCards()	281
A.4.3	WD_PciGetCardInfo()	283
A.4.4	WD_PciConfigDump()	287
A.4.5	WD_PcmciaScanCards()	290
A.4.6	WD_PcmciaGetCardInfo()	292

A.4.7	WD_PcmciaConfigDump()	296
A.4.8	WD_IsapnpScanCards()	299
A.4.9	WD_IsapnpGetCardInfo()	302
A.4.10	WD_IsapnpConfigDump()	305
A.4.11	WD_CardRegister()	307
A.4.12	WD_CardCleanupSetup()	312
A.4.13	WD_CardUnregister()	314
A.4.14	WD_Transfer()	315
A.4.15	WD_MultiTransfer()	318
A.4.16	WD_DMALock()	321
A.4.17	WD_DMAUnlock()	326
A.4.18	WD_DMASyncCpu()	328
A.4.19	WD_DMASyncIo()	330
A.4.20	WD_PcmciaControl()	332
A.4.21	InterruptEnable()	335
A.4.22	InterruptDisable()	339
A.5	PCI/PCMCIA/ISA - Low Level WD_xxx Functions	341
A.5.1	WinDriver Low-Level Interrupt Calling Sequence	341
A.5.2	WD_IntEnable()	342
A.5.3	WD_IntWait()	346
A.5.4	WD_IntCount()	348
A.5.5	WD_IntDisable()	350
A.6	Plug and Play and Power Management	352
A.6.1	Calling Sequence	352
A.6.2	EventRegister()	353
A.6.3	EventUnregister()	357
A.7	General WD_xxx Functions	358
A.7.1	Calling Sequence WinDriver – General Use	358
A.7.2	WD_Open()	360
A.7.3	WD_Version()	361
A.7.4	WD_Close()	363
A.7.5	WD_Debug()	364
A.7.6	WD_DebugAdd()	366
A.7.7	WD_DebugDump()	368
A.7.8	WD_Sleep()	369
A.7.9	WD_License()	371
A.7.10	WD_LogStart()	373
A.7.11	WD_LogStop()	374
A.7.12	WD_LogAdd()	375
A.8	Kernel PlugIn - User-Mode Functions	376
A.8.1	WD_KernelPlugInOpen()	376
A.8.2	WD_KernelPlugInClose()	378

A.8.3	WD_KernelPlugInCall()	379
A.8.4	WD_IntEnable()	381
A.9	Kernel PlugIn - Kernel-Mode Functions	383
A.9.1	KP_Init()	384
A.9.2	KP_Open()	386
A.9.3	KP_Close()	388
A.9.4	KP_Call()	389
A.9.5	KP_Event()	392
A.9.6	KP_IntEnable()	394
A.9.7	KP_IntDisable()	396
A.9.8	KP_IntAtIrql()	397
A.9.9	KP_IntAtDpc()	399
A.9.10	COPY_TO_USER_OR_KERNEL, COPY_FROM_USER_OR_KERNEL	401
A.9.11	Kernel PlugIn Synchronization APIs	402
A.9.11.1	Kernel PlugIn Synchronization Types	402
A.9.11.2	kp_spinlock_init()	403
A.9.11.3	kp_spinlock_wait()	404
A.9.11.4	kp_spinlock_release()	405
A.9.11.5	kp_spinlock_uninit()	406
A.9.11.6	kp_interlocked_init()	407
A.9.11.7	kp_interlocked_uninit()	408
A.9.11.8	kp_interlocked_increment()	409
A.9.11.9	kp_interlocked_decrement()	410
A.9.11.10	kp_interlocked_add()	411
A.9.11.11	kp_interlocked_read()	412
A.9.11.12	kp_interlocked_set()	413
A.9.11.13	kp_interlocked_exchange()	414
A.10	Kernel PlugIn - Structure Reference	415
A.10.1	WD_KERNEL_PLUGIN	415
A.10.2	WD_INTERRUPT	416
A.10.3	WD_KERNEL_PLUGIN_CALL	417
A.10.4	KP_INIT	418
A.10.5	KP_OPEN_CALL	419
A.11	WinDriver Status/Error Codes	421
A.11.1	Introduction	421
A.11.2	Status Codes Returned by WinDriver	421
A.12	User-Mode Utility Functions	423
A.12.1	Stat2Str()	423
A.12.2	get_os_type()	424
A.12.3	ThreadStart()	425
A.12.4	ThreadWait()	426

A.12.5	OsEventCreate()	427
A.12.6	OsEventClose()	428
A.12.7	OsEventWait()	429
A.12.8	OsEventSignal()	430
A.12.9	OsEventReset()	431
A.12.10	OsMutexCreate()	432
A.12.11	OsMutexClose()	433
A.12.12	OsMutexLock()	434
A.12.13	OsMutexUnlock()	435
A.12.14	PrintDbgMessage()	436
B	Troubleshooting and Support	438
C	Evaluation Version Limitations	439
C.1	Windows 98/Me/2000/XP/Server 2003	439
C.2	Windows CE	439
C.3	Linux	439
C.4	Solaris	440
C.5	VxWorks	440
C.6	DriverWizard GUI	440
D	Purchasing WinDriver	441
E	Distributing Your Driver – Legal Issues	442
F	Additional Documentation	443

List of Figures

1.1	WinDriver Architecture	19
2.1	Monolithic Drivers	28
2.2	Layered Drivers	29
2.3	Miniport Drivers	30
4.1	Select Your Plug and Play Device	58
4.2	DriverWizard INF File Information	60
4.3	PCI Diagnostics Screen	61
4.4	Code Generation Options	62
4.5	Notification Events	62
6.1	Start Debug Monitor	73
6.2	Set Trace Options	74
11.1	Kernel PlugIn Architecture	113
11.2	Interrupt Handling Without Kernel PlugIn	129
11.3	Interrupt Handling With the Kernel PlugIn	130
A.1	WinDriver PCI/PCMCIA/ISA Calling Sequence	279
A.2	Low-Level WinDriver Interrupt API Calling Sequence	341
A.3	Plug and Play Calling Sequence	352
A.4	WinDriver API Calling Sequence	358

Chapter 1

WinDriver Overview

In this chapter you will explore the uses of WinDriver, and learn the basic steps of creating your driver.

NOTE

This manual outlines WinDriver's support for **PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express** devices. The WinDriver toolkit also supports the Universal Serial Bus **USB**. For detailed information regarding WinDriver USB, please refer to the WinDriver Product Line page on our web-site (<http://www.jungo.com/windriver.html>) and to the WinDriver USB User's Manual, which is available on-line at: <http://www.jungo.com/support/manuals.html#manuals>.

1.1 Introduction to WinDriver

WinDriver is a development toolkit that dramatically simplifies the difficult task of creating device drivers and hardware access applications. WinDriver includes a wizard and code generation features that automatically detect your hardware and generate the driver to access it from your application. The driver and application you develop using WinDriver is source code compatible between all supported operating systems (WinDriver currently supports Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks.). The driver is binary compatible between Windows 98/Me/NT/2000/XP/Server 2003. Bus architecture support includes PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express (PCMCIA is supported only on Windows 2000/XP/Server 2003). WinDriver provides

a complete solution for creating high performance drivers that handle interrupts and I/O at optimal rates.

Don't let the size of this manual fool you. WinDriver makes developing device drivers an easy task that takes hours instead of months. Most of this manual deals with the features that WinDriver offers to the advanced user. However, most developers will find that reading this chapter and glancing through the DriverWizard and function reference chapters is all they need to successfully write their driver.

WinDriver supports development for all PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express chipsets. Enhanced support is offered for PLX/Altera/Marvell/AMCC/QuickLogic/Xilinx PCI chips PCI chipsets, as outlined in Chapter [7] of the manual.

Chapter 10 explains how to tune your driver code to achieve optimal performance, with special emphasis on WinDriver's Kernel PlugIn feature. This feature allows the developer to write and debug the entire device driver in the user mode, and later drop performance critical portions of the code into kernel mode. In this way the driver achieves optimal kernel-mode performance, while the developer need not sacrifice the ease of user-mode development. For a detailed overview of the Kernel PlugIn, refer to Chapters 11 – 12.

Visit Jungo's web site at <http://www.jungo.com> for the latest news about WinDriver and other driver development tools that Jungo offers.

Good luck with your project!

1.2 Background

1.2.1 The Challenge

In protected operating systems such as Windows, Linux and Solaris, a programmer cannot access hardware directly from the application level (user mode), where development work is usually done. Hardware can only be accessed from within the operating system itself (kernel mode or Ring-0), utilizing software modules called device drivers. In order to access a custom hardware device from the application level, a programmer must do the following:

- Learn the internals of the operating system he is working on (Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks).
- Learn how to write a device driver.
- Learn new tools for developing/debugging in kernel mode (DDK, ETK, DDI/DKI).

- Write the kernel-mode device driver that does the basic hardware input/output.
- Write the application in user mode that accesses the hardware through the device driver written in kernel mode.
- Repeat the first four steps for each new operating system on which the code should run.

1.2.2 The WinDriver Solution

Easy Development: WinDriver enables Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks programmers to create **PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express**-based device drivers in an extremely short time. WinDriver allows you to create your driver in the familiar user-mode environment, using MSDEV/Visual C/C++, Borland Delphi, Borland C++, Visual Basic, GCC or any other 32-bit compiler. You do not need to have any device driver knowledge, nor do you have to be familiar with operating system internals, kernel programming, the DDK, ETK or DDI/DKI.

Cross Platform: The driver created with WinDriver will run on Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks. In other words – write it once, run it on many platforms.

Friendly Wizards: DriverWizard (included) is a graphical diagnostics tool that lets you view/define the device's resources and write to/read from the hardware before writing a single line of code. The hardware is diagnosed with just a few clicks of the mouse: memory ranges are read/written to, registers are defined and toggled and interrupts are checked. Once the device is operating to your satisfaction, DriverWizard creates the skeletal driver source code, giving access functions to all the resources on the hardware.

Kernel-Mode Performance: WinDriver's API is optimized for performance. For drivers that need kernel-mode performance, WinDriver offers the Kernel PlugIn. This powerful feature enables you to create and debug your code in user mode and run the performance-critical parts of your code (such as the interrupt handling or access to I/O mapped memory ranges) in kernel mode, thereby achieving kernel-mode performance (zero performance degradation). This unique feature allows the developer to run user-mode code in the OS kernel without having to learn how the kernel works. For a detailed overview of this feature, see Chapter 11. There is no need to use the Kernel PlugIn when working with Windows CE or VxWorks, since there is no separation between user and kernel modes in these operating systems. This enables you to achieve optimal performance

from user-mode code. Section 9.2.3 explains how you can use WinDriver to improve the interrupt handling rate on VxWorks, in place of a Kernel PlugIn driver.

1.3 How Fast Can WinDriver Go?

You can expect the same throughput using the WinDriver Kernel PlugIn as when using a custom kernel driver. Throughput is constrained only by the limitations of your operating system and hardware. A rough estimate of the throughput you can obtain using the Kernel PlugIn is approximately 100,000 interrupts per second.

1.4 Conclusion

Using WinDriver, a developer need only do the following to create an application that accesses the custom hardware:

- Start DriverWizard and detect the hardware and its resources.
- Automatically generate the device driver code from within DriverWizard, or use one of the WinDriver samples as the basis for the application (see Chapter 7 for an overview of WinDriver's enhanced support for specific chipsets).
- Modify the user-mode application, as needed, using the generated/sample functions to implement the desired functionality for your application.

Your hardware access application will run on all the supported platforms: Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks— just re-compile the code for the target platform. (The code is binary compatible between Windows 98/Me/NT/2000/XP/Server 2003 platforms, so there is no need to rebuild the code when porting the driver between these operating systems.)

1.5 WinDriver Benefits

- Easy user-mode driver development.
- Kernel PlugIn for high-performance drivers.
- Friendly DriverWizard allows hardware diagnostics without writing a single line of code.

- Automatically generates the driver code for the project in C, Delphi (Pascal) or Visual Basic.
- Support for any PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express device, regardless of manufacturer.
- Enhanced support for PLX/Altera/Marvell/AMCC/QuickLogic/Xilinx PCI chips allows the developer to disregard the PCI chip details.
- Applications are binary-compatible across Windows 98/Me/NT/2000/XP/Server 2003.
- Applications are source code compatible across Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks.
- Can be used with common development environments, including MSDEV/Visual C/C++, MSDEV .NET, Borland Delphi, Borland C++ Builder, Visual Basic, GCC or any other 32-bit compiler.
- No DDK, ETK, DDI or any system-level programming knowledge required.
- Supports I/O, DMA, interrupt handling and access to memory-mapped cards.
- Supports multiple CPU and multiple bus platforms (PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express).
- Support for 64-bit PCI data transfers.
- Includes dynamic driver loader.
- Comprehensive documentation and help files.
- Detailed examples in C, C#, Visual Basic .NET, Delphi and Visual Basic 6.0.
- WHQL certifiable driver (Windows).
- Two months of free technical support.
- No runtime fees or royalties.

1.6 WinDriver Architecture

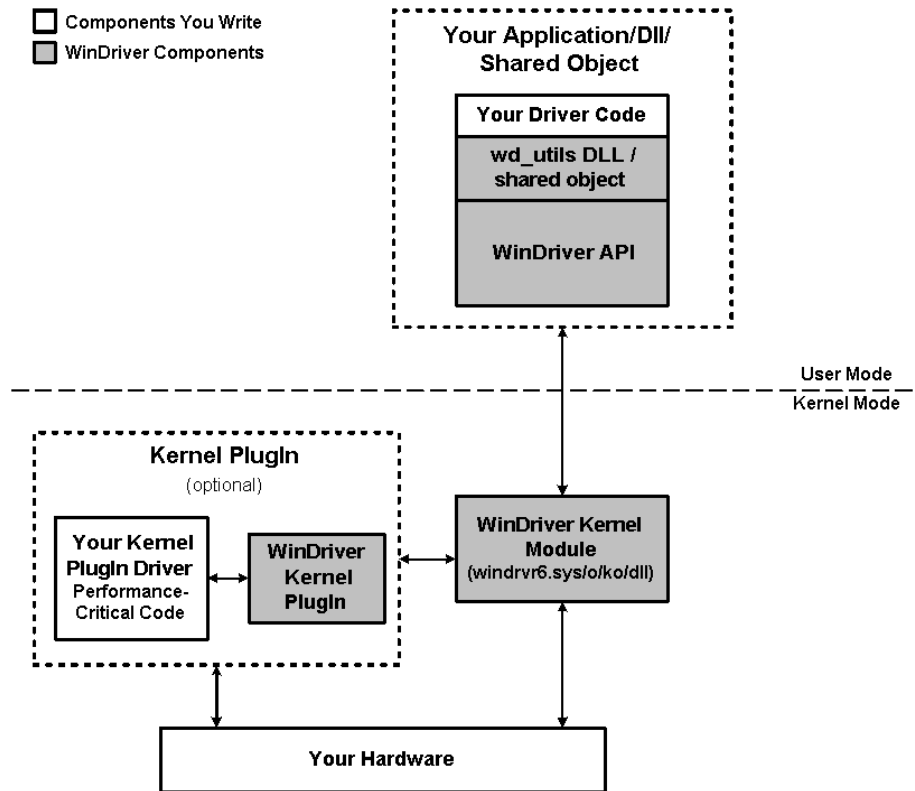


Figure 1.1: WinDriver Architecture

For hardware access, your application calls one of the WinDriver user-mode functions. The user-mode function calls the WinDriver kernel, which accesses the hardware for you through the native calls of the operating system.

WinDriver's design minimizes performance hits on your code, even though it is running in user mode. However, some hardware drivers have high performance requirements that cannot be achieved in user mode. This is where WinDriver's edge sharpens. After easily creating and debugging your code in user mode, you may drop the performance-critical modules of your code (such as a hardware interrupt handler) into the WinDriver Kernel PlugIn without changing them at all. Now, the WinDriver kernel calls this module from kernel mode, thereby achieving maximal performance. This allows you to program and debug in user mode, and still achieve

kernel performance where needed. For a detailed overview of the Kernel PlugIn feature, see Chapter 11.

In Windows CE and VxWorks there is no separation between user mode and kernel mode, therefore, you can achieve optimal performance directly from user mode, without needing to use the Kernel PlugIn in these operating systems.

1.7 What Platforms Does WinDriver Support?

WinDriver supports Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks.

The same source code will run on all supported platforms – simply re-compile it for the target platform. The source code is binary compatible across Windows 98/Me/NT/2000/XP/Server 2003, so executables created with WinDriver can be ported between these operating systems without re-compilation.

Even if your code is meant only for one of the supported operating systems, using WinDriver will give you the flexibility to move your driver to another operating system in the future without needing to change your code.

1.8 Limitations of the Different Evaluation Versions

All the evaluation versions of WinDriver are full featured. No functions are limited or crippled in any way. The evaluation version of WinDriver varies from the registered version in the following ways:

- Each time WinDriver is activated, an **Un-registered** message appears.
- When using the DriverWizard, a dialog box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- In the Linux, Solaris, VxWorks and CE versions, the driver will remain operational for 60 minutes, after which time it must be restarted.
- The Windows evaluation version expires 30 days from the date of installation.

For more details please refer to appendix C.

1.9 How Do I Develop My Driver with WinDriver?

1.9.1 On Windows Windows 98/Me/NT/2000/XP/Server 2003, Linux and Solaris

1. Start DriverWizard and use it to diagnose your hardware – see details in Chapter 4.
2. Let DriverWizard generate skeletal code for your driver, or use one of the WinDriver samples as the basis for your driver application (see Chapter [7] for details regarding WinDriver's enhanced support for specific chipsets).
3. Modify the generated/sample code to suit your application's needs.
4. Run and debug your driver in the user mode.
5. If your code contains performance-critical sections, refer to Chapter 10 for suggestions on how to improve your driver's performance.

NOTE

The code generated by DriverWizard is in fact a diagnostics program that contains functions that read and write to any resource detected or defined (including custom-defined registers), enables your card's interrupts, listens to them, and more.

1.9.2 On Windows CE

1. Plug your hardware into a Windows host machine.
2. Diagnose your hardware using DriverWizard.
3. Let DriverWizard generate your driver's skeletal code.
4. Modify this code using eMbedded Visual C++ to meet your specific needs. If you are using Platform Builder, activate it and insert the generated *.pbp into your workspace.
5. Test and debug your code and hardware from the CE emulation running on the host machine.

NOTE

ISAPnP is not supported under Windows CE.

TIP

If you cannot plug your hardware into a Windows host machine, you can still use DriverWizard to generate code for your device by manually entering all your resources in the wizard. Let DriverWizard generate your code and then test it on your hardware using a serial connection. After verifying that the generated code works properly, modify it to meet your specific needs. You may also use (or combine) any of the sample files for your driver's skeletal code.

1.9.3 On VxWorks

1. Plug your hardware into a Windows host machine.
2. Diagnose your hardware using DriverWizard for Windows. Refer to Chapter 4 for details.
3. Let DriverWizard generate your driver's skeletal code and project makefile for Tornado.
4. Move the code to your tornado environment and compile it.
5. Modify this code using tornado development environment, or any other 32-bit development environment, to meet your specific needs.

1.10 What Does the WinDriver Toolkit Include?

- A printed version of this manual
- Two months of free technical support (Phone/Fax/Email)
- WinDriver modules
- The WinDriver CD
 - Utilities
 - Chipset support APIs
 - Sample files

1.10.1 WinDriver Modules

- WinDriver (**WinDriver\include**) – the general purpose hardware access toolkit. The main files here are:
 - **windrvr.h**: Declarations and definitions of WinDriver’s basic API.
 - **wdc_lib.h** and **wdc_defs.h**: Declarations and definitions of the WinDriver Card (WDC) library, which provides convenient wrapper APIs for accessing PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express devices (see Chapter A.1).
 - **windrvr_int_thread.h**: Declarations of convenient wrapper functions to simplify interrupt handling.
 - **windrvr_events.h**: Declarations of APIs for handling and Plug-and-Play and power management events.
 - **utils.h**: Declarations of general utility functions.
 - **status_strings.h**: Declarations of API for converting WinDriver status codes to descriptive error strings.
- DriverWizard (**WinDriver/wizard/wdwizard**) – a graphical tool that diagnoses your hardware and enables you to easily generate code for your driver (refer to Chapter 4 for details).
- Graphical Debugger (**WinDriver/util/wddebug_gui**) – a graphical debugging tool that collects information about your driver as it runs.
WinDriver also includes a console version of this program (**WinDriver/util/wddebug**), which can be used on platforms that have no GUI support, such as Windows CE or VxWorks.
For details regarding the Debug Monitor, refer to section [6.2].
- WinDriver distribution package (**WinDriver/redist**) – the files you include in the driver distribution to customers.
- WinDriver Kernel PlugIn – the files and samples needed to create a kernel-mode Kernel PlugIn driver (refer to Chapter [11] for details.)
- This manual – the full WinDriver manual (this document) in PDF, Windows Help and HTML formats can be found under the **WinDriver/docs/** directory.

1.10.2 Utilities

- **PCI_DIAG.EXE** (/WinDriver/util/**pci_diag.exe**) – used for reading/writing PCI configuration registers, and accessing the PCI card's IO and memory ranges.
- **PCI_SCAN.EXE** (/WinDriver/util/**pci_scan.exe**) – used to obtain a list of the PCI cards installed and the resources allocated for each of them.
- **PCI_DUMP.EXE** (/WinDriver/util/**pci_dump.exe**) – used to obtain a dump of all the PCI configuration registers of the PCI cards installed.
- **PCMCIA_SCAN.EXE** (/WinDriver/util/**pcmcia_scan.exe**) – used to obtain a list of the PCMCIA cards installed and the resources allocated for each of them.

The Windows CE version also includes:

- **\REDIST\... \X86EMU\WINDRVR_CE_EMU.DLL**: DLL that communicates with the WinDriver kernel—for the x86 HPC emulation mode of Windows CE.
- **\REDIST\... \X86EMU\WINDRVR_CE_EMU.LIB**: an import library that is used to link with WinDriver applications that are compiled for the x86 HPC emulation mode of Windows CE.

1.10.3 WinDriver's Specific Chipset Support

WinDriver provides custom wrapper APIs and sample code for major PCI chipsets (see Chapter 7), including for the following chipsets:

- PLX 9030, 9050, 9052, 9054, 9080, 9056 and 9656 – found under the **WinDriver/plx** directory
- Marvell GT64 – **WinDriver/marvell/gt64**
- WinDriver AMCC APIs (for the AMCC S5933 PCI bridges) – **WinDriver/amcc**
- Altera pci_dev_kit – **WinDriver/altera**
- Xilinx VirtexII – **WinDriver/xilinx/VirtexII**

The samples directories typically include the following sub-directories:

- **<vendor>/lib/** – the custom API for the enhanced-support chip(s), written using the WinDriver API.
- **<chip>/xxx_diag/** – a sample diagnostics application for a specific chip, which was written using the custom API from the **lib/** directory. The sample application can be compiled and executed "as-is".

1.10.4 Samples

In addition to the samples provided for specific chipsets [1.10.3], WinDriver includes a variety of samples that demonstrate how to use WinDriver's API to communicate with your device and perform various driver tasks.

- **WinDriver/samples** – C samples.
These samples also include the source code for the utilities listed above [1.10.2].
- **WinDriver/delphi/samples** – Delphi (Pascal) samples
- **WinDriver/vb/samples** – Visual Basic samples

1.11 Can I Distribute the Driver Created with WinDriver?

Yes. WinDriver is purchased as a development toolkit, and any device driver created using WinDriver may be distributed, royalties free, in as many copies as you wish. See the license agreement (**WinDriver/docs/license.txt**) for more details.

1.12 Identifying the Right Tool for Your Development

Jungo offers two driver development products: WinDriver and KernelDriver.

WinDriver is designed for monolithic type user-mode drivers. It enables you to access your hardware directly from within your user-mode application, without writing a kernel-mode device driver. Using WinDriver you can either access your hardware directly from your application (in user mode) or write a DLL that you can call from many different applications.

In addition, WinDriver provides a complete solution for high-performance drivers. Using WinDriver's Kernel PlugIn [11], you can drop your user-mode code into the kernel and reach full kernel-mode performance.

A PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express driver developed with WinDriver will run on Windows 98/Me/2000/XP/Server2003/CE.NET and Linux (PCMCIA is only supported on Windows 2000/XP/Server 2003; ISAPnP is not supported on Windows CE.)

Typically, using WinDriver a developer that has no previous driver knowledge can get a driver running in a matter of a few hours (compared to several weeks with a kernel-mode driver).

KernelDriver is intended for creating standard operating system internal drivers that require hardware access and that must communicate with the operating system or must be implemented in the kernel.

A PCI/PCMCIA/CardBus/ISA/ISAPnP/EISA/CompactPCI/PCI Express driver created with KernelDriver can run on Windows 98/Me/NT/2000/XP/Server 2003/C and Linux. KernelDriver dramatically simplifies the difficult task of creating kernel-mode device drivers, by providing a hardware access API in the kernel mode, which is portable across the supported operating systems.

Chapter 2

Understanding Device Drivers

This chapter provides you with a general introduction to device drivers and takes you through the structural elements of a device driver.

NOTE

Using WinDriver, you do not need to familiarize yourself with the internal workings of driver development. As explained in Chapter 1 of the manual, WinDriver enables you to communicate with your hardware and develop a driver for your device from the user mode, using only WinDriver's simple APIs, without any need for driver or kernel development knowledge.

2.1 Device Driver Overview

Device drivers are the software segments that provides an interface between the operating system and the specific hardware devices such as terminals, disks, tape drives, video cards and network media. The device driver brings the device into and out of service, sets hardware parameters in the device, transmits data from the kernel to the device, receives data from the device and passes it back to the kernel, and handles device errors.

A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

2.2 Classification of Drivers According to Functionality

There are numerous driver types, differing in their functionality. This subsection briefly describes three of the most common driver types.

2.2.1 Monolithic Drivers

Monolithic drivers are device drivers that embody all the functionality needed to support a hardware device. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through I/O control commands (IOCTLs) and drives the hardware using calls to the different DDK, ETK, DDI/DKI functions.

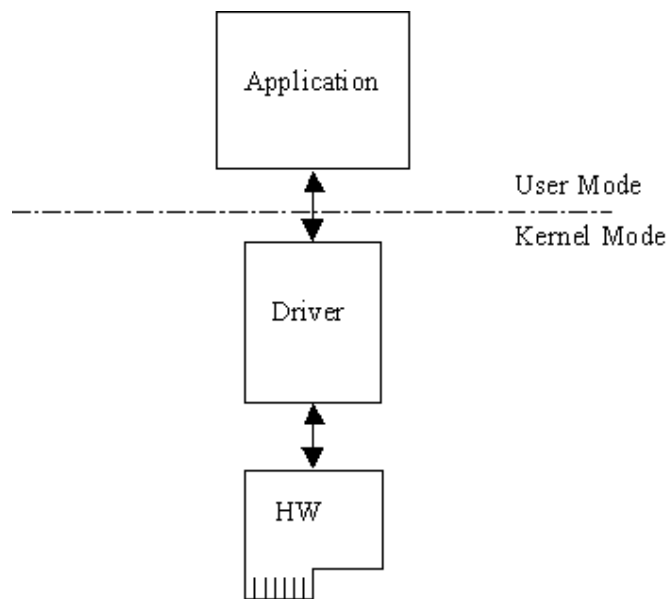


Figure 2.1: Monolithic Drivers

Monolithic drivers are supported in all operating systems including all Windows platforms and all Unix platforms.

2.2.2 Layered Drivers

Layered drivers are device drivers that are part of a stack of device drivers that together process an I/O request. An example of a layered driver is a driver that intercepts calls to the disk and encrypts/decrypts all data being transferred to/from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption/decryption.

Layered drivers are sometimes also known as filter drivers, and are supported in all operating systems including all Windows platforms and all Unix platforms.

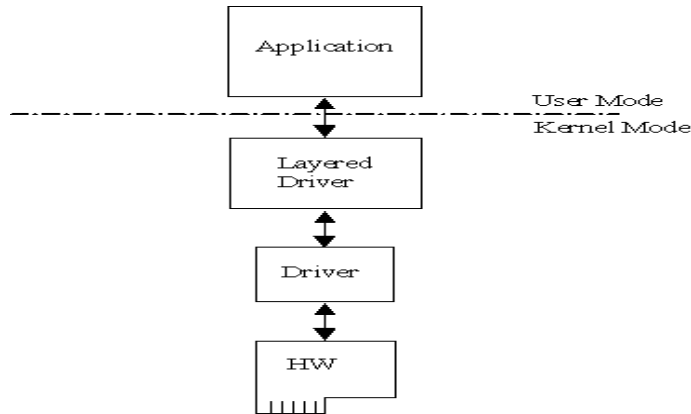


Figure 2.2: Layered Drivers

2.2.3 Miniport Drivers

A Miniport driver is an add-on to a class driver that supports miniport drivers. It is used so the miniport driver does not have to implement all of the functions required of a driver for that class. The class driver provides the basic class functionality for the miniport driver.

A class driver is a driver that supports a group of devices of common functionality, such as all HID devices or all network devices.

Miniport drivers are also called miniclass drivers or minidrivers, and are supported in the Windows NT (or 2000) family, namely Windows NT/2000/XP and Server 2003.

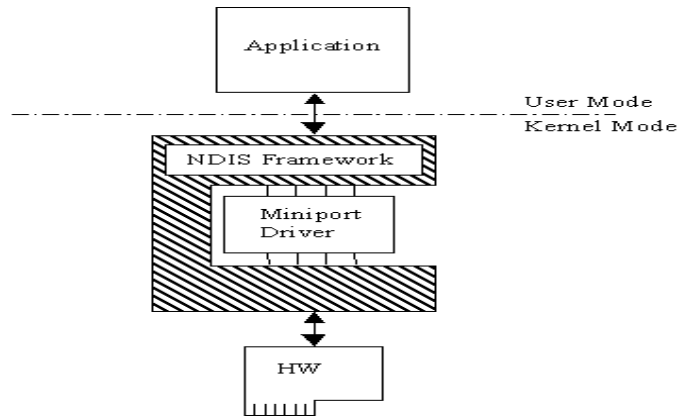


Figure 2.3: Miniport Drivers

Windows NT/2000/XP/Server 2003 provide several driver classes (called ports) that handle the common functionality of their class. It is then up to the user to add only the functionality that has to do with the inner workings of the specific hardware.

The NDIS miniport driver is one example of such a driver. The NDIS miniport framework is used to create network drivers that hook up to NT's communication stacks, and are therefore accessible to common communication calls used by applications. The Windows NT kernel provides drivers for the various communication stacks and other code that is common to communication cards. Due to the NDIS framework, the network card developer does not have to write all of this code, only the code that is specific to the network card he is developing.

2.3 Classification of Drivers According to Operating Systems

2.3.1 WDM Drivers

WDM (Windows Driver Model) drivers are kernel-mode drivers within the Windows NT and Windows 98 operating system families. Windows NT family includes Windows NT/2000/XP/Server 2003, and Windows 98 family includes Windows 98 and Windows Me.

WDM works by channeling some of the work of the device driver into portions of the code that are integrated into the operating system. These portions of code handle all of the low-level buffer management, including DMA and Plug and Play (Pnp) device enumeration.

WDM drivers are PnP drivers that support power management protocols, and include monolithic drivers, layered drivers and miniport drivers.

2.3.2 VxD Drivers

VxD drivers are Windows 95/98/Me Virtual Device Drivers, often called VxDs because the filenames end with the .vxd extension. VxD drivers are typically monolithic in nature. They provide direct access to hardware and privileged operating system functions. VxD drivers can be stacked or layered in any fashion, but the driver structure itself does not impose any layering.

2.3.3 Unix Device Drivers

In the classic Unix driver model, devices belong to one of three categories: character (char) devices, block devices and network devices. Drivers that implement these devices are correspondingly known as char drivers, block drivers or network drivers. Under Unix, drivers are code units linked into the kernel that run in privileged kernel mode. Generally, driver code runs on behalf of a user-mode application. Access to Unix drivers from user-mode applications is provided via the file system. In other words, devices appear to the applications as special device files that can be opened.

Unix device drivers are either layered or monolithic drivers. A monolithic driver can be perceived as a one-layer layered driver.

2.3.4 Linux Device Drivers

Linux device drivers are based on the classic Unix device driver model. In addition, Linux introduces some new characteristics.

Under Linux, a block device can be accessed like a character device, as in Unix, but also has a block-oriented interface that is invisible to the user or application.

Traditionally, under Unix, device drivers are linked with the kernel, and the system is brought down and restarted after installing a new driver. Linux introduces the concept of a dynamically loadable driver called a module. Linux modules can be loaded or removed dynamically without requiring the system to be shut down. A Linux driver can be written so that it is statically linked or written in a modular form that allows it to be dynamically loaded. This makes Linux memory usage very efficient because modules can be written to probe for their own hardware and unload themselves if they cannot find the hardware they are looking for.

Like Unix device drivers, Linux device drivers are either layered or monolithic drivers.

2.3.5 Solaris Device Drivers

Solaris device drivers are also based on the classic Unix device driver model. Like Linux drivers, Solaris drivers may be either statically linked with the kernel or dynamically loaded and removed from the kernel.

Like Unix and Linux device drivers, Solaris device drivers are either layered or monolithic drivers.

2.4 The Entry Point of the Driver

Every device driver must have one main entry point, like the `main()` function in a C console application. This entry point is called `DriverEntry()` in Windows and `init_module()` in Linux. When the operating system loads the device driver, this driver entry procedure is called.

There is some global initialization that every driver needs to perform only once when it is loaded for the first time. This global initialization is the responsibility of the `DriverEntry()/init_module()` routine. The entry function also registers which driver callbacks will be called by the operating system. These driver callbacks are operating system requests for services from the driver. In Windows, these callbacks are called *dispatch routines*, and in Linux they are called *file operations*. Each registered callback is called by the operating system as a result of some criteria, such as disconnection of hardware, for example.

2.5 Associating the Hardware to the Driver

Operating systems differ in how they link a device to its driver.

In Windows, the link is performed by the INF file, which registers the device to work with the driver. This association is performed before the `DriverEntry()` routine is called. The operating system recognizes the device, looks up in its database which INF file is associated with the device, and according to the INF file, calls the driver's entry point.

In Linux, the link between a device and its driver is defined in the `init_module()` routine. The `init_module()` routine includes a callback which states what hardware the driver is designated to handle. The operating system calls the driver's entry point, based on the definition in the code.

2.6 Communicating with Drivers

A driver can create an instance, thus enabling an application to open a handle to the driver through which the application can communicate with it.

The applications communicate with the drivers using a file access API (Application Program Interface). Applications open a handle to the driver using `CreateFile()` call (in Windows), or `open()` call (in Linux) with the name of the device as the file name. In order to read from and write to the device, the application calls `ReadFile()` and `WriteFile()` (in Windows), or `read()`, `write()` in Linux.

Sending requests is accomplished using an I/O control call, called `DeviceIoControl()` (in Windows), and `ioctl()` in Linux. In this I/O control call, the application specifies:

- The device to which the call is made (by providing the device's handle).
- An IOCTL code that describes which function this device should perform.
- A buffer with the data on which the request should be performed.

The IOCTL code is a number that the driver and the requester agree upon for a common task.

The data passed between the driver and the application is encapsulated into a structure. In Windows, this structure is called an I/O Request Packet (IRP), and is encapsulated by the I/O Manager. This structure is passed on to the device driver, which may modify it and pass it down to other device drivers.

Chapter 3

Installing WinDriver

This chapter takes you through the WinDriver installation process, and shows you how to verify that your WinDriver is properly installed. The last section discusses the uninstall procedure.

3.1 System Requirements

3.1.1 For Windows 98/Me

- An x86 processor
- Any 32-bit development environment supporting C, VB or Delphi

3.1.2 For Windows NT/2000/XP/Server 2003

- An x86 processor
- Any 32-bit development environment supporting C, VB or Delphi
- Windows NT: Service Pack 3 or higher (Service Pack 6 is recommended)

3.1.3 For Windows CE

- An x86 / MIPS / ARM Windows CE 4.x - 5.0 (.Net) target platform
- Windows NT/2000/XP/Server 2003 host development platform

- Microsoft eMbedded Visual C++ with a corresponding target SDK or Microsoft Platform Builder with corresponding BSP (Board Support Package) for the target platform

3.1.4 For Linux

- Any 32-bit x86 architecture with a Linux 2.2.x, 2.4.x or 2.6.x kernel
or:
An x86 64-bit architecture – AMD64 or Intel EM64T (**x86_64**) – with a Linux 2.4.x or 2.6.x kernel
or:
Any PowerPC 32-bit architecture with a Linux 2.4.x or 2.6.x kernel
- A GCC compiler for WinDriver installation and for Kernel PlugIn

NOTE

The version of the GCC compiler should match the compiler version used for building the running Linux kernel.

- Any 32-bit or 64-bit development environment (depending on your target configuration) supporting C for user mode.
- On your development PC: **glibc2.3.x**
- **libstdc++.so.5** is required for running GUI WinDriver applications (e.g. DriverWizard [4] ; Debug Monitor [6.2]).

3.1.5 For Solaris

- Solaris 8.0/9.0

NOTE

For Solaris 8 it is recommended to use update 3 or higher (available from Sun: <http://www.sun.com>).

- 64-bit or 32-bit kernel on SPARC platform
or
32-bit kernel on x86 platform
- Any development environment supporting C (such as GCC)
- WinDriver 5.22 is still provided for Solaris 2.6/7.0 32-bit kernel on Intel x86 platform.

NOTE

If you have chosen a development environment other than GCC, make sure *libgcc* is installed on your computer. You may download it from <http://www.sunfreeware.com>.

Set the LD_LIBRARY_PATH to the location of your *libgcc*, a probable location would be:

LD_LIBRARY_PATH= /usr/local/lib:/usr/local/lib/sparcv9

3.1.6 For VxWorks

- VxWorks 5.4
- Windows host development platform
- Tornado 2.0 IDE
- Target Platform running a processor that has a BSP (Board Support Package) compatible with the list of CPU/BSP combinations supported by DriverBuilder.

For an up-to-date list see:

<http://www.jungo.com/db-vxworks.html#platforms>

For information on BSP compatibility, please contact your nearest WindRiver Systems support representative.

3.2 WinDriver Installation Process

The WinDriver CD contains all versions of WinDriver for all the different operating systems. The CD's root directory contains the Windows 98/Me and NT/2000/XP/Server 2003 version. This will automatically begin when you insert the CD into your CD drive. The other versions of WinDriver are located in subdirectories, i.e., **\Linux**, **\Wince** and so on.

3.2.1 Windows 98/Me/NT/2000/XP/Server 2003 WinDriver Installation Instructions

NOTE

You must have administrative privileges in order to install WinDriver on Windows 98, Me, NT, 2000, XP and Server 2003.

1. Insert the WinDriver CD into your CD-ROM drive.
(When installing WinDriver by downloading it from Jungo's web site instead of using the WinDriver CD, double click the downloaded WinDriver file (**WDxxx.EXE**) in your download directory, and go to Step 3).
2. Wait a few seconds until the installation program starts automatically. If for some reason it does not start automatically, double-click the file **WDxxx.EXE** (where 'xxx' is the version number) and click the **Install WinDriver** button.
3. Read the license agreement carefully, and click **Yes** if you accept its terms.
4. Choose the destination location in which to install WinDriver.
5. In the **Setup Type** screen, choose one of the following:
 - **Typical** – to install all WinDriver modules (generic WinDriver toolkit + specific chipset APIs)
 - **Compact** – to install only the generic WinDriver toolkit
 - **Custom** – to choose which modules of WinDriver to install; you may choose which APIs will be installed
6. After the installer finishes copying the required files, choose whether to view the Quick Start guides.
7. You may be prompted to reboot your computer.

NOTE

The WinDriver installation defines a **WD_BASEDIR** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [4] code generation – it determines the default directory for saving your generated code and is used in the include paths of the generated project/make files. This variable is also used from the sample Kernel PlugIn projects and makefiles.

Therefore, if you decide to change the name and/or location of your WinDriver directory after the installation, you should also edit the value of the **WD_BASEDIR** environment variable and set it to point to the location of your new WinDriver directory. You can edit the value of **WD_BASEDIR** by following these steps:

1. Open the **System Properties** dialog: **Start | System | Control Panel | System**.
2. In the **Advanced** tab, click the **Environment Variables** button.
3. In the **System variables** box, select the **WD_BASEDIR** variable and click the **Edit ...** button or double-click the mouse on the variable.
4. In the **Edit System Variable** dialog, replace the **Variable Value** with the full path to your new WinDriver directory, then click **OK**, and click **OK** again from the **System Properties** dialog.

The following steps are for registered users only:

In order to register your copy of WinDriver with the license you received from Jungo, follow the steps below:

1. Activate DriverWizard GUI (**Start | Programs | WinDriver | DriverWizard**).
2. Select the **Register WinDriver** option from the **File** menu and insert the license string you received from Jungo. Click the **Activate License** button.
3. To register source code that you developed during the evaluation period, refer to the documentation of `WDC_DriverOpen()` [A.2.2].

When using the lower-level `WD_XXX` API instead of the `WDC_XXX` API [A.1] (which is used by default), refer to the documentation of `WD_License()` [A.7.9].

3.2.2 Windows CE WinDriver Installation Instructions

3.2.2.1 Installing WinDriver CE when Building New CE-based Platforms

The following instructions apply to platform developers who build Windows CE kernel images using Windows CE Platform Builder:

NOTE

We recommend that you read Microsoft's documentation and understand the Windows CE and device driver integration procedure before you perform the installation.

1. Run Microsoft **Platform Builder** and open your platform.
2. Select **Open Build Release Directory** from the **Build** menu.
3. Copy the WinDriver CE kernel file
 `\WinDriver\redist\TARGET_CPU\windrvr6.dll`
to the `%_FLATRELEASEDIR%` subdirectory on your development platform
(should be the current directory in the new command window).
4. Append the contents of the file
 `\WinDriver\samples\wince_install\PROJECT_WD.REG`
to the file **PROJECT.REG** in the `%_FLATRELEASEDIR%` subdirectory.
5. Append the contents of the file
 `\WinDriver\samples\wince_install\PROJECT_WD.BIB`
to the file **PROJECT.BIB** in the `%_FLATRELEASEDIR%` subdirectory.

This step is only necessary if you want the WinDriver CE kernel file
(**windrvr6.dll**) to be a permanent part of the Windows CE image (**NK.BIN**).
This would be the case if you were transferring the file to your target platform
using a floppy disk. If you prefer to have the file **windrvr6.dll** loaded on
demand via the CESH/PPSH services, you need not carry out this step until
you build a permanent kernel.
6. Select **Make Image** from the **Build** menu and name the new image **NK.BIN**.
7. Download your new kernel to the target platform and initialize it either by
 selecting **Download/Initialize** from the **Target** menu or by using a floppy disk.
8. Restart your target CE platform. The WinDriver CE kernel will automatically
 load.
9. Compile and run the sample programs to make sure that WinDriver CE is
 loaded and is functioning correctly. (See Section 3.4, which describes how to
 check your installation.)

3.2.2.2 Installing WinDriver CE when Developing Applications for CE Computers

The following instructions apply to driver developers who do not build the Windows CE kernel, but only download their drivers, built using Microsoft eMbedded Visual C++, to a ready-made Windows CE platform:

1. Insert the WinDriver CD into your Windows host CD drive.
2. Exit from the auto installation.
3. Double click the **Cd_setup.exe** file found in the **\Wince** directory on the CD. This will copy all needed WinDriver files to your host development platform.
4. Copy the WinDriver CE kernel file
\WinDriver\redist\TARGET_CPU\windrvr6.dll
to the **\WINDOWS** subdirectory of your target CE computer.
5. Use the Windows CE Remote Registry Editor tool (**cregedt.exe**) or the Pocket Registry Editor (**pregedt.exe**) on your target CE computer to modify your registry so that the WinDriver CE kernel is loaded appropriately. The file **\WinDriver\samples\wince_install\PROJECT_WD.REG** contains the appropriate changes to be made.
6. Restart your target CE computer. The WinDriver CE kernel will automatically load. You will have to do a warm reset rather than just suspend/resume (use the reset or power button on your target CE computer).
7. Compile and run the sample programs (see Section 3.4, which describes how to check your installation) to make sure that WinDriver CE is loaded and is functioning correctly.

3.2.2.3 Windows CE Installation Note

The WinDriver installation on the host Windows 2000/XP/Server 2003 PC defines a **WD_BASEDIR** environment variable, which is set to point to the location of your WinDriver directory, as selected during the installation. This variable is used during the DriverWizard [4] code generation – it determines the default directory for saving your generated code and is used in the include paths of the generated project/make files.

Therefore, if you decide to change the name and/or location of your host WinDriver directory after the installation, you should also edit the value of the **WD_BASEDIR** environment variable and set it to point to the location of your new WinDriver directory. You can edit the value of **WD_BASEDIR** by following these steps:

1. Open the **System Properties** dialog: **Start | System | Control Panel | System**.

2. In the **Advanced** tab, click the **Environment Variables** button.
3. In the **System variables** box, select the `WD_BASEDIR` variable and click the **Edit ...** button or double-click the mouse on the variable.
4. In the **Edit System Variable** dialog, replace the **Variable Value** with the full path to your new WinDriver directory, then click **OK**, and click **OK** again from the **System Properties** dialog.

Note that if you install the WinDriver Windows 98/Me/NT/2000/XP/Server 2003 tool-kit on the same host PC, the installation will override the value of the `WD_BASEDIR` variable from the Windows CE installation.

3.2.3 Linux WinDriver Installation Instructions

3.2.3.1 Preparing the System for Installation

In Linux, kernel modules must be compiled with the same header files that the kernel itself was compiled with. Since WinDriver installs the kernel module **windrvr6.o/ko**, it must compile with the header files of the Linux kernel during the installation process.

Therefore, before you install WinDriver for Linux, verify that the Linux source code and the file **versions.h** are installed on your machine:

Install the Linux kernel source code:

- If you have yet to install Linux, install it, including the kernel source code, by following the instructions for your Linux distribution.
- If Linux is already installed on your machine, check whether the Linux source code was installed. You can do this by looking for 'linux' in the `/usr/src` directory. If the source code is not installed, either install it, or reinstall Linux with the source code, by following the instructions for your Linux distribution.

Install version.h:

- The file **version.h** is created when you first compile the Linux kernel source code. Some distributions provide a compiled kernel without the file **version.h**. Look under `/usr/src/linux/include/linux/` to see if you have this file. If you do not, please follow these steps:
 1. Type:
\$ **make xconfig**
 2. Save the configuration by choosing **Save and Exit**.
 3. Type:
\$ **make dep**

In order to run GUI WinDriver applications (e.g. DriverWizard [4] ; Debug Monitor [6.2]) you must also have version 5.0 of the **libstdc++** library – **libstdc++-so.5**. If you do not have this file, install it from the relevant RPM in your Linux distribution (e.g. **compat-libstdc++**).

Before proceeding with the installation, you must also make sure that you have a 'linux' symbolic link. If you do not, please create one by typing:

```
/usr/src$ ln -s <target kernel>/ linux
```

For example, for the Linux 2.4 kernel type:

```
/usr/src$ ln -s linux-2.4/ linux
```

3.2.3.2 Installation

1. Insert the WinDriver CD into your Linux machine's CD drive or copy the downloaded file to your preferred directory.
2. Change directory to your preferred installation directory, for example to your home directory:

```
$ cd ~
```

3. Extract the file **WDxxxLN.tgz** (where 'xxx' is the version number):

```
$ tar xvfz /<file location>/WDxxxLN.tgz
```

For example:

- From a CD:

```
$ tar xvfz /mnt/cdrom/LINUX/WDxxxLN.tgz
```

- From a downloaded file:

```
$ tar xvfz /home/username/WDxxxLN.tgz
```

4. Change directory to your WinDriver **redist/** directory (the tar automatically creates a **WinDriver/** directory):

```
$ cd <path to your WinDriver directory>/redist/
```

5. Install WinDriver:

```
(a) <WinDriver directory>/redist/$ ./configure
```

NOTE

The **configure** script creates a **makefile** based on your specific running kernel. You may run the **configure** script based on another kernel source you have installed, by adding the flag **--with-kernel-source=<path>** to the configure script. The <path> is the full path to the kernel source directory, e.g. **/usr/src/linux**.

```
(b) <WinDriver directory>/redist/$ make
```

- (c) Become super user:
`<WinDriver directory>/redist/$ su`
- (d) Install the driver:
`<WinDriver directory>/redist/# make install`
- 6. Create a symbolic link so that you can easily launch the DriverWizard GUI:
`$ ln -s <full path to WinDriver>/wizard/wdwizard/
usr/bin/wdwizard`
- 7. Change the read and execute permissions on the file **wdwizard** so that ordinary users can access this program.
- 8. Change the user and group IDs and give read/write permissions to the device file **/dev/windrvr6** depending on how you wish to allow users to access hardware through the device.

 If you are using a Linux 2.6.x kernel that has the **udev** file system, change the permissions by modifying your **/etc/udev/permissions.d/50-udev.permissions** file. For example, add the following line to provide read and write permissions:
`windrvr6:root:root:0666`

 Otherwise, use the **chmod** command, for example:
`chmod /dev/windrvr6 666`
- 9. Define a new **WD_BASEDIR** environment variable and set it to point to the location of your WinDriver directory, as selected during the installation. This variable is used in the make and source files of the WinDriver samples and generated DriverWizard [4] code and is also used to determine the default directory for saving your generated DriverWizard project. If you do not define this variable you will be instructed to do so when attempting to build the sample/generated code using the WinDriver makefiles.
 NOTE: If you decide to change the name and/or location of your WinDriver directory after the installation, you should also edit the value of the **WD_BASEDIR** environment variable and set it to point to the location of your new WinDriver directory.
- 10. You can now start using WinDriver to access your hardware and generate your driver code!

TIP

To avoid the need to reload the driver module (**windrvr6.o/.ko**) each time you restart your system, add the following line to your Linux **/etc/rc.d/rc.local** file:
/sbin/modprobe windrvr6

The following steps are for registered users only

In order to register your copy of WinDriver with the license you received from Jungo, follow the steps below:

1. Activate the DriverWizard GUI:
`<path to WinDriver>/wizard/wdwizard`
2. Select the **Register WinDriver** option from the **File** menu and insert the license string you received from Jungo.
3. Click the **Activate License** button.
4. To register source code that you developed during the evaluation period, refer to the documentation of `WDC_DriverOpen()` [A.2.2].

When using the lower-level `WD_xxx` API instead of the `WDC_xxx` API [A.1] (which is used by default), refer to the documentation of `WD_License()` [A.7.9].

Restricting Hardware Access on Linux**CAUTION!**

Since `/dev/windrvr6` gives direct hardware access to user programs, it may compromise kernel stability on multi-user Linux systems. Please restrict access to the DriverWizard and the device file `/dev/windrvr6` to trusted users.

For security reasons the WinDriver installation script does not automatically perform the steps of changing the permissions on `/dev/windrvr6` and the DriverWizard executable (`wdwizard`).

3.2.4 Solaris WinDriver Installation Instructions

Installation of WinDriver should be performed by the system administrator logged in as root, or with root privileges, since the WinDriver installation process includes installation of the kernel module `windrvr6`.

1. Insert your CD into your Solaris machine CD drive or copy the downloaded file to your preferred directory.
2. Change directory to your preferred installation directory (your home directory, for example):
\$ `cd ~`
3. Copy the file `WDxxxSLS.tgz` to the current directory (where 'xxx' stands for the version number – for example 700):
\$ `cp /home/username /WDxxxSLS.tgz /`

4. Extract the file:

```
$ gunzip -c WDxxxSLS.tgz | tar xvf -
```

5. Change directory to WinDriver.

6. Install WinDriver using the `/WinDriver/install_windrvr` installation script:

```
~/WinDriver# ./install_windrvr
```

To use WinDriver to handle **PCI** devices, specify the vendor and device IDs of your PCI devices in the installation command (where **<vid>** represents the device's vendor ID and **<did>** represents the device's device ID):

```
~/WinDriver# ./install_windrvr <vid>,<did> [<vid>,<did>
...]
```

For example, to use WinDriver to handle PLX 9030 and 9054 devices, run:

```
~/WinDriver# ./install_windrvr 10b5,9030 10b5,9054
```

7. Install the **libgcc** package, available for download from

<http://www.sunfreeware.com/>.

8. Add an environment variable:

- For SPARC 32-bit and x86 platforms:
LD_LIBRARY_PATH=/usr/local/bin
- For SPARC 64-bit platforms:
LD_LIBRARY_PATH=/usr/local/lib:/usr/local/lib/sparcv9

The following three steps are optional:

1. Create a symbolic link so that you can easily launch the DriverWizard GUI:
~/WinDriver# **ln -s ~/WinDriver/wizard/wdwizard**
/usr/bin/wdwizard
2. Change the read and execute permissions on the file **wdwizard** so that ordinary users can access this program.
3. Change the user and group IDs and give read/write permissions to the device file **/dev/windrvr6** depending on how you wish to allow users to access hardware through the device.
4. You can now start using WinDriver to access your hardware and generate your driver code!

The following steps are for registered users only:

In order to register your copy of WinDriver with the license you have received from Jungo, please follow the steps below:

1. Activate the DriverWizard GUI:

```
~/WinDriver/wizard$ ./wdwizard
```

2. Select the **Register WinDriver** option from the **File** menu and insert the license string you received from Jungo.
3. Click the **Activate License** button.
4. To register source code that you developed during the evaluation period, refer to the documentation of `WDC_DriverOpen()` [A.2.2].

When using the lower-level `WD_xxx` API instead of the `WDC_xxx` API [A.1] (which is used by default), refer to the documentation of `WD_License()` [A.7.9].

3.2.4.1 Restricting Hardware Access on Solaris

CAUTION!

Since `/dev/windrvr6` gives direct hardware access to user programs, it may compromise kernel stability on multi-user Solaris systems. Please restrict access to DriverWizard and the device file `/dev/windrvr6` to trusted users.

For security reasons the WinDriver installation script does not automatically perform the steps of changing the permissions on `/dev/windrvr6` and the DriverWizard executable (`wdwizard`).

3.2.5 VxWorks DriverBuilder Installation Instructions

The following describes the installation of DriverBuilder for VxWorks. DriverBuilder development environment works with Tornado 2 for Windows only (on x86 platform). Drivers generated using version 5.x and above of DriverBuilder will run on Intel x86 BSPs (pc486, pcPentium and pcPentiumPro), PPC 821/860 with MBX821/860 and PPC 750 (IBM PPC 604) with MCP750. For an up-to-date list see: <http://www.jungo.com/db-vxworks.html#platforms>.

Installation:

1. Download DriverBuilder for VxWorks.
2. Change drive to the preferred root drive for DriverBuilder. For example:
`> c:\`
3. Unpack the file you downloaded:
`> unzip -d DBxxxVX.zip c:\` (Here 'xxx' stands for the version number, e.g., 500.)

NOTE

The extraction creates a directory called DriverBuilder and then places all of the DriverBuilder installation files in it. If working with a version prior to 5.00, you will have to create a directory for DriverBuilder manually, and then perform the extraction. For example:

```
> c:\cd_vxworks and unpack the file to it:  
> unzip -d DBxxxVX.zip c:\db_vxworks
```

NOTE

In WinDriver, samples for VxWorks have the .out extension, e.g., **pci_diag.out**. To invoke these programs, use the WindShell to load them, and execute the routine `xxx_main()`. For example:

wddebug.out : `wddebug_main`

pci_diag.out : `pci_diag_main`

TIP

DriverBuilder is based on Jungo's WinDriver product line. You can save time by downloading the Windows version of WinDriver and using its graphical development environment for fast hardware validation and automatic code generation. If you choose to do so, follow these steps:

1. Download and install DriverBuilder for VxWorks.
2. Download and install WinDriver for Windows. *Don't skip this part.*
3. Create a shortcut on your desktop to DriverWizard (**C:\WinDriver\wizard\wdwizard.exe**) so that you can easily launch and develop your driver using the GUI DriverWizard.

3.3 Upgrading Your Installation

To upgrade to a new version of WinDriver on Windows, follow the steps outlined in Section 3.2.1, which illustrates the process of installing WinDriver for Windows 98/Me/NT/2000/XP/Server 2003. You can either choose to overwrite the existing installation or install to a separate directory.

After installation, start DriverWizard and enter the new license string, if you have received one. This completes the upgrade of WinDriver.

To upgrade your source code, pass the new license string as a parameter to `WDC_DriverOpen()` [A.2.2] (or to `WD_License()` [A.7.9] when using the low-level

WD_xxx API instead of the WDC_xxx API [A.1]).

The procedure for upgrading your installation on other operating systems is the same as the one described above. Please check the respective installation sections for installation details.

3.4 Checking Your Installation

3.4.1 On Your Windows, Linux and Solaris Machines

1. Start DriverWizard:
On Windows, by choosing **Programs | WinDriver | DriverWizard** from the **Start** menu, or using the shortcut that is automatically created on your Desktop. A third option for activating the DriverWizard on Windows is by running **wdwizard.exe** from a command prompt under the **wizard** sub-directory.
On Linux and Solaris you can access the wizard application via the file manager under the **wizard** sub-directory, or run the wizard application via a shell.
2. Make sure that your WinDriver license is installed (see Section 3.2, which explains how to install WinDriver). If you are an evaluation version user, you do not need to install a license.
3. For PCI cards – Insert your card into the PCI bus, and verify that DriverWizard detects it.
4. For ISA cards – Insert your card into the ISA bus, configure DriverWizard with your card's resources and try to read/write to the card using DriverWizard. (Not relevant for Solaris)

3.4.2 On Your Windows CE Machine

1. Start DriverWizard on your Windows host machine by choosing **Programs | WinDriver | DriverWizard** from the **Start Menu**.
2. Make sure that your WinDriver license is installed. If you are an evaluation version user, you do not need to install a license.
3. For Plug and Play devices (such as PCI, PCMCIA or CardBus) – Insert your device into the relevant bus slot, and verify that DriverWizard detects it.
4. For ISA cards – Insert your card into the ISA bus, configure DriverWizard with your card's resources and try to read/write to the card using DriverWizard.

5. Activate Visual C++ for CE.
6. Load one of the WinDriver samples, e.g.,
 \WinDriver\samples\speaker\speaker.dsw.
7. Set the target platform to x86em in the Visual C++ WCE configuration toolbar.
8. Compile and run the speaker sample. The Windows host machine's speaker should be activated from within the CE emulation environment.

NOTE

ISAPnP is not supported under Windows CE.

3.4.3 On VxWorks

1. In x86 only:
 Make sure MMU is set to basic support (**hardware/memory/MMU/MMU Mode**).
2. Load DriverBuilder, download the object file:
 (**DriverBuilder \redist\eval\intelx86\PENTIUM\windrvr6.o**).
3. Initialize DriverBuilder from the WindShell:

```
=> drvInit()  
  
function returned (return value = 0)  
  
=>
```
4. Run a sample driver. Load
 C:\DriverBuilder\samples\pci_diag\PENTIUM\pci_diag.out from the WindShell:

```
=> pci_diag_main()
```
5. Scan the PCI bus, open cards and access them.

3.5 Uninstalling WinDriver

This section will help you to uninstall either the evaluation or registered version of WinDriver.

3.5.1 On Windows 98/Me/NT/2000/XP/Server 2003

NOTES

- For **Windows 98/Me**, replace references to **wdreg** below with **wdreg16**.
- For **Windows 2000/XP/Server 2003**, you can also use the **wdreg_gui.exe** utility instead of **wdreg.exe**.
- **wdreg.exe**, **wdreg_gui.exe** and **wdreg16.exe** are found under the **WinDriver\util** directory (see Chapter 13 for details regarding these utilities).

1. Close any open WinDriver applications, including DriverWizard, the Debug Monitor (**wddebug_gui.exe**) and user-specific applications.
2. If you created a Kernel PlugIn driver:

- If your Kernel PlugIn driver is currently installed, uninstall it using the **wdreg** utility:

```
wdreg -name <Kernel PlugIn name> uninstall
```

NOTE

The Kernel PlugIn driver name should be specified without the *.sys extension.

- Erase your Kernel PlugIn driver from the **%windir%\system32\drivers** directory.
3. On Plug-and-Play Windows systems (**Windows 98/Me/2000/XP/Server 2003**): Uninstall any Plug-and-Play devices (USB/PCI/PCMCIA) that have been registered with WinDriver via an INF file:
 - On **Windows 2000/XP/Server 2003**: Uninstall the device using the **wdreg** utility:

```
wdreg -inf <path to the device-INF file> uninstall
```
 - On **Windows 98/Me**: Uninstall (Remove) the device manually from the Device Manager.

- Verify that no INF files that register your device(s) with WinDriver's kernel module (**windrvr6.sys**) are found in the **%windir%\inf** directory and/or **%windir%\inf\other** directory (Windows 98/Me).

4. Uninstall WinDriver:

- **On the development PC**, on which you installed the WinDriver toolkit: Run **Start | WinDriver | Uninstall**, **OR** run the **uninstall.exe** utility from the **WinDriver** installation directory.

The uninstall will stop and unload the WinDriver kernel module (**windrvr6.sys**); delete the copy of the **windrvr6.inf** file from the **%windir%\inf** directory (on Windows 2000/XP/Server 2003) or **%windir%\inf\other** directory (on Windows 98/Me); delete WinDriver from Windows' **Start** menu; delete the **WinDriver** installation directory (except for files that you added to this directory); and delete the short-cut icons to the DriverWizard and Debug Monitor utilities from the Desktop.

- **On a target PC**, on which you installed the WinDriver kernel module (**windrvr6.sys**), but not the entire WinDriver toolkit:

Use the **wdreg** utility to stop and unload the driver:

- On **Windows 98/Me/2000/XP/Server 2003** run:

```
wdreg -inf <path to windrvr6.inf> uninstall
```

NOTE

When running this command, **windrvr6.sys** should reside in the same directory as **windrvr6.inf**.

- On **Windows NT 4.0** run:

```
wdreg uninstall
```

(On the development PC, the relevant **wdreg** uninstall command is executed for you by the uninstall utility.)

NOTES

- If there are open handles to WinDriver when attempting to uninstall it (either using the **uninstall** utility or by running the **wdreg** uninstall command directly) – for example if there is an open WinDriver application or a connected Plug-and-Play device that has been registered to work with WinDriver via an INF file (on Windows 98/Me/2000/XP/Server 2003) – an appropriate warning message will be displayed. The message will provide you with the option to either close the open application(s) / uninstall/disconnect the relevant device(s), and **Retry** to uninstall the driver; or **Cancel** the uninstall of the driver, in which case the **windrvr6.sys** kernel driver will not be uninstalled. This ensures that you do not uninstall the WinDriver kernel module (**windrvr6.sys**) as long as it is being used.
- You can check if the WinDriver kernel module is loaded by running the Debug Monitor utility (**WinDriver\util\wddebug_gui.exe**). When the driver is loaded the Debug Monitor log displays driver and OS information; otherwise it displays a relevant error message. On the development PC the uninstall command will delete this utility, therefore in order to use it after you execute the uninstallation, create a copy of **wddebug_gui.exe** before performing the uninstall procedure.

5. If **windrvr6.sys** was successfully unloaded, erase the following files (if they exist):

- **%windir%\system32\drivers\windrvr6.sys**
- **%windir%\inf\windrvr6.inf** (Windows 2000/XP/Server 2003)
- **%windir%\inf\Jungowindrvr6.inf** (Windows 98/Me)
- **%windir%\system32\wd_utils.dll**
- **%windir%\system32\wdnetlib.dll**

6. Reboot the computer.

3.5.2 On Linux

NOTE

You must be logged in as root to perform the uninstall procedure.

1. Verify that the WinDriver module is not being used by another program:
 - View a list of modules and the programs using each of them:
`/# /sbin/lsmmod`
 - Close any applications that are using the WinDriver module.
 - Unload any modules that are using the WinDriver module:
`/sbin# rmmmod`
2. Unload the WinDriver module:
`/sbin# rmmmod windrvr6`
3. If you are not using a Linux 2.6.x kernel that supports the **udev** file system, remove the old device node in the **/dev** directory:
`/# rm -rf /dev/windrvr6`
4. If you created a Kernel PlugIn, remove it as well.
5. Remove the file **.windriver.rc** from the **/etc** directory:
`/# rm -rf /etc/.windriver.rc`
6. Remove the file **.windriver.rc** from **\$HOME**:
`/# rm -rf $HOME/.windriver.rc`
7. If you created a symbolic link to DriverWizard, delete the link using the command:
`/# rm -f /usr/bin/wdwizard`
8. Delete the WinDriver installation directory using the command:
`/# rm -rf ~/WinDriver`

3.5.3 On Solaris

NOTE

You must be logged in as root to perform the uninstall procedure.

1. Make sure no programs are using WinDriver.
2. If you created a Kernel PlugIn, remove it by following these steps:
 - (a) # **/usr/sbin/rem_drv kpname**
 - (b) On 64-bit platforms (64-bit SPARC):
rm /kernel/drv/sparcv9/kpname

On 32-bit platforms (32-bit x86/SPARC):
rm /kernel/drv/kpname
 - (c) # **rm /kernel/drv/kpname.conf**
3. Run the following uninstallation script:
~/WinDriver# **./remove_windrvr**
4. Run the following command:
 - On 64-bit platforms (64-bit SPARC):
rm -rf /kernel/drv/sparcv9/windrvr6
rm /kernel/drv/windrvr6.conf
 - On 32-bit platforms (32-bit x86/SPARC):
rm -rf /kernel/drv/windrvr6
rm /kernel/drv/windrvr6.conf
5. Remove the **.windriver.rc** file in the **/etc** directory. To do this, run the following command:
rm -rf /etc/.windriver.rc
6. Remove the **.windriver.rc** file in the **\$HOME** directory. To do this, run the following command:
rm -rf \$HOME/.windriver.rc
7. If you created a symbolic link to DriverWizard, delete the link:
rm -f /usr/bin/wdwizard
8. Delete the WinDriver installation directory, after changing the directory to the one above WinDriver:
rm -rf ~/WinDriver

3.5.4 On VxWorks

1. Delete the DriverBuilder installation directory (**C:\DriverBuilder**, for example) using Windows Explorer.
2. If you created any shortcuts to DriverWizard on your desktop, delete them.

Chapter 4

Using DriverWizard

This chapter describes WinDriver DriverWizard's hardware diagnostics and driver code generation capabilities.

NOTE

CardBus devices are handled via WinDriver's PCI API, therefore any references to **PCI** in this chapter also include **CardBus**.

4.1 An Overview

DriverWizard (included in the WinDriver toolkit) is a GUI-based diagnostics and driver generation tool that allows you to write to and read from the hardware, before writing a single line of code. The hardware is diagnosed through a Graphical User Interface—memory ranges are read, registers are toggled and interrupts are checked. Once the card is operating to your satisfaction, DriverWizard creates the skeletal driver source code, with functions to access all your hardware resources.

If you are developing a driver for a device that is based on one of the enhanced-support PCI chipsets (PLX 9030, 9050, 9052, 9054, 9056, 9080, 9656, Marvell gt64, Altera, Xilinx VirtexII, QuickLogic PBC/QuickPCI, AMCC 5933), we recommend you read Chapter 7, which explains WinDriver's enhanced support for specific chipsets, before starting your driver development.

DriverWizard can be used to diagnose your hardware and can generate an INF file for hardware running under Windows 98/Me/2000/XP/Server 2003 (an INF file should not be generated for hardware running under Windows NT). Avoid using DriverWizard to generate code for a card based on one of the supported PCI chipsets

[7], as DriverWizard generates generic code which will have to be modified according to the specific functionality of the card in question. Preferably, use the complete source code libraries and sample applications (supplied in the package) tailored to the various PCI chipsets.

DriverWizard is an excellent tool for two major phases in your HW/Driver development:

Hardware diagnostics: After the hardware has been built, insert the hardware into the appropriate bus slot on your machine, and use DriverWizard to verify that the hardware is performing as expected.

Code generation: Once you are ready to build your code, let DriverWizard generate your driver code for you.

The code generated by DriverWizard is composed of the following elements:

Library functions for accessing each element of your device's resources (memory ranges, I/O ranges, registers and interrupts).

A 32-bit diagnostics program in console mode with which you can diagnose your device. This application utilizes the special library functions described above. Use this diagnostics program as your skeletal device driver.

A project workspace/solution that you can use to automatically load all of the project information and files into your development environment. For Linux and Solaris, DriverWizard generates the required makefile.

4.2 DriverWizard Walkthrough

To use DriverWizard:

1. **Attach your hardware to the computer:**

Attach the card to the appropriate bus slot on your computer. **OR**
You have the option of using DriverWizard to generate code for a PCI device without having the actual device installed. When selecting this option, DriverWizard will generate code for your *virtual PCI device*.

NOTE

When selecting the *virtual PCI device* option, the DriverWizard allows you to define the device's resources. By specifying the IO/Memory ranges, you may further define run-time registers (the offsets are relative to BARs). In addition, the IRQ must be specified if you want to generate code that acknowledges interrupts via run-time registers. Note, that the IRQ number and the size of the IO/Memory ranges are irrelevant, since these will be automatically detected by DriverWizard when you install a physical device.

2. Run DriverWizard and select your device:

- (a) Click **Start | Programs | WinDriver | DriverWizard** or double click the DriverWizard icon on your desktop (on Windows), or run the **wdwizard** utility from the **/WinDriver/wizard/** directory.
- (b) Click **Next** in the **Choose Your Project** dialog box.
- (c) Select your **Plug and Play card** from the list of devices detected by DriverWizard.

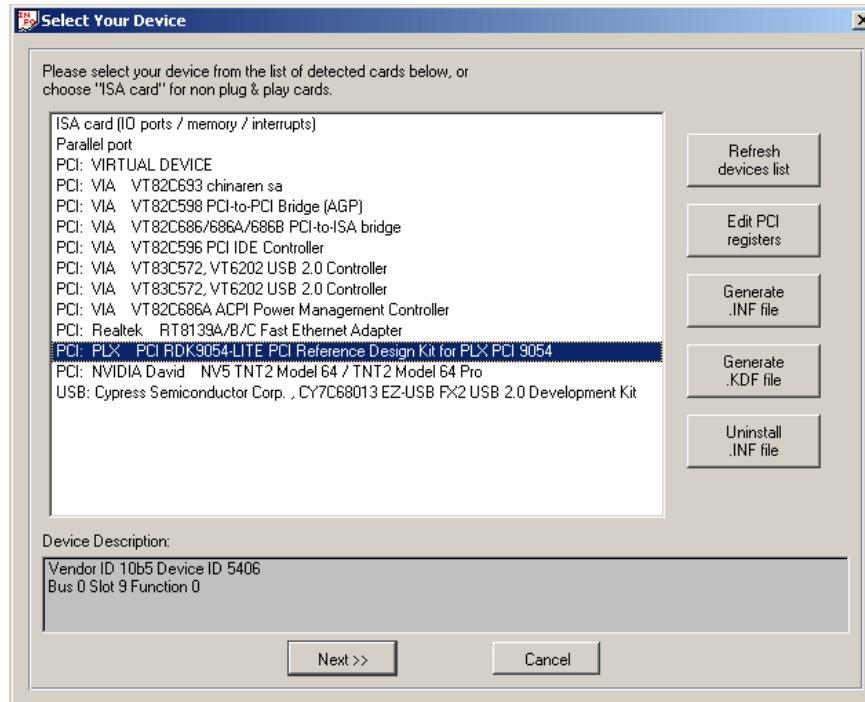


Figure 4.1: Select Your Plug and Play Device

For non-Plug and Play cards, select **ISA**. To generate code for a PCI device that is not currently attached to the computer, select **PCI: VIRTUAL DEVICE**.

3. Generate an INF file for DriverWizard:

Whenever developing a driver for a Plug and Play Windows operating system (i.e., Windows 98/Me/2000/XP/Server 2003) you are required to install an INF file for your device. This file will register your Plug and Play device to work with the **windrvr6.sys** driver. The file generated by the DriverWizard in this step should later be distributed to your customers using Windows 98/Me/2000/XP/Server 2003, and installed on their PCs.

The INF file you generate here is also designed to enable DriverWizard to diagnose your device (for example, when no driver is installed for your PCI/PCMCIA device). As explained earlier, this is required only when using WinDriver to support a Plug and Play device (PCI/PCMCIA) on a Plug and Play system (Windows 98/Me/2000/XP/Server 2003). Additional information concerning the need for an INF file is explained in Section 14.4.1.

If you do not need to generate an INF file, skip this step and proceed to the next one.

To generate the INF file with the DriverWizard, follow the steps below:

- (a) In the **Select Your Device** screen, click the **Generate .INF file** button or click **Next**.
- (b) DriverWizard will display information detected for your device – Vendor ID, Device ID, Device Class, manufacturer name and device name – and allow you to modify the manufacturer and device names and the device class information.
- (c) When you are done, click **Next** and choose the directory in which you wish to store the generated INF file. DriverWizard will then automatically generate the INF file for you.

On **Windows 2000/XP/Server 2003** you can choose to automatically install the INF file from the DriverWizard by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation dialog.

On **Windows 98/Me** you must install the INF file manually, using Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained in Section 14.4.

If the automatic INF file installation on Windows 2000/XP/Server 2003 fails, DriverWizard will notify you and provide manual installation instructions for this OS as well.

- (d) When the INF file installation completes, select and open your device from the list in the **Select Your Device** screen.

4. Uninstall the INF file of your device:

You can use the **Uninstall** option to uninstall the INF file of your Plug and Play device (PCI/PCMCIA). Once you uninstall the INF file, the device will

Enter Information for INF File

Please fill in the information below for your device.

This information will be incorporated into the INF file, which WinDriver will generate for your device.

The information you specify will appear in the Device Manager after the installation of the INF file.

Vendor ID: Device ID:

Manufacturer name:

Device name:

Device Class:

WinDriver's unique Class.

Use this option for a non-standard type of device. WinDriver will set a new Class type for your device.

☐ Automatically Install the INF file.
Note: This will replace any existing driver you may have for your device.

Figure 4.2: DriverWizard INF File Information

no longer be registered to work with the **windrvr6.sys**, and the INF file will be deleted from the Windows root directory.

If you do not need to uninstall an INF file, skip this step and proceed to the next one.

- (a) In the **Select Your Device** screen, click the **Uninstall .INF file** button.
- (b) Select the INF file to be removed.

5. Diagnose your device:

Before writing your device driver, it is important to make sure your hardware is working as expected. Use DriverWizard to diagnose your hardware. All of your activity will be logged in the DriverWizard log so that you may later analyze your tests:

- (a) Define and test your device's I/O and memory ranges, registers and interrupts:

- DriverWizard will automatically detect your Plug-and-Play hardware's resources (I/O ranges, memory ranges and interrupts). You can define the registers manually.
- For non-Plug-and-Play hardware, define your hardware's resources manually.
- Read and write to the I/O ports, memory space and your defined registers.

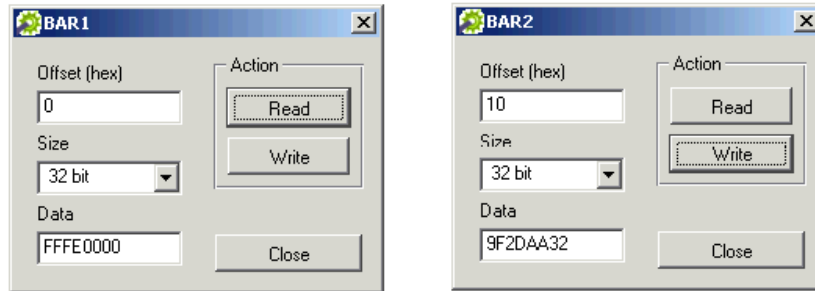


Figure 4.3: PCI Diagnostics Screen

NOTE

You have the option to check the **Auto Read** box from the **Register Information** window. The registers that are marked with the **Auto Read** option will automatically be read with any register read/write operation performed from the Wizard (the read results will be displayed in the wizard's Log window).

- 'Listen' to your hardware's interrupts.

NOTE

When accessing memory mapped ranges, be aware that **Linux PowerPC** uses big-endian for handling memory storage, as opposed to the PCI bus that uses little-endian. For more information regarding little/big-endian issues, refer to Section 9.4.

6. Generate the skeletal driver code:

- Select **Generate Code** from the **Build** menu, or click **Next** in the **Define and Test Resources for Your Device** dialog box.
- In the **Select Code Generation Options** dialog box that will appear, choose the code language and development environment(s) for the generated code and select **Next** to generate the code.

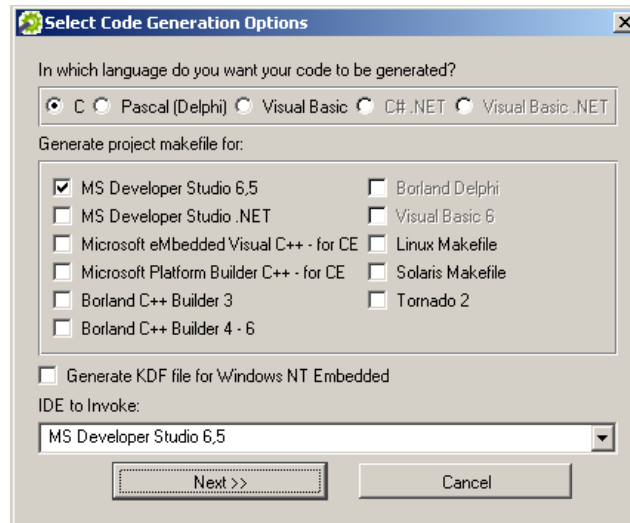


Figure 4.4: Code Generation Options

- (c) Click **Next** and indicate whether you wish to handle Plug and Play and power management events from within your driver code and whether you wish to generate Kernel PlugIn code.

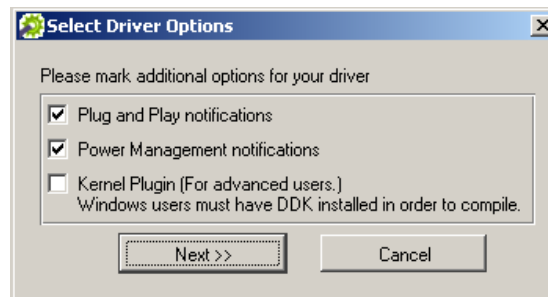


Figure 4.5: Notification Events

NOTE

In order to work with a Kernel PlugIn, you must have an appropriate Microsoft DDK installed on your computer before you generate Kernel PlugIn code.

- (d) Save your project (if required) and click **OK** to open your development

environment with the generated driver.

7. Compile and run the generated code:

- Use this code as a starting point for your device driver. Modify where needed to perform your driver's specific functionality.
- The source code DriverWizard creates can be compiled with any 32-bit compiler, and will run on all supported platforms (Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks) without modification.

4.3 DriverWizard Notes

4.3.1 Sharing a Resource

If you want more than one driver to share a single resource, you must define that resource as shared:

1. Select the resource.
2. Right click on the resource.
3. Select **Share** from the menu.

NOTE

New interrupts are set as **Shared** by default. If you wish to define an interrupt as unshared, follow Steps 1 and 2, and select **Unshared** in Step 3.

4.3.2 Disabling a Resource

During your diagnostics, you may wish to disable a resource so that DriverWizard will ignore it and not create code for it.

1. Select the resource.
2. Right click on the resource name.
3. Choose **Disable** from the menu.

4.3.3 Logging WinDriver API Calls

You have the option to log all the WinDriver API calls using the DriverWizard, with the API calls input and output parameters. You can select this option by selecting the **Log API calls** option from the **Tools** menu or by clicking on the **Log API calls** toolbar icon in the DriverWizard's opening window.

4.3.4 DriverWizard Logger

The wizard logger is the empty window that opens along with the **Device Resources** dialog box when you open a new project. The logger keeps track of all of the input and output during the diagnostics stage, so that you may analyze your device's physical performance at a later time. You can save the log for future reference. When saving the project, your log is saved as well. Each log is associated with one project.

4.3.5 Automatic Code Generation

After you have finished diagnosing your device and have ensured that it runs according to your specifications, you are ready to write your driver.

4.3.5.1 Generating the Code

Choose **Generate Code** from the **Build** menu. DriverWizard will generate the source code for your driver, and place it along with the project file (**xxx.wdp**, where "xxx" is the project name). The files are saved in a directory DriverWizard creates for every development environment and operating system selected in the **Generate Code** dialog box.

4.3.5.2 Generated PCI/PCMCIA/ISA Code

In the source code directory you now have a new **xxx_lib.h** file, which contains type definitions and functions declarations for the API created for you by the DriverWizard, and an **xxx_lib.c** source file, which contains the implementation of the generated device-specific API.

In addition, you will find an **xxx_diag.c** source file, which includes a `main()` function and implements a sample diagnostics application that utilizes the generated DriverWizard API to communicate with your device.

The code generated by DriverWizard is composed of the following elements and files, where **xxx** represents your DriverWizard project name:

- Library functions for accessing each element of your card's resources (memory ranges and I/O, registers and interrupts):

xxx_lib.c Here you can find the implementation of the hardware-specific API (declared in **xxx_lib.h**), using the WinDriver Card (WDC) API [\[A.1\]](#).

xxx_lib.h Header file that contains type definitions and function declarations for the API implemented in the **xxx_lib.c** source file.

You should include this file in your source code in order to use the API generated by the DriverWizard for your device.

- A diagnostics program that utilizes the generated DriverWizard API (declared in **xxx_lib.h**) to communicate with your device(s):

xxx_diag.c The source code of the generated diagnostics console application. Use this diagnostics program as your skeletal device driver.

- A list of all files created can be found at **xxx_files.txt**.

After creating your code, compile it with your favorite compiler, and see it work!

Change the function `main()` of the program so that the functionality fits your needs.

4.3.5.3 Compiling the Generated Code

For Windows 98, Me, NT, 2000, XP, CE and Server 2003 (Using MSDEV):

1. For Windows platforms, DriverWizard generates the project files (for MSDEV 5, 6 and 7 (.Net), Borland C/C++ Builder, Visual Basic and Delphi). After code generation, the chosen IDE (Integrated Development Environment) will be launched automatically. You can then immediately compile and run the generated code.

4.3.5.4 Visual Basic or Delphi Code Generation

This will generate Visual Basic or Delphi project and files, similar to the MSDEV projects described in above [4.3.5.2].

4.3.5.5 For Linux and Solaris:

1. DriverWizard creates a makefile for your project.
2. Compile the source code using the makefile generated by DriverWizard.
3. Use any compilation environment to build your code, preferably GCC.

4.3.5.6 For Other OSs or IDEs:

1. Create a new project in your IDE (Integrated development environment).
2. Include the source files created by DriverWizard in your project.
3. Compile and run the project.
4. The project contains a working example of the custom functions that DriverWizard created for you. Use this example to create the functionality you want.

Chapter 5

Developing a Driver

This chapter takes you through the WinDriver driver development cycle.

NOTE

If your device is based on one of the chipsets for which WinDriver provides enhanced support (PLX 9030, 9050, 9052, 9054, 9056, 9080, 9656, Marvell gt64, Altera, Xilinx VirtexII, QuickLogic PBC/QuickPCI, AMCC 5933), read the following overview and then skip straight to Chapter 7.

5.1 Using the DriverWizard to Build a Device Driver

- Use DriverWizard to diagnose your card: Read/write the I/O and memory ranges, view the PCI configuration registers information, define registers for your card and read/write the registers, and listen to interrupts.
- Use DriverWizard to generate skeletal code for your device in C, Delphi or Visual Basic. Refer to Chapter 4 for details about DriverWizard.
- If you are using one of the specific chipsets for which WinDriver offers enhanced support (PLX 9030, 9050, 9052, 9054, 9056, 9080, 9656, Marvell gt64, Altera, Xilinx VirtexII, QuickLogic PBC/QuickPCI, AMCC 5933), we recommend that you use the specific sample code provided for your chip as your skeletal driver code. For more details regarding WinDriver's enhanced support for specific chipsets, refer to Chapter 7.
- Use any 32-bit compiler (such as MSDEV, Visual C/C++, Borland Delphi, Borland C++, Visual Basic, GCC) to compile the skeletal driver you need.

- For Linux and Solaris, use any compilation environment, preferably GCC to build your code.
- That is all you need do to create your user-mode driver. If you discover that better performance is needed, please refer to Chapter 10 for details on performance improvement.

Please see Appendix A for a detailed description of WinDriver's PCI/ISA/PCMCIA API. To learn how to perform operations that DriverWizard cannot automate, refer to Chapter 9 of the manual.

5.2 Writing the Device Driver Without the DriverWizard

There may be times when you choose to write your driver directly without using DriverWizard. Sometimes you may be compelled to do so, for example, when working with VxWorks without using Windows as a host, since DriverBuilder does not provide the DriverWizard utility. In either case, proceed according to the steps outlined below, or choose a sample that most closely resembles what your driver should do, and modify it. For further information on VxWorks, please refer to Sections 3.2.5 and 3.4.3.

5.2.1 Include the Required WinDriver Files

1. Include the relevant WinDriver header files in your driver project (all header files are found under the **/WinDriver/include** directory).
All WinDriver projects require the **windrvr.h** header file.
When using the WDC_XXX API [A.1], include the **wdc_lib.h** and **wdc_defs.h** header files (these files already include **windrvr.h**).
Include any other header file that provides APIs that you wish to use from your code (e.g. files from the **/WinDriver/samples/shared/** directory, which provide convenient diagnostics functions.)
2. Include the relevant header files from your source code: For example, if your source file uses API defined in the **windrvr.h** file, add the following line to the code:

```
#include "windrvr.h"
```
3. Link your code with the **wd_utils** DLL/shared object from the **WinDriver/lib/** directory (**wd_utils.lib** / **wd_utils_borland.lib** (Borland C++ Builder) – for

Windows 98/Me/NT/2000/XP/Server 2003 and Windows CE ; **libwd_utils.so** – for Linux and Solaris), or otherwise include the relevant WinDriver source files from the **WinDriver/src/** directory.

When using the **wd_utils** DLL/shared object, you will need to distribute **WinDriver/redist/wd_utils.dll** (Windows 98/Me/NT/2000/XP/Server 2003 and Windows CE) / **WinDriver/lib/libwd_utils.so** (Linux and Solaris) with your driver – see Chapter 14.

4. Add any other WinDriver source files that implement API that you which to use in your code (e.g. files from the **/WinDriver/samples/shared/** directory.)

5.2.2 Write Your Code

This section outlines the calling sequence when using the WDC_xxx API [A.1].

1. Call **WDC_DriverOpen()** [A.2.2] to open a handle to WinDriver and the WDC library, compare the version of the loaded driver with that of your driver source files, and register your WinDriver license (for registered users).
2. For PCI/CardBus/PCMCIA devices, call **WDC_PciScanDevices()** [A.2.4] / **WDC_PcmciaScanDevices()** [A.2.5] to scan the PCI/PCMCIA bus and locate your device.
3. For PCI/CardBus/PCMCIA devices, call **WDC_PciGetDeviceInfo()** [A.2.6] / **WDC_PcmciaGetDeviceInfo()** [A.2.7] to retrieve the resources information for your selected device.
For ISA devices, define the resources yourself within a **WD_CARD** structure.
4. Call **WDC_PciDeviceOpen()** [A.2.8] / **WDC_PcmciaDeviceOpen()** [A.2.9] / **WDC_IsaDeviceOpen()** [A.2.10] (depending on your device) and pass to the function the device's resources information. These functions return a handle to the device, which you can later use to communicate with the device using the **WDC_xxx** API.
5. Communicate with the device using the **WDC_xxx** API (see description in Appendix [A]).
To enable interrupts, call **WDC_IntEnable()** [A.2.41].
To register to receive notifications for Plug and Play and power management events, call **WDC_EventRegister()** [A.2.44].
6. When you are done, call **WDC_IntDisable()** [A.2.42] to disable interrupt handling (if previously enabled), call **WDC_EventRegister()** [A.2.44] to unregister Plug and Play and power management event handling (if previously registered), and then call **WDC_PciDeviceClose()** [A.2.11] / **WDC_PcmciaDeviceClose()** [A.2.12] / **WDC_IsaDeviceClose()** [A.2.13] (depending on your device) in order to close the handle to the device.

7. Call `WDC_DriverClose()` [A.2.3] to close the handles to WinDriver and the WDC library.

5.3 Developing Your Driver on Windows CE Platforms

When developing your driver on Windows CE platforms, you must first register your device to work with WinDriver. This is similar to installing an INF file for your device when developing a driver for a Plug and Play Windows operating system (i.e., Windows 98, Me, 2000, XP or Server 2003). Refer to Section 14.4 for understanding the INF file.

The following registry example shows how to register your device with the PCI bus driver (can be added to your **platform.reg** file).

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PCI\Template\MyCard]
"Class"=dword:04
"SubClass"=dword:01
"ProgIF"=dword:00
"VendorID"=multi_sz:"1234","1234"
"DeviceID"=multi_sz:"1111","2222"
```

For more information, refer to MSDN Library, under *PCI Bus Driver Registry Settings* section.

5.4 Developing in Visual Basic and Delphi

The entire WinDriver API can be used when developing drivers in Visual Basic and Delphi.

5.4.1 Using DriverWizard

DriverWizard can be used to diagnose your hardware and verify that it is working properly before you start coding. You can then proceed to automatically generate source code with the wizard in a variety of languages, including Delphi and Visual Basic. For more information, refer to Chapter 4 and Section 5.4.4 below.

5.4.2 Samples

Samples for drivers written using the WinDriver API in Delphi or Visual Basic can be found in:

1. `\WinDriver\delphi\samples`
2. `\WinDriver\vb\samples`

Use these samples as a starting point for your own driver.

5.4.3 Kernel PlugIn

Delphi and Visual Basic cannot be used to create a Kernel PlugIn. Developers using WinDriver with Delphi or VB in user mode must use C when writing their Kernel PlugIn.

5.4.4 Creating your Driver

The method of development in Visual Basic is the same as the method in C using the automatic code generation feature of DriverWizard.

Your work process should be as follows:

- Use DriverWizard to easily diagnose your hardware.
- Verify that it is working properly.
- Generate your driver code.
- Integrate the driver into your application.
- You may find it useful to use the WinDriver samples to get to know the WinDriver API and as your skeletal driver code.

Chapter 6

Debugging Drivers

The following sections describe how to debug your hardware access application code.

6.1 User-Mode Debugging

- Since WinDriver is accessed from user mode, we recommend that you first debug your code using your standard debugging software.
- When the Debug Monitor [6.2] is activated, WinDriver's kernel module performs verification of the validity of the memory ranges when using `WD_Transfer()` [A.4.14], i.e. it verifies that the reading/writing from/to the memory is in the range that is defined for the card.
- Use DriverWizard to check values of memory and registers in the debugging process.

6.2 Debug Monitor

Debug Monitor is a powerful graphical- and console-mode tool for monitoring all activities handled by the WinDriver kernel (**windr6.sys/windr6.dll/windr6.o/.ko**). You can use this tool to monitor how each command sent to the kernel is executed.

Debug Monitor has two modes: graphical mode and console mode. The following sections explain how to operate Debug Monitor in both modes.

6.2.1 Using Debug Monitor in Graphical Mode

Applicable for Windows 98, Me, NT, 2000, XP, Server 2003, Linux and Solaris. You may also use Debug Monitor to debug your Windows CE driver code running on CE emulation on a Windows 2000/XP/Server 2003 platform. For Windows CE targets use Debug Monitor in console mode.

1. Run the Debug Monitor using one the following three ways:
 - The Debug Monitor is available as **wddebug_gui** in the **\WinDriver\util** directory.
 - The Debug Monitor can be launched from the **Tools** menu in DriverWizard.
 - In Windows, use **Start | Programs | WinDriver | Debug Monitor** to start Debug Monitor.

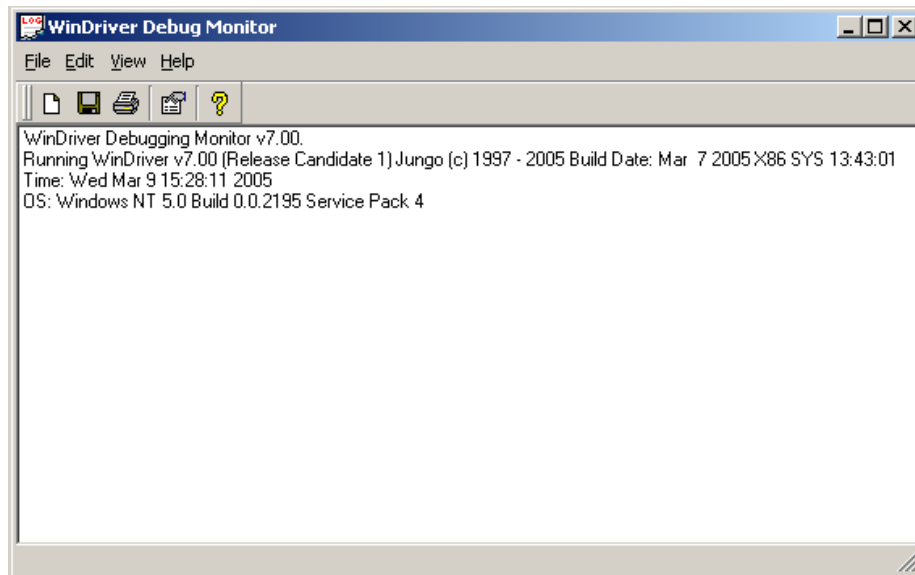


Figure 6.1: Start Debug Monitor

2. Activate and set the trace level using either the **View | Debug Options** menu or the **Change Status** button.

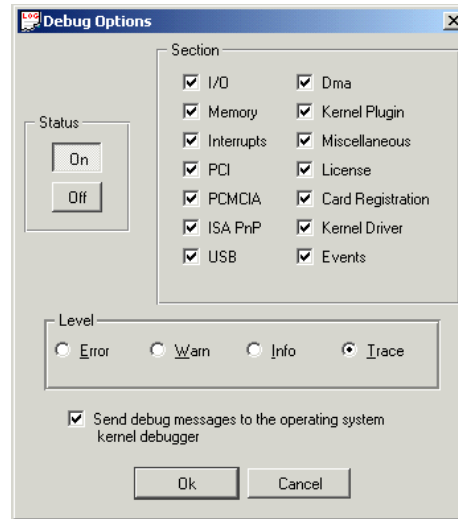


Figure 6.2: Set Trace Options

- **Status** – Set trace on or off.
- **Section** – Choose what part of the WinDriver API you would like to monitor. If you are developing a PCI card and experiencing problems with your interrupt handler, select the **Int** and **PCI** check boxes.

TIP

Choose carefully those sections that you would like to monitor. Checking more options than necessary could result in an overflow of information, making it harder for you to locate your problem.

- **Level** – Choose the level of messages you want to see for the resources defined.

Error is the lowest level of trace, resulting in minimum output to the screen.

Trace is the highest level of tracing, displaying every operation the WinDriver kernel performs.

- Select the **Send WinDriver Debug Messages To Kernel Debugger** check box if you want debugging messages to be sent to an external kernel debugger as well.

This option enables you to send to an external kernel debugger all the debug information that is received from WinDriver's kernel module (which calls `WD_DebugAdd()` [A.7.6] in your code).

Now run your application, reproduce the problem, and view the debug information in the external kernel debugger's log.

Windows users can use Microsoft's WinDbg tool, for example, which is freely supplied with Microsoft's Driver Development Kit (DDK) and from Microsoft's web site (Microsoft Debugging Tools page).

3. Once you have defined what you want to trace and on what level, click **OK** to close the **Modify Status** window.
4. Activate your program (step-by-step or in one run).
5. Watch the monitor screen for errors or any unexpected messages.

6.2.2 Using Debug Monitor in Console Mode

This tool is available in all supported operating systems. To use it, run:

```
\WinDriver\util> wddebug
```

with the appropriate switches.

For a list of switches that can be used with Debug Monitor in console mode, type:

```
\> wddebug
```

To see activity logged by the Debug Monitor, type:

```
\> wddebug dump.
```

6.2.2.1 Using Debug Monitor on Windows CE

On Windows CE, Debug Monitor is only available in console mode. You first need to start a Windows CE command window (**CMD.EXE**) on the Windows CE target computer and then run the program **WDDEBUG.EXE** inside this shell.

6.2.2.2 Using Debug Monitor on VxWorks

On VxWorks, Debug Monitor is only available in console mode. However, because of the special syntax of the Tornado WindShell, we show a sample session with Tornado II IDE below, where we first load the debug monitor, then set the options and then run it to capture information.

```
-> ld < wddebug.out
Loading wddebug.out |
value = 10893848 = 0xa63a18
-> wdddebug

-> wddebug_main "on", "trace", "all"
Debug level (4) TRACE, Debug sections (0xffffffff) ALL ,
Buffer size 16384
value = 0 = 0x0
-> wddebug_main "dump"
WDDEBUG v5.00 Debugging Monitor.
Running DriverBuilder V5.00 Jungo (c) 2001 evaluation copy
Time: THU JAN 01 01:06:56 2001
OS: VxWorks
Press CTRL-BREAK to exit
```

Please note the following:

- The Debug Monitor object binary module is called **wddebug.out**.
- The main program entry point is called `wddebug_main()`.
- The arguments are enclosed in double quotes and are separated by commas. This syntax is required by WindShell.

Chapter 7

Enhanced Support for Specific Chipsets

7.1 Overview

In addition to the standard WinDriver API and the DriverWizard code generation capabilities described in this manual, which support development of drivers for any PCI/ISA/PCMCIA/CardBus device, WinDriver offers enhanced support for specific PCI chipsets. The enhanced support includes custom API and sample diagnostics code, which are designed specifically for these chipsets.

WinDriver's enhanced support is currently available for the following chipsets: PLX 9030, 9050, 9052, 9054, 9056, 9080, 9656, Marvell gt64, Altera, Xilinx VirtexII, QuickLogic PBC/QuickPCI, AMCC 5933.

7.2 Developing a Driver Using the Enhanced Chipset Support

When developing a driver for a device based on one of the enhanced-support chipsets [7.1], you can use WinDriver's chipset-set specific support by following these steps:

1. Locate the sample diagnostics program for your device under the **/WinDriver/chip_vendor/chip_name** directory.

Most of the sample diagnostics programs are named **xxx_diag** and their source code is normally found under an **xxx_diag/** sub-directory.
The program's executable is found under a sub-directory for your target operating system (e.g. **WIN32** for Windows.)
2. Run the custom diagnostics program to diagnose your device and familiarize yourself with the options provided by the sample program.
3. Use the source code of the diagnostics program as your skeletal device driver and modify the code, as needed, to suit your specific development needs. When modifying the code, you can utilize the custom WinDriver API for your specific chip. The custom API is typically found under the **/WinDriver/chip_vendor/lib/** directory.
4. If the user-mode driver application that you created by following the steps above contains parts that require enhanced performance (e.g. an interrupt handler), you can move the relevant portions of your code to a Kernel PlugIn driver for optimal performance, as explained in Chapter 11.

Chapter 8

PCI Express

8.1 PCI Express Overview

The PCI Express (**PCIe**) bus architecture (formerly 3GIO or 3rd Generation I/O) was introduced by Intel, in partnership with other leading companies, including IBM, Dell, Compaq, HP and Microsoft, with the intention that it will become the prevailing standard for PC I/O in the years to come.

PCI-Express allows for larger bandwidth and higher scalability than the standard PCI 2.2 bus.

The standard PCI 2.2 bus is designed as a single parallel data bus through which all data is routed at a set rate. The bus shares the bandwidth between all connected devices, without the ability to prioritize between devices. The maximum bandwidth for this bus is 132MB/s, which has to be shared among all connected devices.

PCI Express is comprised of serial, point-to-point wired, individually clocked 'lanes', each lane consisting of two pairs of data lines that can carry data upstream and downstream simultaneously (full-duplex). The bus slots are connected to a switch that controls the data flow on the bus. A connection between a PCI Express device and a PCI Express switch is called a 'link'. Each link is composed of one or more lanes. A link composed of a single lane is called an x1 link; a link composed of two lanes is called an x2 link; etc. PCI Express supports x1, x2, x4, x8, x12, x16, and x32 link widths (lanes). The PCI Express architecture allows for a maximum bandwidth of approximately 500MB/s per lane. Therefore, the maximum potential bandwidth of this bus is 500MB/s for x1, 1,000MB/s for x2, 2,000MB/s for x4, 4,000MB/s for x8, 6,000MB/s for x12, and 8,000MB/s for x16. These values provide a significant improvement over the maximum 132MB/s bandwidth of the standard 32-bit PCI bus.

The increased bandwidth support makes PCI Express ideal for the growing number of devices that require high bandwidth, such as hard drive controllers, video streaming devices and networking cards.

The usage of a switch to control the data flow in the PCI Express bus, as explained above, provides an improvement over a shared PCI bus, because each device essentially has direct access to the bus, instead of multiple components having to share the bus. This allows each device to use its full bandwidth capabilities without having to compete for the maximum bandwidth offered by a single shared bus. Adding to this the lanes of traffic that each device has access to in the PCI Express bus, PCI Express truly allows for control of much more bandwidth than previous PCI technologies. In addition, this architecture enables devices to communicate with each other directly (peer-to-peer communication).

In addition, the PCI Express bus topology allows for centralized traffic-routing and resource-management, as opposed to the shared bus topology. This enables PCI Express to support quality of service (QoS): The PCI Express switch can prioritize packets, so that real-time streaming packets (i.e. a video stream or an audio stream) can take priority over packets that are not as time critical.

Another main advantage of the PCI Express is that it is cost-efficient to manufacture when compared to PCI and AGP slots or other new I/O bus solutions such as PCI-X.

PCI Express was designed to maintain complete hardware and software compatibility with the existing PCI bus and PCI devices, despite the different architecture of these two buses.

As part of the backward compatibility with the PCI 2.2 bus, legacy PCI 2.2 devices can be plugged into a PCI Express system via a PCI Express-to-PCI bridge, which translates PCI Express packets back into standard PCI 2.2 bus signals. This bridging can occur either on the motherboard or on an external card.

8.2 WinDriver for PCI Express

WinDriver fully supports backward compatibility with the standard PCI features on PCI Express boards. The wide support provided by WinDriver for the standard PCI bus – including a rich set of APIs, code samples and the graphical DriverWizard for hardware debugging and driver code generation – is also applicable to PCI Express devices, which by design are backward compatible with the legacy PCI bus.

You can also use WinDriver's PCI API to easily communicate with PCI devices connected to the PC via PCI Express-to-PCI bridges and switches (e.g. the PLX 8111/8114 bridges or the PLX 8532 switch, respectively).

In addition, on Windows and Linux WinDriver provides you with a set of APIs for easy access to the PCI Express extended configuration space – see the description of the `WDC_PciReadCfgXXX()` and `WDC_PciWriteCfgXXX()` functions in sections [A.2.24](#) – [A.2.31](#) of the manual, or the lower-level `WD_PciConfigDump()` function in section [A.4.4](#).

Additional enhanced support for the PCI Express bus, in the form of specific PCI Express WinDriver libraries and special features, are expected to be added in future versions of WinDriver.

Chapter 9

Advanced Issues

This chapter covers advanced driver development issues and contains guidelines for using WinDriver to perform tasks that cannot be fully automated by the DriverWizard.

Note that WinDriver's enhanced support for specific chipsets [7] includes custom APIs for performing hardware-specific tasks like DMA and interrupt handling, thus freeing developers of drivers for these chipsets from the need to implement the code for performing these tasks themselves.

9.1 Performing Direct Memory Access (DMA)

This section describes how to use WinDriver to implement bus-master Direct Memory Access (**DMA**) for devices capable of acting as bus masters. Such devices have a DMA controller, which the driver should program directly.

DMA is a capability provided by some computer bus architectures, including PCI, PCMCIA and CardBus, which allows data to be sent directly from an attached device to the memory on the host, freeing the CPU from involvement with the data transfer and thus improving the host's performance.

A DMA buffer can be allocated in two ways:

- **Contiguous Buffer:** A contiguous block of memory is allocated.
- **Scatter/Gather:** The allocated buffer can be fragmented in the physical memory and does not need to be allocated contiguously. The allocated physical memory blocks are mapped to a contiguous buffer in the calling process's

virtual address space, thus enabling easy access to the allocated physical memory blocks.

The programming of a device's DMA controller is hardware specific. Normally, you need to program your device with the *local address* (on your device), the *host address* (the physical memory address on your PC) and the *transfer count* (the size of the memory block to transfer), and then set the register that initiates the transfer.

WinDriver provides you with API for implementing both Contiguous Buffer DMA and Scatter/Gather DMA (if supported by the hardware) – see the description of `WDC_DMAContigBufLock()` [A.2.36], `WDC_DMASGBufLock()` [A.2.37] and `WDC_DMABufUnlock()` [A.2.38]. (The lower-level `WD_DMAxxx` API is described in sections A.4.16 – A.4.17, but we recommend using the convenient wrapper `WDC_xxx` API instead.)

This section includes code samples that demonstrate how to use WinDriver to implement Scatter/Gather and Contiguous Buffer DMA.

NOTES

- The sample routines demonstrate using either an interrupt mechanism or a polling mechanism to determine DMA completion.
- The sample routines allocate a DMA buffer and enable DMA interrupts (if polling is not used) and then free the buffer and disable the interrupts (if enabled) for each DMA transfer. However, when you implement your actual DMA code, you can allocate DMA buffer(s) once, at the beginning of your application, enable the DMA interrupts (if polling is not used), then perform DMA transfers repeatedly, using the same buffer(s), and disable the interrupts (if enabled) and free the buffer(s) only when your application no longer needs to perform DMA.

9.1.1 Scatter/Gather DMA

Following is a sample routine that uses WinDriver's WDC API [A.1] to allocate a Scatter/Gather DMA buffer and perform bus-master DMA transfers.

A more detailed example, which is specific to the enhanced support for PLX chipsets ([7]) can be found in the `/WinDriver/plx/lib/plx_lib.c` library file and `/WinDriver/plx/diag_lib/plx_diag_lib.c` diagnostics library file (which utilizes the `plx_lib.c` DMA API.)

A sample that uses the basic `WD_DMAxxx`

API for implementing Scatter/Gather DMA for the Marvell gt64 chip can be found in the `/WinDriver/marvell/gt64/lib/gt64_lib.c` library file.

9.1.1.1 Sample Scatter/Gather DMA Implementation

```

BOOL DMARoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMAChannel, DWORD dwDMABufSize,
    UINT32 u32LocalAddr, DWORD dwOptions, BOOL fPolling)
{
    PVOID pBuf;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;

    /* Allocate a user-mode buffer for Scatter/Gather DMA */
    pBuf = malloc(dwDMABufSize); /* dwDMABufSize = number of bytes to allocate */
    if (!pBuf)
    {
        printf("Failed allocating a user-mode DMA buffer\n");
        return FALSE;
    }
    memset(pBuf, 0, dwDMABufSize);

    /* Allocate a DMA buffer and open DMA for the selected channel */
    if (!DMAOpen(hDev, pBuf, u32LocalAddr, dwDMAChannel, dwDMABufSize, &pDma))
    {
        free(pBuf);
        return FALSE;
    }

    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
        {
            printf("Failed enabling DMA interrupts\n");
            goto Error;
        }
    }

    /* Flush the data from the CPU caches in order to synchronize these caches
        with the DMA buffer (see documentation of WDC_DMASyncCpu()) */
    WDC_DMASyncCpu(pDma);

    /* Start DMA - write to the device to initiate the DMA transfer */
    if (!MyDMAStart(hDev, pDma))
    {

```

```

        printf("Failed initiating DMA\n");
        goto Error;
    }

    /* Wait for the DMA transfer to complete */
    MyDMAIsDone(hDev, pDma);

    /* Flush the data from the I/O caches and update the CPU caches in order
       to synchronize the I/O caches with the DMA buffer (see documentation of
       WDC_DMASyncIo()) */
    WDC_DMASyncIo(pDma);

    fRet = TRUE;

Exit:
    DMAClose(pDma, pBuf, fPolling);
    return fRet;
}

/* DMAOpen: Allocates and locks a Scatter/Gather DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID pBuf, UINT32 u32LocalAddr,
             DWORD dwDMAChannel, DWORD dwDMABufSize, WD_DMA *ppDma)
{
    DWORD dwStatus, dwPageNumber;
    BOOL fRead = dwOptions & DMA_READ_FROM_DEVICE ? TRUE : FALSE;

    /* Allocate and lock a Scatter/Gather DMA buffer */
    dwStatus = WDC_DMASGBufLock(hDev, pBuf, dwOptions, dwDMABufSize, ppDma);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed locking a Scatter/Gather DMA buffer. Error 0x%x - %s\n",
              dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }

    /* Program the device's DMA registers for each physical page */
    for(dwPageNumber = 0; dwPageNumber < (*ppDma)->dwPages; dwPageNumber++)
    {
        MyDMAPageProgram(u32LocalAddr, (*ppDma)->Page[dwPageNumber].pPhysicalAddr,
                        (*ppDma)->Page[dwPageNumber].dwBytes, fRead);
    }
}

```

```

    return TRUE;
}

/* DMAClose: Frees a previously allocated Scatter/Gather DMA buffer */
void DMAClose(WD_DMA *pDma, PVOID pBuf, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);

    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);

    /* Free the virtual user-mode DMA buffer */
    if (pBuf)
        free(pBuf);
}

```

9.1.1.2 What Should You Implement?

In the code sample above, it is up to you to implement the following `MyDMAxxx()` routines, according to your device's specification

- `MyDMAPageProgram()`: Program the device's DMA registers.
The exact method of supporting Scatter/Gather DMA varies between different devices.
- `MyDMAStart()`: Write to the device to initiate DMA transfers.
- `MyDMAInterruptEnable()` and `MyDMAInterruptDisable()`: Use `WDC_IntEnable()` [A.2.41] and `WDC_IntDisable()` [A.2.42] (respectively) to enable/disable the software interrupts and write/read the relevant register(s) on the device in order to physically enable/disable the hardware DMA interrupts (see section 9.2 for details regarding interrupt handling with WinDriver.)
- `MyDMAIsDone()`: When using a polling mechanism to determine DMA completion, implement this function as a loop that reads continuously from the device in order to determine when the DMA transfer has completed. When using interrupts, wait for a signal from your interrupt handler routine to determine DMA completion.

NOTE

When using the basic WD_xxxx API to allocate a Scatter/Gather DMA buffer that is larger than 1MB, you need to set the DMA_LARGE_BUFFER flag in the call to WD_DMALock() [A.4.16] and allocate memory for the additional memory pages, as explained in the following FAQ: <http://www.jungo.com/support/faq.html#dma1>. However, when using WDC_DMASGBufLock() [A.2.37] to allocate the DMA buffer, you do not need any special implementation for allocating large buffers, since the function handles this for you.

9.1.2 Contiguous Buffer DMA

Following is a sample routine that uses WinDriver's WDC API [A.1] to allocate a Contiguous DMA buffer and perform bus-master DMA transfers.

A more detailed example specific to the enhanced support PLX chipsets ([7]) can be found in the **/WinDriver/plx/lib/plx_lib.c** library file and **/WinDriver/plx/diag_lib/plx_diag_lib.c** diagnostics library file (which utilizes the **plx_lib.c** DMA API.)

A sample of using the basic WD_DMAxxx API for implementing Contiguous Buffer DMA for the AMCC 5933 chip can be found in the **/WinDriver/amcc/lib/amcclib.c** library file.

9.1.2.1 Sample Contiguous Buffer DMA Implementation

```

BOOL DMARoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMAChannel, DWORD dwDMABufSize,
    UINT32 u32LocalAddr, DWORD dwOptions, BOOL fPolling)
{
    PVOID pBuf = NULL;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;

    /* Allocate a DMA buffer and open DMA for the selected channel */
    if (!DMAOpen(hDev, &pBuf, u32LocalAddr, dwDMAChannel, dwDMABufSize, &pDma))
    {
        free(pBuf);
        return FALSE;
    }

    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)

```



```

    {
        if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
        {
            printf("Failed enabling DMA interrupts\n");
            goto Error;
        }
    }

    /* Flush the data from the CPU caches in order to synchronize these caches
       with the DMA buffer (see documentation of WDC_DMASyncCpu()) */
    WDC_DMASyncCpu(pDma);

    /* Start DMA - write to the device to initiate the DMA transfer */
    if (!MyDMAStart(hDev, pDma))
    {
        printf("Failed initiating DMA\n");
        goto Error;
    }

    /* Wait for the DMA transfer to complete */
    MyDMAIsDone(hDev, pDma);

    /* Flush the data from the I/O caches and update the CPU caches in order
       to synchronize the I/O caches with the DMA buffer (see documentation of
       WDC_DMASyncIo()) */
    WDC_DMASyncIo(pDma);

    fRet = TRUE;

Exit:
    DMAClose(pDma, fPolling);
    return fRet;
}

/* DMAOpen: Allocates and locks a Contiguous DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID *ppBuf, UINT32 u32LocalAddr,
             DWORD dwDMAChannel, DWORD dwDMABufSize, WD_DMA *ppDma)
{
    DWORD dwStatus;
    BOOL fRead = dwOptions & DMA_READ_FROM_DEVICE ? TRUE : FALSE;

    /* Allocate and lock a Contiguous DMA buffer */

```

```

    dwStatus = WDC_DMABufLock(hDev, ppBuf, dwOptions, dwDMABufSize, ppDma);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed locking a Contiguous DMA buffer. Error 0x%x - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }

    /* Program the device's DMA registers for the physical DMA page */
    MyDMAPageProgram(u32LocalAddr, (*ppDma)->Page[0].pPhysicalAddr,
        (*ppDma)->Page[0].dwBytes, fRead);

    return TRUE;
}

/* DMAClose: Frees a previously allocated Contiguous DMA buffer */
void DMAClose(WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);

    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);
}

```

9.1.2.2 What Should You Implement?

In the code sample above, it is up to you to implement the following `MyDMAxxx()` routines, according to your device's specification

- `MyDMAPageProgram()`: Program the device's DMA registers
- `MyDMAStart()`: Write to the device to initiate DMA transfers.
- `MyDMAInterruptEnable()` and `MyDMAInterruptDisable()`: Use `WDC_IntEnable()` [A.2.41] and `WDC_IntDisable()` [A.2.42] (respectively) to enable/disable the software interrupts and write/read the relevant register(s) on the device in order to physically enable/disable the hardware DMA interrupts (see section 9.2 for details regarding interrupt handling with WinDriver.)
- `MyDMAIsDone()`: When using a polling mechanism to determine DMA completion, implement this function as a loop that reads continuously from the device in order to determine when the DMA transfer has completed.

When using interrupts, wait for a signal from your interrupt handler routine to determine DMA completion.

9.1.3 Performing DMA on SPARC

The SPARC platform supports Direct Virtual Memory Access (**DVMA**). Platforms that support DVMA provide the device with a virtual address rather than a physical address. With this memory access method, the platform translates device accesses to the provided virtual address into the proper physical addresses using a type of Memory Management Unit (MMU). The device transfers data to and from a contiguous virtual image that can be mapped to dis-contiguous physical pages. Devices that operate on these platforms do not require Scatter/Gather DMA capability.

9.2 Handling Interrupts

WinDriver provides you with API, DriverWizard code generation, and samples, in order to simplify the task of handling interrupts from your driver.

If you are developing a driver for a device based on one of the enhanced-support WinDriver chipsets ([7]), we recommend that you use the custom WinDriver interrupt APIs for your specific chip in order to handle the interrupts, since these routines are implemented specifically for the target hardware.

For other chips, we recommend that you use the DriverWizard to detect/define the relevant information regarding the device interrupt (such as the interrupt request (IRQ) number, its type and its shared state), define commands to be executed in the kernel when an interrupt occurs, and then generate skeletal diagnostics code, which includes interrupt routines that demonstrate how to use WinDriver's API to handle your device's interrupts, based on the information that you defined in the wizard.

The following sections describe how to use WinDriver's API to handle PCI, PCMCIA and ISA interrupts. Read these sections in order to understand the sample and generated DriverWizard interrupt code or to write your own interrupt handler.

NOTE

This section describes how to use WinDriver to handle interrupts from a user-mode application. Since interrupt handling is a performance-critical task, it is very likely that you may want to handle the interrupts directly in the kernel. WinDriver's Kernel PlugIn [11] enables you to implement kernel interrupt routines. To find out how to handle interrupts from the Kernel PlugIn, please refer to section 11.6.5 of the manual.

9.2.1 General – Handling an Interrupt

The interrupt handling sequence using WinDriver is as follows:

1. When the user selects to enable interrupts on the device, a thread is created to handle incoming interrupts.
2. The thread runs an infinite loop that waits for an interrupt to occur.
3. When an interrupt occurs, WinDriver executes, in the kernel, any transfer commands prepared in advance by the user (see section [9.2.2.1]) and when the control returns to the user mode, the driver's interrupt handler routine is called.
4. When the interrupt handler code returns, the wait loop continues.

The low-level WinDriver `WD_IntWait()` function [A.5.3], which is used to wait for interrupts from the device, puts the thread to sleep until an interrupt occurs. There is no CPU consumption while waiting for an interrupt. Once an interrupt occurs, it is first handled by the WinDriver kernel, then the `WD_IntWait()` wakes up the interrupt handler thread and returns.

Since your interrupt thread runs in the user mode, you may call any Windows API from this thread, including file handling and GDI functions.

9.2.1.1 Basic Interrupt Handler Code Sequence

Below is a sample of a simple interrupt handler routine for edge-triggered interrupts (normally ISA/EISA – see section 9.2.2).

NOTE

The code below uses WinDriver's low-level `WD_xxx` interrupt functions. When writing your own interrupt handler, we recommend that you use the convenient wrapper **WDC** interrupt functions [9.2.1.3] or at least the high-level **windrvr_int_thread.c** WinDriver API [9.2.1.2], which encapsulate the low-level interrupt handling.

```
WD_INTERRUPT intrp; /* Interrupt information structure */

DWORD WINAPI wait_interrupt (PVOID pData)
{
    printf("Waiting for interrupt");
    for (;;)
    {
        WD_IntWait(hWD, &intrp);
        if (intrp.fStopped)
```

```

        break; /* WD_IntDisable called by parent */

        /* Call your interrupt routine here */
        printf("Got interrupt %d\n", intrp.dwCounter);
    }
    return 0;
}

void install_interrupt()
{
    BZERO(intrp);
    /* Set the interrupt handle, returned by WD_CardRegister() */
    intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
    /* No kernel transfer commands to perform upon interrupt */
    intrp.Cmd = NULL;
    intrp.dwCmds = 0;
    /* No special interrupt options */
    intrp.dwOptions = 0;
    WD_IntEnable(hWD, &intrp);
    if (!intrp.fEnableOk)
    {
        printf("Failed enabling interrupts\n");
        return;
    }
    printf("Starting interrupt thread\n");
    hThread = CreateThread (0, 0x1000,
        wait_interrupt, NULL, 0, &thread_id);
    /* Call your driver's interrupt handler here */
    WD_IntDisable (hWD, &intrp);
    WaitForSingleObject(hThread, INFINITE);
}

```

9.2.1.2 Simplified Interrupt Handling Using windrvr_int_thread.c

WinDriver provides the following convenience functions to further simplify the interrupt handling: `InterruptEnable()` [A.4.21] and `InterruptDisable()` [A.4.22]. Both functions are implemented in `/WinDriver/src/windrvr_int_thread.c`. Please refer to the implementation of these functions for a better understanding of how this mechanism operates.

NOTE

The WinDriver `InterruptEnable()` and `InterruptDisable()` APIs, described in this section, are utilized by the **WDC** library's interrupt APIs [9.2.1.3], which simplify the task of implementing an interrupt handler even further. We recommend that you use the WDC convenience wrapper interrupt APIs when writing your interrupt handler code.

In the following example, we rewrote the code from section 9.2.1.1 to use the **windrvr_int_thread.c** convenience interrupt functions. This code was extracted from the sample program **int_io.c**, which can be found under the **/WinDriver/samples/int_io** directory. Please refer to this file for the full source code.

```
VOID DLLCALLCONV interrupt_handler (PVOID pData)
{
    WD_INTERRUPT *pIntrp = (WD_INTERRUPT *)pData;

    /* Implement your interrupt handler routine here */

    printf("Got interrupt %d\n", pIntrp->dwCounter);
}

...

int main()
{
    HANDLE hWD;
    WD_CARD_REGISTER cardReg;
    WD_INTERRUPT *pIntrp; /* Pointer to interrupt information structure */
    HANDLE hThread;
    ...
    hWD = WD_Open();
    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_INTERRUPT;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.Int.dwInterrupt = MY_IRQ;
    cardReg.Card.Item[0].I.Int.dwOptions = 0;
    ...
    WD_CardRegister(hWD, &cardReg);
    ...
    pIntrp = malloc(sizeof(WD_INTERRUPT));
```

```

    BZERO(*pIntrp);
    pIntrp->hInterrupt =
        cardReg.Card.Item[0].I.Int.hInterrupt;
    printf ("Starting interrupt thread\n");
    ...
    dwStatus = InterruptEnable(&hThread, hWD, pIntrp,
        interrupt_handler, pIntrp)
    /* InterruptEnable() calls WD_IntEnable() and creates an
       interrupt handler thread */
    if (dwStatus)
    {
        printf ("failed enabling interrupt Status 0x%x - %s\n",
            dwStatus, Stat2Str(dwStatus));
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);

        InterruptDisable(hThread);
        /* InterruptDisable() calls WD_IntDisable() */
    }
    WD_CardUnregister(hWD, &cardReg);
    free(pIntrp);
    ....
}

```

In the above code, the function `interrupt_handler()` serves as our interrupt handler routine, invoked once for every interrupt that occurs. In the simplified code for setting up interrupt handling, we call `InterruptEnable()` [A.4.21]. This function spawns a thread, which in turn calls the function `interrupt_handler()`. A pointer to this function is passed as the fourth parameter to `InterruptEnable()`. Each time an interrupt occurs, the data specified by the fifth parameter (`pData`) is passed to the function.

9.2.1.3 Handling Interrupts with the WDC Library

The WinDriver Card (WDC) library [A.1] provides convenience wrappers to the basic WinDriver PCI/PCMCIA/ISA API, including simplified interrupt handling functions – `WDC_IntEnable()`, `WDC_IntDisable()` and `WDC_IntIsEnabled()`. For a detailed description of these functions, please refer to sections A.2.41 – A.2.43 of the manual.

The sample code below demonstrates how you can use the WDC interrupt APIs to implement a simple interrupt handler, in place of the examples shown above in this section 9.2.1.1 / 9.2.1.2. For complete interrupt handler source code that uses these functions, you can refer to the WinDriver pci_diag (/WinDriver/samples/pci_diag/), pcmcia_diag (/WinDriver/samples/pcmcia_diag/) and PLX (/WinDriver/plx/) samples and to the generated DriverWizard PCI/PCMCIA/ISA code.

```
VOID DLLCALLCONV interrupt_handler (PVOID pData)
{
    PWDC_DEVICE pDev = (PWDC_DEVICE)pData;

    /* Implement your interrupt handler routine here */

    printf("Got interrupt %d\n", pDev->Int.dwCounter);
}

...

int main()
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    ...
    WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, NULL);
    ...
    hDev = WDC_IsaDeviceOpen(...);
    ...
    /* Enable interrupts. This sample passes the WDC device handle as the data
       for the interrupt handler routine */
    dwStatus = WDC_IntEnable(hDev, NULL, 0, 0,
        interrupt_handler, (PVOID)hDev, FALSE);
    /* WDC_IntEnable() allocates and initializes the required WD_INTERRUPT
       structure, stores it in the WDC_DEVICE structure, then calls
       InterruptEnable(), which calls WD_IntEnable() and creates an interrupt
       handler thread */
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf ("Failed enabling interrupt. Error: 0x%x - %s\n",
            dwStatus, Stat2Str(dwStatus));
    }
    else
    {

```



```

printf("Press Enter to uninstall interrupt\n");
fgets(line, sizeof(line), stdin);

WDC_IntDisable(hDev);
/* WDC_IntDisable() calls InterruptDisable(), which calls WD_IntDisable() */
}
...
WDC_IsaDeviceClose(hDev);
...
WDC_DriverClose();
}

```

9.2.2 ISA/EISA and PCI Interrupts

Generally, ISA/EISA interrupts are edge triggered, in contrast to PCI interrupts, which are level sensitive. This has many implications on how the interrupt handler routine is written.

Edge-triggered interrupts are generated once, when the physical interrupt signal goes from low to high. Therefore, exactly one interrupt is generated. As a result, the operating system calls the WinDriver kernel interrupt handler, which releases the thread waiting for the interrupt (i.e. the thread that called `WD_IntWait()` [A.5.3].) No special action is required in order to acknowledge this type of interrupt.

Level-sensitive interrupts are generated as long as the physical interrupt signal is high. If the interrupt signal is not lowered by the end of the interrupt handling in the kernel, the operating system will call the WinDriver kernel interrupt handler again, causing the PC to hang! To prevent such a situation, the interrupt must be acknowledged by the WinDriver kernel interrupt handler.

9.2.2.1 Transfer Commands at Kernel Level (Acknowledging the Interrupt)

Usually, interrupt handlers for level-sensitive interrupts, such as PCI interrupts, need to perform transfer commands – i.e. write to and/or read from the device – at the kernel in order to lower the interrupt level (acknowledge the interrupt). The transfer commands typically write to run-time registers on the card, thus clearing the hardware interrupt. However, the exact transfer commands to be performed in order to acknowledge the interrupts are hardware-specific. Therefore, when using WinDriver to handle level-sensitive interrupts, you must inform WinDriver in advance what transfer commands to perform in the kernel when an interrupt is received.

To pass transfer commands to be performed in the WinDriver kernel interrupt handler (before `WD_IntWait()` [A.5.3] returns), you need to prepare an array of commands (defined using a `WD_TRANSFER` structure), and pass the array to WinDriver when enabling the interrupts using `WDC_IntEnable()` [A.2.41] or the lower-level `InterruptEnable()` function [A.4.21] (both of which call the low-level `WD_IntEnable()` function [A.5.2].)

The interrupt enable functions also enable you to define an interrupt mask in order to verify the source of the interrupt. Note that interrupt mask commands must be set directly after a read transfer command in the transfer commands array. If you set an interrupt mask command (`trans[i].cmdTrans = CDM_MASK`), upon the arrival of an interrupt in the kernel, WinDriver will compare the value read from the card in the preceding read command with the mask set in the interrupt mask command. If the values match, WinDriver will claim control of the interrupt, execute the rest of the transfer commands in the array, and invoke your interrupt handler routine when the control returns to the user mode. However, if the values do not match, WinDriver will reject control of the interrupt, the rest of the interrupt transfer commands will not be executed and your interrupt handler routine will not be invoked.

For example, suppose that when an interrupt occurs you expect the value of your card's interrupt command-status register (INTCSR), which is mapped to an I/O port address (`dwAddr`), to be `intrMask`, and that in order to clear the interrupt you need to write 0 to the INTCSR. In this case, you could use the following code to define an array of transfer commands that first reads the INTCSR register, saves its value, then masks it to verify the source of the interrupt and writes 0 to the INTCSR to acknowledge the interrupt (all commands in the example are performed in modes of `DWORD`):

```
WD_TRANSFER trans[3]; /* Array of WinDriver transfer command structures */
BZERO(trans);

/* 1st command: Read a DWORD from the INTCSR I/O port */
trans[0].cmdTrans = RP_DWORD;
/* Set address of IO port to read from: */
trans[0].dwPort = dwAddr; /* Assume dwAddr holds the address of INTCSR */

/* 2nd command: Mask the interrupt to verify its source */
trans[1].cmdTrans = CMD_MASK;
trans[1].Data.Dword = intrMask; /* Assume intrMask holds your interrupt mask */

/* 3rd command: Write DWORD to the INTCSR I/O port.
   This command will only be executed if the value read from INTCSR in the
   1st command matches the interrupt mask set in the 2nd command. */
trans[2].cmdTrans = WP_DWORD;
```

```

/* Set the address of IO port to write to: */
trans[2].dwPort = dwAddr; /* Assume dwAddr holds the address of INTCSR */
/* Set the data to write to the INTCSR IO port: */
trans[2].Data.Dword = 0;

```

After defining the transfer commands, you can proceed to enable the interrupts.

The following code demonstrates how to use the WDC library [A.1] to enable the interrupts, using the transfer commands prepared above:

```

/* Enable the interrupts:
   hDev: WDC_DEVICE_HANDLE received from a previous call to WDC_PciDeviceOpen()
   INTERRUPT_CMD_COPY: Used to save the read data - see explanation below
   interrupt_handler: Your user-mode interrupt handler routine
   pData: The data to pass to the interrupt handler routine */
WDC_IntEnable(hDev, &trans, 3, INTERRUPT_CMD_COPY, interrupt_handler,
  pData, FALSE);

```

If you are not using the WDC library, you can enable the interrupts using InterruptEnable() [A.4.21], as demonstrated below:

```

WD_INTERRUPT intrp; /* WinDriver interrupt information structure */
BZERO(intrp);
/* Set the interrupt handle with the handle returned from a previous call to
   WD_CardRegister() */
intrp.hInterrupt = cardReg.Card.Item[i].I.Int.hInterrupt;
/* Set the number of interrupt transfer commands */
intrp.dwCmds = 3;
/* Set the interrupt transfer commands (allocated above) */
intrp.Cmd = trans;
/* Set the interrupt options.
   Use INTERRUPT_CMD_COPY to store read data - see explanation below */
intrp.dwOptions = INTERRUPT_CMD_COPY;

/* Enable the interrupts:
   hThread: Thread handle, which will be updated by the function. This handle
           should later be passed to InterruptDisable().
   hWD: Handle to WinDriver, received from a previous call to WD_Open()
   intrp: Interrupt information structure (see above)
   interrupt_handler: Your user-mode interrupt handler routine
   pData: The data to pass to the interrupt handler routine */
InterruptEnable(&hThread, hWD, &intrp, interrupt_handler, pData);

```

The **INTERRUPT_CMD_COPY** flag is used to retrieve the value read by the first transfer command, before the write command is issued. This is useful when you need to read the value of a register, and then write to it to lower the interrupt level.

If you try to read this register from the user-mode when an interrupt occurs, it will already be "0" because the write transfer command was issued at kernel level. Using `INTERRUPT_CMD_COPY`, the read value of the first transfer command will be stored, in the kernel, in the `trans[0].Data.Dword` field and you will be able to refer to this value from your user-mode interrupt handler routine.

9.2.3 Improving the Interrupt Handling Rate on VxWorks

WinDriver for VxWorks (a.k.a. DriverBuilder) implements a call-back routine, which, if set by the user, is executed immediately when an interrupt is received, thus enabling you to speed up interrupt acknowledgment and processing. This routine is not needed on Windows 98/Me/NT/2000/XP/Server 2003, Linux and Solaris, since you can use the Kernel PlugIn feature to improve the interrupt handling rate on these platforms. See Section 11 or <http://www.jungo.com/kpi.html> for more information about the Kernel PlugIn.

To use the `windrivr_isr()` routine:

1. Include the following declaration in your code:

```
int (__cdecl *windrivr_isr)(void);
```

2. Set `windrivr_isr()` to point to the interrupt handler routine that you wish to have performed immediately upon the arrival of an interrupt. For example:

```
int __cdecl my_isr(void)
{
    /* Add code here in order to verify that the ISR is called */
    return TRUE; /* If TRUE, continue regular handling of WinDriver;
                  If FALSE, exit ISR */
}

extern int (__cdecl *windrivr_isr)(void);

/* after calling drvInit() */
windrivr_isr = my_isr;
```

9.2.4 Interrupts in Windows CE

Windows CE uses a logical interrupt scheme rather than the physical interrupt number. It maintains an internal kernel table that maps the physical IRQ number to the logical IRQ number. Device drivers are expected to use the logical interrupt number when requesting interrupts from Windows CE. In this context, there are three approaches to interrupt mapping:

1. Use Windows CE Plug-and-Play for Interrupt Mapping (PCI bus driver)

This is the recommended approach to interrupt mapping in Windows CE. Register the device with the PCI bus driver. Following this method will cause the PCI bus driver to perform the IRQ mapping and direct WinDriver to use it.

Refer to Section 5.3 for an example how to register your device with the PCI bus driver.

2. Use the Platform Interrupt Mapping (On X86 or ARM)

In most of the x86 or MIPS platforms, all the physical interrupts except for some reserved ones are statically mapped using this simple mapping:

```
logical interrupt = SYSINTR_FIRMWARE + physical interrupt
```

When the device is not registered with Windows CE Plug-and-Play, WinDriver will follow this mapping.

3. Specify the Mapped Interrupt Value

NOTE

This option can only be performed by the Platform Builder.

Provide the device's mapped logical interrupt value. If unavailable, statically map the physical IRQ to a logical interrupt. Then call **WD_CardRegister** with the logical interrupt and with the **INTERRUPT_CE_INT_ID** flag set. The static interrupt map is in the file **CFWPC.C** (located in the **%_TARGETPLATROOT%\KERNEL\HAL** directory).

You will then need to rebuild the Windows CE image **NK.BIN** and download the new executable onto your target platform.

Static mapping is helpful also in the case of using reserved interrupt mapping. Suppose your platform static mapping is:

- **IRQ0**: Timer Interrupt
- **IRQ2**: Cascade interrupt for the second PIC
- **IRQ6**: The floppy controller
- **IRQ7**: LPT1 (because the PPSH does not use interrupts)
- **IRQ9**
- **IRQ13**: The numeric coprocessor

An attempt to initialize and use any of these interrupts will fail. However, you may want to use one or more of these interrupts on occasion, such as when you do not want to use the PPSH, but you want to reclaim the parallel port for some other purpose. To solve this problem, simply modify the file **CFWPC.C**

(located in the `%_TARGETPLATROOT%\KERNEL\HAL` directory) to include code, as shown below, that sets up a value for interrupt 7 in the interrupt mapping table:

```
SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+7,7);
```

Suppose you have a PCI card which was assigned IRQ9. Since Windows CE does not map this interrupt by default, you will not be able to receive interrupts from this card. In this case, you will need to insert a similar entry for IRQ9:

```
SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+9,9);
```

9.2.4.1 Improving Interrupt Latency in Windows CE

You can reduce the interrupt latency in Windows CE for PCI devices by making slight changes in the registry and in your code:

1. When developing your driver on Windows CE platforms, you must first register your device to work with WinDriver, as explained in Section 5.3.

Change the last value in the registry from:

```
"WdIntEnh"=dword:0
```

to:

```
"WdIntEnh"=dword:1
```

If you exclude this line, or leave the value 0, the interrupt latency will not be reduced.

2. Add **WD_CE_ENHANCED_INTR** to your Preprocessor Definitions of your project and recompile your entire project. When using Microsoft eMbedded Visual C++, the Preprocessor Definitions are found under Project Settings.
3. Immediately after calling `InterruptEnable()`, call `CEInterruptEnhance()`, which receives two parameters:
 - A handle to the interrupt thread received from `InterruptEnable()`.
 - The mapped interrupt value. You can find the mapped interrupt value by calling `PCI_GetInterruptItemIndex()` and retrieving the information from the returned value. For example:

```
CEInterruptEnhance(hThread, hPci->cardReg.Card.Item  
[PCI_GetInterruptItemIndex(hPci)].I.Val.dw4);
```

NOTE

The high-level `WDC_IntEnable()` function [A.2.41] automatically calls `CEInterruptEnhance()`, therefore when using WinDriver's **WDC** APIs [A.1] to handle the interrupts, you do not need to call `CEInterruptEnhance()` yourself

9.3 Support for 64-bit Operating Systems

NOTES

- Starting from version 6.02, WinDriver supports Solaris 8.0/9.0 64-bit kernels. Refer to Section 3.1.5 for the full list of Solaris platforms supported by WinDriver.
 - Starting from version 7.00 WinDriver supports the Linux AMD64 and Intel EM64T architectures (**x86_64**). Refer to Section 3.1.4 for the full list of Linux architectures supported by WinDriver.
- WinDriver for Solaris 64-bit and WinDriver for Linux 64-bit kernels support both 32-bit and 64-bit applications.
Note that 64-bit applications are more efficient.
 - In general, DWORD is unsigned long. On 32-bit applications DWORD is a 32-bit variable and on 64-bit applications DWORD is a 64-bit variable. The exception is the transfer commands used in the `WD_Transfer()` struct. In `WD_Transfer()` struct DWORD stands for 32-bit and QWORD stands for 64-bit regardless of how the application was compiled. This exception was made for backward code compatibility and for compatibility between 32-bit and 64-bit applications.
 - On 64-bit operating systems, 64-bit data transfers are performed directly. There is no need to use the WinDriver transfer functions.

9.4 Byte Ordering

9.4.1 Introduction to Endianness

There are two main architectures for handling memory storage. They are called Big Endian and Little Endian and refer to the order in which the bytes are stored in memory.

- Big endian means that the most significant byte of any multi-byte data field is stored at the lowest memory address.
This means a Hex word like 0x1234 is stored in memory as (0x12 0x34). The big end, or upper end, is stored first. The same is true for a four-byte value; for example, 0x12345678 would be stored as (0x12 0x34 0x56 0x78).
- Little endian means that the least significant byte of any multi-byte data field is stored at the lowest memory address.

This means a Hex word like 0x1234 is stored in memory as (0x34 0x12). The little end, or lower end, is stored first. The same is true for a four-byte value; for example, 0x12345678 would be stored as (0x78 0x56 0x34 0x12).

All processors are designated as either big endian or little endian. Intel's x86 processors and their clones are little endian. Sun's SPARC, Motorola's 68K, and the PowerPC families are all big endian.

An endianness difference can cause problems if a computer unknowingly tries to read binary data written in the opposite format from a shared memory location or file.

The terms big endian and little endian are derived from the Lilliputians of Gulliver's Travels (Jonathan Swift 1726), whose major political issue was which end of the soft-boiled egg should be opened, the little or the big end.

9.4.2 WinDriver Byte Ordering Macros

The PCI bus is designated as little endian, complying with x86 architecture. In order to prevent problems resulting from byte ordering incompatibility between the PCI bus and SPARC and PowerPC architectures, WinDriver includes macro definitions that convert data between little and big endian.

When developing drivers using WinDriver, these macro definitions enable cross platform portability. Using these macro definitions is safe even for drivers that are going to be deployed on x86 architecture.

The following sections describe the macros and when to use them.

9.4.3 Macros for PCI Target Access

WinDriver's macros for PCI target access are used for converting endianness while reading/writing from/to PCI cards using memory mapped ranges of PCI devices.

NOTE

These macro definitions apply to Linux PowerPC architecture.

- **dtoh16** - Macro definition for converting a WORD (device to host)
- **dtoh32** - Macro definition for converting a DWORD (device to host)
- **dtoh64** - Macro definition for converting a QWORD (device to host)

Use WinDriver's macro definitions in the following situations:

1. Apply the macro on the data you write to the device in cases of direct write access to the card using a memory mapped range.

For example:

```
DWORD data = VALUE;  
*mapped_address = dtoh32(data);
```

2. Apply the macro on the data you read from the device in cases of direct read access from the card using a memory mapped range.

For example:

```
WORD data = dtoh16(*mapped_address);
```

NOTE

WinDriver's APIs – WDC_Read/WriteXXX() [A.2.16 – A.2.21], WDC_MultiTransfer() [A.2.22], and the lower level WD_Transfer() [A.4.14] and WD_MultiTransfer() [A.4.15] functions – already perform the required byte ordering translations, therefore when using these APIs to read/write memory addresses you do not need to use the dtoh16/32/64() macros to convert the data (nor is this required for I/O addresses).

9.4.4 Macros for PCI Master Access

WinDriver's macros for PCI master access are used for converting endianness of data in host memory that is accessed by the PCI master device, i.e. in cases of access that is initiated by the device rather than the host.

NOTE

These macro definitions apply to both Linux PowerPC and SPARC architectures.

- **htod16** - Macro definition for converting a WORD (host to device)
- **htod32** - Macro definition for converting a DWORD (host to device)
- **htod64** - Macro definition for converting a QWORD (host to device)

Use WinDriver's macro definitions in the following situations:

Apply the macro on data you prepare on the host memory that will be read/written by the card. An example of such a case is a chain of descriptors for scatter/gather DMA.

The following example is an extract from the PLX_DMAOpen() function in WinDriver's PLX library (see **WinDriver/plx/lib/plx_lib.c**):

```

/* Setting chain of DMA pages in the memory */
for (dwPageNumber = 0, u32MemoryCopied = 0;
     dwPageNumber < pPLXDma->pDma->dwPages;
     dwPageNumber++)
{
    pList[dwPageNumber].u32PADR =
        htod32((UINT32)pPLXDma->pDma->Page[dwPageNumber].pPhysicalAddr);
    pList[dwPageNumber].u32LADR =
        htod32((u32LocalAddr + (fAutoinc ? u32MemoryCopied : 0)));
    pList[dwPageNumber].u32SIZ =
        htod32((UINT32)pPLXDma->pDma->Page[dwPageNumber].dwBytes);
    pList[dwPageNumber].u32DPR =
        htod32((u32StartOfChain + sizeof(DMA_LIST) * (dwPageNumber + 1))
                | BIT0 | (fIsRead ? BIT3 : 0));
    u32MemoryCopied += pPLXDma->pDma->Page[dwPageNumber].dwBytes;
}

pList[dwPageNumber - 1].u32DPR |= htod32(BIT1); /* Mark end of chain */

```

Chapter 10

Improving Performance

10.1 Overview

Once your user-mode driver has been written and debugged, you might find that certain modules in your code do not operate fast enough (for example: an interrupt handler or accessing I/O-mapped regions). If this is the case, try to improve performance in one of the following ways:

- Improve the performance of your user-mode driver.
- Move the performance-critical parts of your code into WinDriver's Kernel PlugIn.

NOTE

Kernel PlugIn is not implemented under Windows CE and VxWorks. In these operating systems there is no separation between kernel mode and user mode, therefore top performance can be achieved without using the Kernel PlugIn. To improve the interrupt handling rate on VxWorks, use the **windrvr_isr** hook, as explained in section 9.2.3 of the manual.

Use the following checklist to determine how to best improve the performance of your driver.

10.1.1 Performance Improvement Checklist

The following checklist will help you determine how to improve the performance of your driver:

Problem	Solution
ISA Card – accessing an I/O-mapped range on the card	<p>Try to convert multiple calls to <code>WD_Transfer()</code> to one call to <code>WD_MultiTransfer()</code> (see Section 10.2.2 later in this chapter).</p> <p>If this does not solve the problem, handle the I/O at kernel mode by writing a Kernel PlugIn (see the Kernel PlugIn-related chapters for details, Chapters 11 and 12).</p>
PCI Card – accessing an I/O-mapped range on the card	<p>First, try to change the card from I/O-mapped to memory-mapped by changing bit 0 of the address space PCI configuration register to 0 and then try the solutions for problem #3. You will probably need to reprogram the EPROM to initialize BAR0/1/2/3/4/5 registers with different values.</p> <p>If this is not possible, try the solutions suggested for problem #1.</p> <p>If this does not solve the problem, handle the I/O in kernel mode by writing a Kernel PlugIn (see the Kernel PlugIn-related chapters for details, Chapters 11 and 12).</p>
Accessing a memory-mapped range on the card	<p>Try to access memory without using <code>WD_Transfer()</code>, and instead using direct access to memory-mapped regions (see Section 10.2.1 later in this chapter).</p> <p>If the problem persists, then there is a hardware design problem. You will not be able to increase performance by using any software design method, writing a Kernel PlugIn, or even by writing a full kernel driver.</p>
Interrupt latency – missing interrupts, receiving interrupts too late	Handle the interrupts in kernel mode by writing a kernel PlugIn (see the Kernel PlugIn-related chapters for details, Chapters 11 and 12).
USB devices – slow transfer rate	To increase the transfer rate, try to increase the packet size by choosing a different device configuration.

10.2 Improving the Performance of a User-Mode Driver

As a general rule, transfers to memory-mapped regions are faster than transfers to I/O-mapped regions. The reason is that WinDriver enables the user to directly access the memory-mapped regions without calling the `WD_Transfer()` function.

10.2.1 Using Direct Access to Memory-Mapped Regions

After registering a memory-mapped region using `WD_CardRegister()` [A.4.11], two results are returned: `dwTransAddr` and `dwUserDirectAddr`.

The `dwTransAddr` result should be used as a base address when calling `WD_Transfer()` [A.4.14] to read or write to the memory region. A more efficient way to perform memory transfers would be to use `dwUserDirectAddr` directly as a pointer, and then use it to access the memory-mapped range. This method enables you to read/write data to your memory-mapped region without any function calls overhead, i.e., zero performance degradation.

Whether you use `WD_Transfer()` or `dwUserDirectAddr`, it is important to align the base address according to the size of the data type, especially when issuing string transfer commands. Otherwise, the transfers are split into smaller portions. The easiest way to align data is to use basic types when defining a buffer, i.e.

```
BYTE buf[len]; // for BYTE transfers - not aligned
WORD buf[len]; // for WORD transfers - aligned on 2-byte boundary
UINT32 buf[len]; // for DWORD transfers - aligned on 4-byte boundary
UINT64 buf[len]; // for QWORD transfers - aligned on 8-byte boundary
```

10.2.2 Accessing I/O-Mapped Regions

The only way to transfer data on I/O-mapped regions is by calling a `WD_Transfer()` function [A.4.14]. If you need to transfer a large buffer, the String (Block) Transfer commands can be used. For example, `RP_SBYTE`, the Read Port String Byte command, will transfer a buffer of bytes to the I/O port. In such cases, the function calling overhead is negligible when compared to the block transfer time.

In a case where many short transfers are called, the function calling overhead may increase to such an extent that it causes overall performance degradation. This might happen if you need to call `WD_Transfer()` more than 20,000 times per second.

Take for example a case in which a 1-MB block of data needs to be transferred word-by-word, and in each word that is transferred first the LOW byte is transferred to I/O port 0x300 and then the HIGH byte is transferred to I/O port 0x301.

Normally this would mean calling `WD_Transfer()` 1 million times—byte 0 to port 0x300, byte 1 to port 0x301, byte 2 to port 0x300, byte 3 to port 0x301, etc. (`WP_BYTE` – Write Port Byte).

A quick way to save 50% of the function call overhead would be to call `WD_Transfer()` with a `WP_SBYTE` (Write Port String Byte), with two bytes at a time. The first call would transfer byte 0 and byte 1 to ports 0x300 and 0x301, the second call would transfer byte 2 and byte 3 to ports 0x300 and 0x301, etc. This way, `WD_Transfer()` will only be called 500,000 times to transfer the block.

The third method would be to prepare an array of 1000 `WD_TRANSFER` commands. Each command in the array would have a `WP_SBYTE` command that transfers two bytes at a time. Then you would call `WD_MultiTransfer()` [A.4.15] with a pointer to the array of `WD_TRANSFER` commands. In one call to `WD_MultiTransfer()`, 2000 bytes of data will be transferred. You will need only 500 calls to `WD_Transfer()` to transfer the 1 MB of data. This is only 0.05% of the original number of calls to `WD_Transfer()`. The trade-off in this case is between the number of calls to `WD_Transfer()` and the memory that is used to setup the 500 `WD_TRANSFER` commands.

10.2.3 Performing 64-bit Data Transfers

NOTE

The ability to perform actual 64-bit transfers is dependant on the existence of support for such transfers by the hardware, CPU, bridge, etc, and can be affected by any of these or their specific combination.

WinDriver supports 64-bit PCI data transfer on x86 platforms running 32-bit operating systems. If your PCI hardware (device and bus) is 64-bit, this feature will enable you to utilize your hardware's broader bandwidth, even though your host operating system is only 32-bit.

This innovative technology makes possible data transfer rates previously unattainable on such platforms. Drivers developed using WinDriver will attain significantly better performance results than drivers written with the DDK or other driver development tools. To date, such tools do not enable 64-bit data transfer on x86 platforms running 32-bit operating systems. Jungo's benchmark performance testing results for 64-bit data transfer indicate a significant improvement of data transfer rates compared to 32-bit data transfer, guaranteeing that drivers developed with WinDriver and KernelDriver will achieve far better performance than 32-bit data transfer normally allows.

To perform 64-bit data transfers, refer to the documentation of `WD_Transfer()` [A.4.14].

NOTE

On 64-bit operating systems, 64-bit data transfers are performed directly. There is no need to use `WD_Transfer()`.

Chapter 11

Understanding the Kernel PlugIn

This chapter provides a description of WinDriver's Kernel PlugIn feature.

NOTE

Kernel PlugIn is not implemented under Windows CE and VxWorks. In these operating systems there is no separation between kernel mode and user mode, therefore top performance can be achieved without using the Kernel PlugIn. To improve the interrupt handling rate on VxWorks, use the **windrvr_isr** hook, as explained in section 9.2.3 of the manual.

11.1 Background

The creation of drivers in user mode imposes a fair amount of function call overhead from the kernel to user mode, which may cause performance to drop to an unacceptable level. In such cases, the Kernel PlugIn feature allows critical sections of the driver code to be moved to the kernel while keeping most of the code intact. Using WinDriver's Kernel PlugIn feature, your driver will operate without any degradation in performance.

The advantages of writing a Kernel PlugIn driver over a standard OS kernel-mode driver are:

- All the driver code is written and debugged in user mode.
- The code segments that are moved to kernel mode remain essentially the same and therefore typically no kernel debugging is needed.

- The parts of the code that will run in the kernel through the Kernel PlugIn are platform independent and therefore will run on every platform supported by WinDriver and the Kernel PlugIn. A standard kernel-mode driver will run only on the platform it was written for.

Using WinDriver's Kernel PlugIn feature, your driver will operate without any performance degradation.

11.2 Do I Need to Write a Kernel PlugIn Driver?

Not every performance problem requires you to write a Kernel PlugIn driver. Some performance problems can be solved in the user-mode driver by better utilization of the features that WinDriver provides. For further information, please refer to [Chapter 10](#).

11.3 What Kind of Performance Can I Expect?

Since you can write your own interrupt handler in the kernel with the WinDriver Kernel PlugIn, you can expect to handle about 100,000 interrupts per second without missing any one of them.

11.4 Overview of the Development Process

Using the WinDriver Kernel PlugIn, you normally first develop and debugs the driver in the user mode, using with the standard WinDriver tools. After identifying the performance-critical parts of the code (such as the interrupt handling or access to I/O-mapped memory ranges), you can create a Kernel PlugIn driver, which runs in kernel mode, and drop the performance-critical portions of your code into the Kernel PlugIn driver, thus eliminating the calling overhead and context switches that occur when implementing the same tasks in the user mode.

This unique architecture allows the developer to start with quick and easy development in the user mode, and progress to performance-oriented code only where needed, thus saving development time and providing for virtually zero performance degradation.

11.5 The Kernel PlugIn Architecture

11.5.1 Architecture Overview

A driver written in user mode uses WinDriver's API (WDC_xxx and/or WD_xxx [A.1]) to access devices. If a certain function that was implemented in the user mode requires kernel performance (the interrupt handler, for example), that function is moved to the WinDriver Kernel PlugIn. Generally it should be possible to move code that uses WDC_xxx / WD_xxx function calls from the user mode to the kernel without modification, since the same WinDriver API is supported both in the user mode and in the Kernel PlugIn.

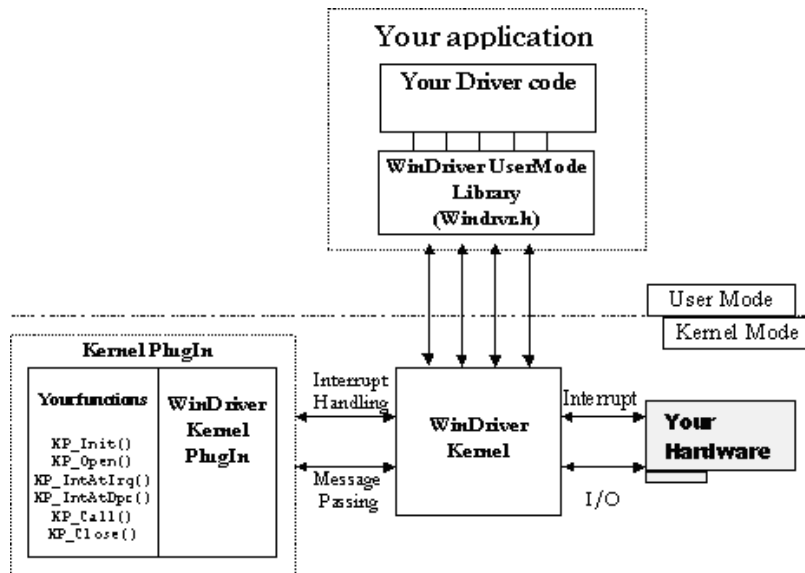


Figure 11.1: Kernel PlugIn Architecture

11.5.2 WinDriver's Kernel and Kernel PlugIn Interaction

There are two types of interaction between the WinDriver kernel and the WinDriver Kernel PlugIn:

Interrupt handling: When WinDriver receives an interrupt, by default it will activate the caller's user-mode interrupt handler. However, if the interrupt was

set to be handled by a Kernel PlugIn driver, then once WinDriver receives the interrupt, it activates the Kernel PlugIn driver's kernel-mode interrupt handler. Your Kernel PlugIn interrupt handler could essentially be comprised of the same code that you wrote and debugged in the user-mode interrupt handler, before moving to the Kernel PlugIn, although some of the user-mode code should be modified. We recommend you rewrite the interrupt acknowledge and handling code in the Kernel PlugIn to utilize the flexibility offered by Kernel the PlugIn (see section 11.6.5).

Message passing: To execute functions in kernel mode (such as I/O processing functions), the user-mode driver simply passes a message to the WinDriver Kernel PlugIn. The message is mapped to a specific function, which is then executed in the kernel. This function can typically contain the same code as it did when it was written and debugged in user mode. You can also use messages to pass data from the user-mode application to the Kernel PlugIn driver.

11.5.3 Kernel PlugIn Components

At the end of your Kernel PlugIn development cycle, your driver will have the following components:

- User-mode driver application (<**application name**>/.exe), written with the WDC_xxx / WD_xxx API
- The WinDriver kernel module – **windrvr6/.sys/.o**
- Kernel PlugIn driver (<**Kernel PlugIn driver name**>/.sys/.o), which was also written with the WDC_xxx / WD_xxx API and contains the driver functionality that you have selected to bring down to the kernel level

11.5.4 Kernel PlugIn Event Sequence

The following is a typical event sequence that covers all the functions that you can implement in your Kernel PlugIn:

11.5.4.1 Opening Handle from the User Mode to a Kernel PlugIn Driver

Event/Callback	Notes
Event: Windows loads your Kernel PlugIn driver.	This takes place at boot time, by dynamic loading, or as instructed by the registry.
Callback: Your <code>KP_Init()</code> Kernel PlugIn routine [A.9.2] is called	<code>KP_Init()</code> informs WinDriver of the name of your <code>KP_Open()</code> routine [A.9.2]. WinDriver will call this routine when the application wishes to open your driver – when it calls <code>WDC_xxxDeviceOpen()</code> (PCI: [A.2.8], PCMCIA: [A.2.9], ISA: [A.2.10]) with the name of a Kernel PlugIn driver to open, or when it calls the low-level <code>WD_KernelPlugInOpen()</code> function [A.8.1] (which is called by the wrapper <code>WDC_xxxDeviceOpen()</code> functions)
Event: Your user-mode driver application calls <code>WDC_xxxDeviceOpen()</code> (PCI: [A.2.8], PCMCIA: [A.2.9], ISA: [A.2.10]) with the name of a Kernel PlugIn driver to open, or it calls the low-level <code>WD_KernelPlugInOpen()</code> function [A.8.1] (which is called by the wrapper <code>WDC_xxxDeviceOpen()</code> functions)	
Callback: Your <code>KP_Open()</code> Kernel PlugIn routine [A.9.2] is called.	The <code>KP_Open()</code> function [A.9.2] is used to inform WinDriver of the names of all the callback functions that you have implemented in your Kernel PlugIn driver and to initiate the Kernel PlugIn driver, if needed.

11.5.4.2 Handling User-Mode Requests from the Kernel PlugIn

Event/Callback	Notes
Event: Your application calls <code>WDC_CallKerPlug()</code> [A.2.15], or the lower-level <code>WD_KernelPlugInCall()</code> function [A.8.3].	Your application calls <code>WDC_CallKerPlug()</code> / <code>WD_KernelPlugInCall()</code> to execute code in the kernel mode (in the Kernel PlugIn driver). The application passes a message to the Kernel PlugIn driver. The Kernel PlugIn driver will select the code to execute according to the message sent.
Callback: Your <code>KP_Call()</code> Kernel PlugIn routine [A.9.4] is called.	<code>KP_Call()</code> [A.9.4] executes code according to the message passed to it from the user mode.

11.5.4.3 Interrupt Handling – Enable/Disable and High Interrupt Request Level Processing

Event/Callback	Notes
Event: Your application calls <code>WDC_IntEnable()</code> [A.2.41] with the <code>fUseKP</code> parameter set to <code>TRUE</code> (after having opened the device with a Kernel PlugIn), or calls the lower-level <code>InterruptEnable()</code> [A.4.21] or <code>WD_IntEnable()</code> [A.5.2] functions with a handle to a Kernel PlugIn driver (set in the <code>hKernelPlugIn</code> field of the <code>WD_INTERRUPT</code> structure passed to the function).	
Callback: Your <code>KP_IntEnable()</code> Kernel PlugIn routine [A.9.6] is called	This function should contain any initialization required for your Kerne PlugIn interrupt handling.
Event: Your hardware creates an interrupt.	
Callback: Your <code>KP_IntAtIrql()</code> Kernel PlugIn routine [A.9.8] is called.	<code>KP_IntAtIrql()</code> runs at a high priority, and therefore should perform only the basic interrupt handling (such as lowering the HW interrupt signal of level sensitive interrupts in order to acknowledge the interrupt). If more interrupt processing is needed, <code>KP_IntAtIrql()</code> can return <code>TRUE</code> in order to defer additional processing to the <code>KP_IntAtDpc()</code> function [A.9.9].

Event/Callback	Notes
Event: Your application calls <code>WDC_IntDisable()</code> [A.2.42], or the lower-level <code>InterruptDisable()</code> [A.4.22] or <code>WD_IntDisable()</code> [A.5.5] functions, when the interrupts were previously enabled in the Kernel PlugIn (see the description of the interrupt enable event above.)	
Callback: Your <code>KP_IntDisable()</code> Kernel PlugIn routine [A.9.7] is called.	This function should free any memory that was allocated by the <code>KP_IntEnable()</code> [A.9.6] callback.

11.5.4.4 Interrupt Handling – Deferred Procedure Calls

Event/Callback	Notes
Event: The Kernel PlugIn <code>KP_IntAtIrql()</code> function [A.9.8] returns <code>TRUE</code> .	This informs WinDriver that additional interrupt processing as a deferred procedure call in the kernel.
Callback: Your <code>KP_IntAtDpc()</code> Kernel PlugIn routine [A.9.9] is called.	Processes the rest of the interrupt code, but at a lower priority than <code>KP_IntAtIrql()</code> .
Event: <code>KP_IntAtDpc()</code> [A.9.9] returns a value greater than 0.	This informs WinDriver that additional user-mode interrupt processing is also required.
Callback: <code>WD_IntWait()</code> [A.5.3] returns.	Your user-mode interrupt handler routine is executed.

11.5.4.5 Plug and Play and Power Management Events

Event/Callback	Notes
Event: Your application registers to receive Plug and Play and power management notifications using a Kernel PlugIn driver, by calling <code>WDC_EventRegister()</code> [A.2.44] with the <code>fUseKP</code> parameter set to <code>TRUE</code> (after having opened the device with a Kernel PlugIn), or calls the lower-level <code>EventRegister()</code> [A.6.2] or <code>WD_EventRegister()</code> functions with a handle to a Kernel PlugIn driver (set in the <code>hKernelPlugIn</code> field of the <code>WD_EVENT</code> structure that is passed to the function).	
Event: A Plug and Play or power management event (to which the application registered to listen) occurs.	

Event/Callback	Notes
Callback: Your <code>KP_Event()</code> Kernel PlugIn routine [A.9.5] is called.	<code>KP_Event()</code> receives information about the event that occurred and can proceed to handle it as needed.
Event: <code>KP_Event()</code> [A.9.5] returns <code>TRUE</code> .	This informs WinDriver that the event also requires user-mode handling.
Callback: <code>WD_IntWait()</code> [A.5.3] returns.	Your user-mode event handler routine is executed.

11.6 How Does Kernel PlugIn Work?

The following sections take you through the development cycle of a Kernel PlugIn driver.

It is recommended that you first write and debug your entire driver code in user mode, and then if you encounter performance problems or require greater flexibility, porting portions of your code to a Kernel PlugIn driver.

11.6.1 Minimal Requirements for Creating a Kernel PlugIn Driver

To build a Kernel PlugIn driver you need the following tools:

- On **Windows 98/Me/NT/2000/XP/Server 2003**:
 - The Visual C (VC) compiler (**cl.exe**, **rc.exe**, **link.exe** and **nm.exe**).
 - Install the Windows Device Driver Development Kit (DDK) for your target operating system on your host machine

NOTES

- The Windows DDKs are available as part of the Microsoft Development Network (MSDN) subscription. You can also order them from <http://www.microsoft.com/whdc/ddk/winddk.msp>
- Development of a Kernel PlugIn driver for a **Windows 98/Me** target PC should be done on a Windows 2000 or above host platform

- On **Linux** and **Solaris**: You need **GCC**, **gmake** or **make**.

NOTE

While this is not a minimal requirement, when developing a Kernel PlugIn driver it is highly recommended that you use two computers: set up one computer as your host platform and the other as your target platform. The host computer is the computer on which you develop your driver and the target computer is the computer on which you run and test the driver you develop

11.6.2 Kernel PlugIn Implementation

11.6.2.1 Before You Begin

The functions described in this section are callback functions, implemented in the Kernel PlugIn driver, which are called when their calling event occurs – see section 11.5.4 for details. For example, `KP_Init()` [A.9.1] is the callback function that is called when the driver is loaded and should include any code that you want to execute upon loading.

The name of your driver is given in `KP_Init()`, which must be implemented with this name. For the other callback functions, it is the convention of this reference guide to mark these functions as `KP_xxx()` functions (e.g. `KP_Open()`). However, when developing your Kernel PlugIn driver you can also select different names for these callback functions. When generating Kernel PlugIn code with the DriverWizard, for example, the names of the callback functions (apart from `KP_Init()`) comply to the following format: **KP_<Driver Name>_<Callback Function>**. For example, if you named your project **MyDevice** the name of your Kernel PlugIn `KP_Open()` function will be `KP_MyDevice_Open()`.

11.6.2.2 Write Your KP_Init() Function

Your `KP_Init()` function [A.9.1] should be of the following prototype:

```
BOOL __cdecl KP_Init(KP_INIT {*} kpInit);
```

where `KP_INIT` is the following structure:

```
typedef struct {
    DWORD dwVerWD;           /* Version of the WinDriver Kernel PlugIn library */
    CHAR cDriverName[12];    /* The Kernel PlugIn driver name (up to 8 chars) */
    KP_FUNC_OPEN funcOpen;   /* The Kernel PlugIn driver's KP_Open() function */
} KP_INIT;
```

This function is called once, when the driver is loaded. The `KP_INIT` structure should be filled with the name of your Kernel PlugIn and the address of your `KP_Open()` function [A.9.2] (see example in **WinDriver/samples/pci_diag/kp_pci/kp_pci.c**).

NOTES

- The name that you select for your Kernel PlugIn driver – by setting it in the `cDriverName` field of the `KP_INIT` structure in `KP_Init()` [A.9.1] – should be the name of the driver that you wish to create – i.e., if you are creating a driver called **XXX.sys**, you should set the name "XXX" in the `cDriverName` field of the `KP_INIT` structure.
- You should verify that the driver name that is set in the user mode, in the call to `WDC_xxxDeviceOpen()` (PCI: [A.2.8] / PCMCIA: [A.2.9] / ISA: [A.2.10]) or in the `pcDriverName` field of the `WD_KERNEL_PLUGIN` structure that is passed to `WD_KernelPlugInOpen()` [A.8.1] (when not using the WDC library), is identical to the driver name that was set in the `cDriverName` field of the `KP_INIT` structure that is passed to `KP_Init()`.
The best way to implement this without errors is to define the Kernel PlugIn driver name in a header file that is shared by the user-mode application and the Kernel PlugIn driver and use the defined value in all relevant locations.

From the **KP_PCI** sample (**WinDriver/samples/pci_diag/kp_pci/kp_pci.c**):

```

BOOL __cdecl KP_Init(KP_INIT *kpInit)
{
    /* Verify that the version of the WinDriver Kernel PlugIn library
       is identical to that of the windrvr.h and wd_kp.h files */
    if (WD_VER != kpInit->dwVerWD)
    {
        /* Re-build your Kernel PlugIn driver project with the compatible
           version of the WinDriver Kernel PlugIn library (kp_nt.lib)
           and windrvr.h and wd_kp.h files */

        return FALSE;
    }

    kpInit->funcOpen = KP_PCI_Open;
    strcpy (kpInit->cDriverName, KP_PCI_DRIVER_NAME);

    return TRUE;
}

```

Note that the driver name was set using a preprocessor definition. This definition is found in the **/WinDriver/samples/pci_diag/pci_lib.h** header file, which is shared by the **pci_diag** user-mode application and the **KP_PCI** Kernel PlugIn driver:

```

/* Kernel PlugIn driver name (should be no more than 8 characters) */

```

```
#define KP_PCI_DRIVER_NAME "KP_PCI"
```

11.6.2.3 Write Your KP_Open() Function

Your `KP_Open()` function [A.9.2] should be of the following prototype:

```
BOOL __cdecl KP_Open(KP_OPEN_CALL {*} kpOpenCall, HANDLE hWD,  
    PVOID pOpenData, PVOID {*} ppDrvContext);
```

This callback is called when the user-mode application calls `WDC_xxxDeviceOpen()` (PCI: [A.2.8], PCMCIA: [A.2.9], ISA: [A.2.10]) with the name of a Kernel PlugIn driver, or when it calls the lower-level `WD_KernelPlugInOpen()` function [A.8.1] (which is called by the wrapper `WDC_xxxDeviceOpen()` functions).

In the `KP_Open()` function, define the callbacks that you wish to implement in the Kernel PlugIn.

The following is a list of the callbacks that can be implemented:

Callback	Functionality
KP_Close() [A.9.3]	Called when the user-mode application calls WDC_xxxDeviceClose() (PCI: [A.2.11], PCMCIA: [A.2.12], ISA: [A.2.13]) for a device that was opened with a Kernel PlugIn driver, or when it calls the lower-level WD_KernelPlugInClose() function [A.8.2] (which is called by the wrapper WDC_xxxDeviceClose() functions).
KP_Call() [A.9.4]	Called when the user-mode application calls the WDC_CallKerPlug() function [A.2.15] or the lower-level WD_KernelPlugInCall() [A.8.3] function (which is called by the wrapper WDC_CallKerPlug() function.) This function implements a Kernel PlugIn message handler.
KP_IntEnable() [A.9.6]	Called when the user-mode application enables Kernel PlugIn interrupts, by calling WDC_IntEnable() with the fUseKP parameter set to TRUE (after having opened the device with a Kernel PlugIn), or by calling the lower-level InterruptEnable() [A.4.21] or WD_IntEnable() [A.2.41] functions with a handle to a Kernel PlugIn driver (set in the hKernelPlugIn field of the WD_INTERRUPT structure that is passed to the function). This function should contain any initialization required for your Kernel PlugIn interrupt handling.
KP_IntDisable() [A.9.7]	Called when the user-mode application calls WDC_IntDisable() [A.2.42], or the lower-level InterruptDisable() [A.4.22] or WD_IntDisable() [A.5.5] functions, if the interrupts were previously enabled with a Kernel PlugIn driver (see the description of KP_IntEnable() above.) This function should free any memory that was allocated by the KP_IntEnable() [A.9.6] callback.
KP_IntAtIrql() [A.9.8]	Called when WinDriver receives an interrupt (provided the interrupts were enabled with a handle to the Kernel PlugIn). This is the function that will handle your interrupt in the kernel mode. The function runs at high interrupt request level. Additional deferred processing can be performed in KP_IntAtDpc() and also in the user mode (see below.)

Callback	Functionality
KP_IntAtDpc() [A.9.9]	Called if the KP_IntAtIrql() callback [A.9.8] has requested deferred handling of the interrupt by returning TRUE. This function should include lower-priority kernel-mode interrupt handler code. The return value of this function determines the amount of times that the application's user-mode interrupt handler routine will be invoked (if at all).
KP_Event() [A.9.5]	Called when a Plug and Play or power management event occurs, provided the user-mode application previously registered to receive notifications for this event in the Kernel PlugIn by calling WDC_EventRegister() [A.2.44] with the fUseKP parameter set to TRUE (after having opened the device with a Kernel PlugIn), or by calling the lower-level EventRegister() [A.6.2] or WD_EventRegister() functions with a handle to a Kernel PlugIn driver (set in the hKernelPlugIn field of the WD_EVENT structure that is passed to the function).

As indicated above, these handlers will be called (respectively) when the user-mode program opens/closes a Kernel PlugIn driver (using WDC_xxxDeviceOpen() / WD_KernelPlugInOpen(), WDC_xxxDeviceClose() / WD_KernelPlugInClose()), sends a message to the Kernel PlugIn driver (by calling WDC_CallKerPlug() / (WD_KernelPlugInCall()), enables interrupts with a Kernel PlugIn driver (by calling WDC_IntEnable() with the fUseKP parameter set to TRUE, after having opened the device with a Kernel PlugIn / calling InterruptEnable() or WD_InterruptEnable() with a handle to the Kernel PlugIn set in the hKernelPlugIn() field of the WD_INTERRUPT structure that is passed to function), or disables interrupts (WDC_IntDisable() / InterruptDisable() / WD_IntDisable() that have been enabled using a Kernel PlugIn driver; The Kernel PlugIn interrupt handlers will be called when an interrupt occurs, if the interrupts were enabled using a Kernel PlugIn driver (see above.)

The Kernel PlugIn event handler will be called when a Plug and Play or power management event occurs, if the application registered to receive notifications for the event that occurred using a Kernel PlugIn driver (by calling WDC_EventRegister() with the fUseKP parameter set to TRUE, after having opened the device with a Kernel PlugIn / calling EventRegister() or WD_EventRegister() with a handle to a Kernel PlugIn driver set in the hKernelPlugIn field of the WD_EVENT structure that is passed to the function).

In addition to defining the Kernel PlugIn callback functions, you can implement code to perform any required initialization for the Kernel PlugIn in KP_Open(). In the sample **KP_PCI** driver and in the generated DriverWizard Kernel PlugIn driver,

for example, `KP_Open()` also calls the shared library's initialization function and allocates memory for the Kernel PlugIn driver context, which is then used to store the device information that was passed to the function from the user mode.

From the **KP_PCI** sample (**WinDriver/samples/pci_diag/kp_pci/kp_pci.c**):

```
/* KP_PCI_Open is called when WD_KernelPlugInOpen() is called from the
   user mode. */
/* pDrvContext will be passed to rest of the Kernel PlugIn callback functions. */
BOOL __cdecl KP_PCI_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
    PVOID pOpenData, PVOID *ppDrvContext)
{
    DWORD dwStatus;

    KP_PCI_Trace("KP_PCI_Open entered\n");

    kpOpenCall->funcClose = KP_PCI_Close;
    kpOpenCall->funcCall = KP_PCI_Call;
    kpOpenCall->funcIntEnable = KP_PCI_IntEnable;
    kpOpenCall->funcIntDisable = KP_PCI_IntDisable;
    kpOpenCall->funcIntAtIrql = KP_PCI_IntAtIrql;
    kpOpenCall->funcIntAtDpc = KP_PCI_IntAtDpc;
    kpOpenCall->funcEvent = KP_PCI_Event;

    /* Initialize the PCI library */
    dwStatus = PCI_LibInit();
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        KP_PCI_Err("KP_PCI_Open: Failed to initialize the PCI library: %s",
            PCI_GetLastError());
        return FALSE;
    }

    /* Allocate memory to hold the device information in the driver context */
    *ppDrvContext = malloc(sizeof(WDC_DEVICE));
    if (!*ppDrvContext)
    {
        KP_PCI_Err("KP_PCI_Open: Failed allocating memory for the driver context\n");
        PCI_LibUninit();
        return FALSE;
    }

    /* Copy the device information passed from the user mode to the
       Kernel PlugIn driver context */
}
```

```

COPY_FROM_USER(*ppDrvContext, *((PWDC_DEVICE*)pOpenData), sizeof(WDC_DEVICE));

KP_PCI_Trace("KP_PCI_Open: Kernel PlugIn driver opened successfully\n");

return TRUE;
}

```

11.6.2.4 Write the Remaining PlugIn Callbacks

Implement the remaining Kernel PlugIn routines that you wish to use (such as the `KP_Intxxx()` functions – for handling interrupts, or `KP_Event()` – for handling Plug and Play and power management events.)

11.6.3 Sample/Generated Kernel PlugIn Driver Code Overview

You can use the **DriverWizard** to generate a skeletal Kernel PlugIn driver for your device, and use it as the basis for your Kernel PlugIn driver development (recommended), or use the **Kernel PlugIn sample (KP_PCI)**, found under the **WinDriver/samples/pci_diag/kp_pci/** directory, as the basis for your Kernel PlugIn driver.

The Kernel PlugIn driver is not a stand-alone module. It requires a user-mode application that initiates the communication with the drive. A relevant application will be generated for you when using the DriverWizard to generate Kernel PlugIn code. The **pci_diag** application (found under the **/WinDriver/samples/pci_diag** directory) communicates with the sample **KP_PCI** driver.

Both the **KP_PCI** sample and the generated wizard code demonstrate communication between a user-mode application (**pci_diag.exe / xxx.exe** – where **xxx** is the name of the generated driver project) and a Kernel PlugIn driver (**kp_pci.sys/.o / xxx.sys/.o**).

The sample/generated code demonstrates how to pass data to the Kernel PlugIn's `KP_Open()` function and how to use this function to allocate and store a global Kernel PlugIn driver context, which can be used by other functions in the Kernel PlugIn.

The sample/generated Kernel PlugIn code implements a message for getting the driver's version number, in order to demonstrate how to initiate specific functionality in the Kernel PlugIn from the user mode and how to pass data between the Kernel PlugIn driver and a user-mode WinDriver application via messages.

The sample/generated code also demonstrates how to handle interrupts in the Kernel PlugIn.

The Kernel PlugIn implements an interrupt counter. The Kernel PlugIn interrupt handler performs deferred processing and notifies the user-mode application of the arrival of the interrupt for every fifth incoming interrupt.

The **KP_PCI** sample demonstrates PCI interrupt handling. However, as indicated in the comments of the sample `KP_IntAtIrql()` function, you will need to modify this function in order to implement the correct code for acknowledging the interrupt on your specific device, since interrupt acknowledgment is hardware-specific.

The generated DriverWizard code will include sample interrupt code for the selected device (PCI/PCMCIA/ISA). The generated `Kp_IntAtIrql()` function will include code to implement the interrupt transfer commands that you defined in the wizard (by assigning registers read/write commands to the card's interrupt in the **Interrupt** tab, if indeed such commands were defined. For PCI and PCMCIA interrupts, which need to be acknowledged in the kernel when the interrupt is received (see section 9.2.2), it is recommended that you use the wizard to define the commands for acknowledging (clearing) the interrupt, before generating the Kernel PlugIn code, so that the generated code will already include the required code for executing the commands you defined.

In addition, the sample/generated code demonstrates how to receive notifications of Plug and Play and power management events in the Kernel PlugIn.

TIP

We recommend that you build and run the sample/generated Kernel PlugIn project (and corresponding user-mode application) "as-is" before modifying the code or writing your own Kernel PlugIn driver. (Note, however, that as explained above, you cannot use the generic **KP_PCI** driver to handle the interrupts on your specific PCI device without first modifying the code in order to implement the required interrupt acknowledged for your device.)

11.6.4 Kernel PlugIn Sample/Generated Code Directory Structure

11.6.4.1 pci_diag and kp_pci Sample Directories

The Kernel PlugIn sample code – **KP_PCI** – is implemented in the `kp_pci.c` file. This sample driver is part of the WinDriver PCI diagnostics sample – **pci_diag** – which includes, in addition to the **KP_PCI** driver, a user-mode application that communicates with the driver (**pci_diag**) and a shared library that includes API that can be utilized by both the user-mode application and the Kernel PlugIn driver. The source files for this sample are implemented in C.

Following is an outline of the files found in the `/WinDriver/pci_diag/` directory:

- **kp_pci/** – Contains the **KP_PCI** Kernel PlugIn driver files:
 - **kp_xxx.c**: The source code of the **KP_XXX** driver
 - Project and/or make files and related files for building the Kernel PlugIn driver
 - **WINNT/kp_pci.sys**: A pre-compiled version of the **KP_PCI** Kernel PlugIn driver for Windows, built with the WinNT DDK
- **pci_lib.c**: Implementation of a library for accessing PCI devices using WinDriver's WDC API [A.1].
The library's API is used both by the user-mode application (**pci_diag**) and by the Kernel PlugIn driver (**kp_pci.c**).
- **pci_lib.h**: Header file for the **pci_lib** library
- **pci_diag.c**: Implementation of a sample diagnostics user-mode console (CUI) application, which demonstrates communication with a PCI device using the **pci_lib** and **WDC** libraries.
The sample also demonstrates how to communicate with a Kernel PlugIn driver from a user-mode WinDriver application. By default, the sample attempts to open the selected PCI device with a handle to the **kp_pci.c** Kernel PlugIn driver. If successful, the sample demonstrates how to interact with a Kernel PlugIn driver, as detailed in section 11.6.3. If the application fails to open a handle to the Kernel PlugIn driver, all communication with the device is performed from the user mode.
- **pci.inf**: A sample WinDriver PCI INF file for Windows 98/Me/2000/XP/Server 2003. NOTE: To use this file, change the vendor and device IDs in the file to comply with those of your specific device.
- Project and/or make files for building the **pci_diag** user-mode application
- A pre-compiled version of the user-mode application (**pci_diag**) for your target operating system
- **files.txt**: A list of the sample **pci_diag** files
- **readme.txt**: An overview of the sample Kernel PlugIn driver and user-mode application and instructions for building and testing the code

11.6.4.2 The Generated DriverWizard Kernel PlugIn Directory

The generated DriverWizard Kernel PlugIn code for your device will include a kernel-mode Kernel PlugIn project and a user-mode application that communicates with it. As opposed to the generic **KP_PCI** and **pci_diag** sample, the generated wizard code will utilize the resources information detected or defined for your

specific device, as well as any device-specific information that you define in the wizard before generating the code.

As indicated in section 11.6.3, when using the driver to handle PCI or PCMCIA interrupts, it is highly recommended that you define the registers that need to be read/written to acknowledge the interrupt, and set up the relevant read/write commands from/to these registers in the DriverWizard, before generating the code, in order to allow the generated interrupt handler code to utilize the hardware-specific information that you defined.

Following is an outline of the generated DriverWizard files when selecting to generate Kernel PlugIn code (where **xxx** represents the name that you selected for the driver when generating the code and **kp_xxx** is the directory in which you selected to save the code):

- **kernelmode/** – Contains the **KP_XXX** Kernel PlugIn driver files:
 - **kp_xxx.c**: The source code of the **KP_XXX** driver
 - Project and/or make files and related files for building the Kernel PlugIn driver
- **xxx_diag.c**: The source code of the generated user-mode **xxx_diag** application
- **xxx_lib.c**: Implementation of a library for accessing your device using WinDriver's WDC API [A.1].
The library's API is used both by the user-mode application (**xxx_diag**) and by the Kernel PlugIn driver (**KP_XXX**).
- **xxx_lib.h**: Header file for the **xxx_lib** library
- Project and/or make files for building the user-mode application
- Workspace/solution files that include both the Kernel PlugIn driver and user-mode application projects (for MSDEV)
- **xxx_files.txt**: A list of the generated files and instructions for building the generated code
- **xxx.inf**: A WinDriver INF file for your device (relevant only for Plug and Play devices, such as PCI or PCMCIA, on Windows 98/Me/2000/XP/Server 2003)

11.6.5 Handling Interrupts in the Kernel PlugIn

Interrupts will be handled in the Kernel PlugIn driver if enabled using a Kernel PlugIn driver, as explained below [11.6.5.2].

If Kernel PlugIn interrupts were enabled, when WinDriver receives a hardware interrupt, it calls the Kernel PlugIn driver's `KP_IntAtIrql()` function [A.9.8]. If

`KP_IntAtIrql()` returns `TRUE`, the deferred `KP_IntAtDpc()` Kernel PlugIn function [A.9.9] will be called, after `KP_IntAtIrql()` completes its processing and returns `TRUE`. The return value of `KP_IntAtDpc()` determines how many times (if at all) the user-mode interrupt handler routine will be executed.

In the **KP_PCI** sample, for example, the Kernel PlugIn interrupt handler code counts five interrupts and notifies the user mode on every fifth interrupt, thus `WD_IntWait()` [A.5.3] (in the user mode) will return on only one out of every five incoming interrupts. [`KP_IntAtIrql()` returns `TRUE` every five interrupts to activate `KP_IntAtDpc()`, and `KP_IntAtDpc()` returns the number of accumulated deferred DPC calls from `KP_IntAtIrql()`, so all at all the user-mode interrupt handler will be executed once for every 5 interrupts.]

11.6.5.1 Interrupt Handling in User Mode (Without Kernel PlugIn)

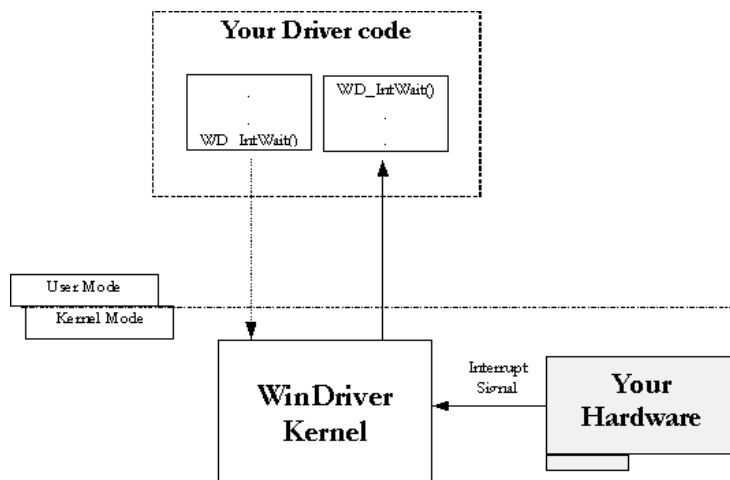


Figure 11.2: Interrupt Handling Without Kernel PlugIn

If the Kernel PlugIn interrupt handle is not enabled, then each incoming interrupt will cause `WD_IntWait()` to return and your user-mode interrupt handler routine will be invoked once WinDriver completes the kernel processing of the interrupts (mainly executing the interrupt transfer commands passed in the call to `WDC_IntEnable()` [A.2.41] or the lower-level `InterruptEnable()` [A.4.21] or `WD_IntEnable()` [A.5.2] functions) – see Figure 11.2.

11.6.5.2 Interrupt Handling in the Kernel (Using a Kernel PlugIn)

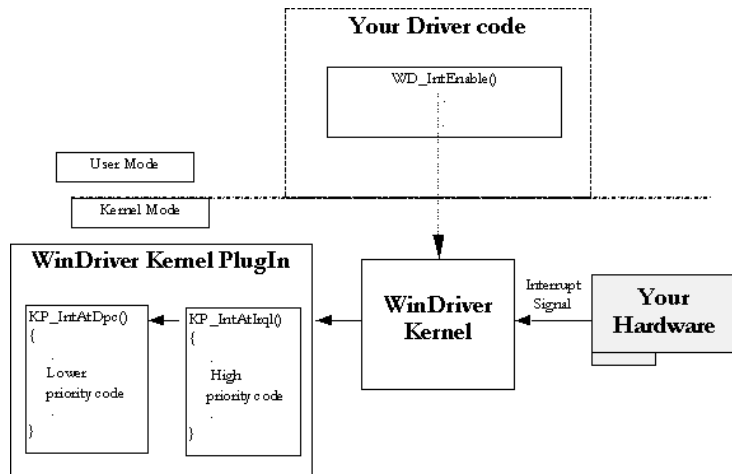


Figure 11.3: Interrupt Handling With the Kernel PlugIn

To have the interrupts handled by the Kernel PlugIn, the user-mode application should open a handle to the device with a Kernel PlugIn driver, by passing the name of a Kernel PlugIn driver to the `WDC_xxxDeviceOpen()` function (PCI: [A.2.8], PCMCIA: [A.2.9], ISA: [A.2.10]), and then call `WDC_IntEnable()` [A.2.41] with the `fUseKP` parameter set to `TRUE`.

If you are not using the `WDC_xxx` API [A.1], your application should pass a handle to the Kernel PlugIn driver to the `WD_IntEnable()` [A.5.2] function [A.5.2] or the wrapper `InterruptEnable()` function [A.4.21] (which calls `WD_IntEnable()` and `WD_IntWait()` [A.5.3]). This enables the Kernel PlugIn interrupt handler. (The Kernel PlugIn handle is passed within the `hKernelPlugIn` field of the `WD_INTERRUPT` structure that is passed to the functions.)

When calling `WDC_IntEnable()/InterruptEnable()/WD_IntEnable()` to enable interrupts in the Kernel PlugIn, your Kernel PlugIn's `KP_IntEnable()` callback function [A.9.6] is activated. In this function you can set the interrupt context that will be passed to the Kernel PlugIn interrupt handlers, as well as write to the device to actually enable the interrupts in the hardware and implement any other code required in order to correctly enable your device's interrupts.

If the Kernel PlugIn interrupt handler is enabled, then `KP_IntAtIrql()` [A.9.8] for each incoming interrupt. The code in the `KP_IntAtIrql()` function is executed at high interrupt request level. While this code is running, the system is halted, i.e., there will be no context switches and no lower-priority interrupts will be handled.

The code in the `KP_IntAtIrql()` function is limited in the following ways:

- It may only access non-pageable memory.
- It may only call the following functions (or wrapper functions that called these functions):
 - `WDC_xxx` read/write address or configuration space functions
 - `WDC_MultiTransfer()` [A.2.22], `WD_Transfer()` [A.4.14], `WD_MultiTransfer()` [A.4.15] or `WD_DebugAdd()` [A.7.6]
 - Specific kernel OS functions (such as WinDDK functions) that can be called from high interrupt request level. (Note that the use of such functions may break the code's portability to other operating systems.)
- It may **not** call `malloc()`, `free()` or any `WDC_xxx` or `WD_xxx` API other than those listed above

Because of the aforementioned limitations, the code in `KP_IntAtIrql()` should be kept to a minimum, such as acknowledgment (clearing) of level sensitive interrupts. Other code that you want to run in the interrupt handler should be implemented in `KP_IntAtDpc()` [A.9.9], which runs at a deferred interrupt level and does not face the same limitations as `KP_IntAtIrql()`. `KP_IntAtDpc()` is called after `KP_IntAtIrql()` returns (provided it returns `TRUE`).

You can also leave some additional interrupt handling to the user mode. The return value of `KP_IntAtDpc()` [A.9.9] determines the amount of times (if any) that your user-mode interrupt handler routine will be called after the kernel-mode interrupt processing is completed.

11.6.6 Message Passing

The WinDriver architecture enables a kernel-mode function to be activated from the user mode by passing a message from the user mode to the Kernel PlugIn driver using `WDC_CallKerPlug()` [A.2.15] or the lower-level `WD_KernelPlugInCall()` function [A.8.3].

The messages are defined by the developer in a header file that is common to both the user-mode and kernel-mode plugin parts of the driver. In the **pci_diag KP_PCI** sample and the generated DriverWizard code, the messages are defined in the shared library header file – **pci_lib.h** in the sample or **xxx_lib.h** in the generated code.

Upon receiving the message from the user mode, WinDriver will execute the `KP_Call()` [A.9.4] Kernel PlugIn callback function, which identifies the message that has been received and executes the relevant code for this message (as implemented in the Kernel PlugIn).

The sample/generated Kernel PlugIn code implement a message for getting the driver's version in order to demonstrate Kernel PlugIn data passing. The code that sets the version number in `KP_Call()` is executed in the Kernel PlugIn whenever the Kernel PlugIn receives a relevant message from the user-mode application. You can see the definition of the message in the shared **pci_lib.h/xxx_lib.h** shared header file. The user-mode application (**pci_diag.exe/xxx_diag.exe**) sends the message to the Kernel PlugIn driver via the `WDC_CallKerPlug()` function [A.2.15].

Chapter 12

Writing a Kernel PlugIn

The easiest way to write a Kernel PlugIn driver is to use the **DriverWizard** to generate the Kernel PlugIn code for your hardware. Alternatively, you can use the sample PCI Kernel PlugIn driver provided with WinDriver – **KP_PCI** – which is part of the **pci_diag** sample and can be found under the **WinDriver/samples/pci_diag/kp_pci/** directory, as the basis for your Kernel PlugIn driver development.

The following is a step-by-step guide to creating your Kernel PlugIn driver.

12.1 Determine Whether a Kernel PlugIn is Needed

The Kernel PlugIn should be used only after your driver code has been written and debugged in user mode. This way, all of the logical problems of creating a device driver are solved in the user mode, where development and debugging are much easier.

Determine whether a Kernel PlugIn should be written by consulting Chapter 10, which explains how to improve the performance of your driver. In addition, the Kernel PlugIn affords greater flexibility, which is not always available when writing the driver in the user mode (specifically with regards to the interrupt handling.)

12.2 Prepare the User-Mode Source Code

1. Isolate the functions you need to move into the Kernel PlugIn.
2. Remove any platform-specific code from the functions. Use only functions that can also be used from the kernel.
3. Recompile your driver in the user mode.
4. Debug your driver in user mode again to see that your code still works after changes have been made.

NOTES

- Keep in mind that the kernel stack is relatively limited in size. Therefore, code that will be moved into the Kernel PlugIn should not contain static memory allocations. Use the `malloc()` function to allocate memory dynamically instead. This is especially important for large data structures.
- If the user-mode code that you are porting to the kernel accesses memory addresses directly - using the user-mode mapping of the physical address, returned from `WD_CardRegister()` - note that in the kernel you will need to use the kernel mapping of the physical address instead (the kernel mapping is also returned by `WD_CardRegister()` - see section [A.4.11](#)).
When using the API of the **WDC** library [\[A.1\]](#) to access memory, you do not need to worry about this, since this API ensures that the correct mapping of the memory is used depending on whether the relevant APIs are used from the user mode or from the kernel mode.

12.3 Create a New Kernel PlugIn Project

You can use the **DriverWizard** to generate a new Kernel PlugIn project (and corresponding user-mode project) for your device (recommended), or use the **KP_PCI** sample as the basis for your development. [You can also develop your code "from scratch", if you wish.]

If you choose to use the **KP_PCI** sample as the basis for your development, please follow these steps:

1. Make a copy of the **WinDriver/samples/pci_diag/kp_pci/** directory. For example, to create a new Kernel PlugIn project called **KP_MyDrv**, copy **WinDriver/samples/pci_diag/kp_pci/** to **/WinDriver/samples/MyDrv/**.
2. Change all instances of "KP_PCI" and "kp_pci" in all the Kernel PlugIn files in your new directory to "KP_MyDrv" and "kp_MyDrv" (respectively.) [Note:

The names of the `KP_PCI_xxx()` functions in the **kp_pci.c** files do not have to be changed in order for the code to function correctly, although the code will be clearer if you use your driver's name in the function names.]

3. Change all occurrences of "KP_PCI" in file names to "kp_MyDrv".
4. To use the shared **pci_lib** library API from your Kernel PlugIn driver and user-mode application, copy the **pci_lib.h** and **pci_lib.c** files from the **/WinDriver/samples/pci_diag/** directory to your new **MyDrv/** directory. You can change the names of the library functions to use your driver's name (**MyDrv**) instead of "PCI", but note that in this case you will also need to modify the names in all calls to these functions from your Kernel PlugIn project and user-mode application.
If you do not copy the shared library to your new project, you will need to modify the sample Kernel PlugIn code and replace all references to the `PCI_xxx` library APIs with alternative code.
5. Modify the files and directory paths in the project and make files and the `#include` paths in the source files as needed (depending on the location in which you selected to save your new project directory.)
6. To use the **pci_diag** user-mode application, copy **/WinDriver/samples/pci_diag/pci_diag.c** and the relevant `pci_diag` project, workspace/solution or make files to your **MyDrv/** directory, rename the files (if you wish) and replace all "pci_diag" references in the files with your preferred user-mode application name. To use the workspace/solution files, also replace the references to "KP_PCI" in the files with your new Kernel PlugIn driver, e.g. "KP_MyDrv". Then modify the sample code to implement your desired driver functionality.

For a general description of the generated/sample Kernel PlugIn code and its structure, see sections 11.6.3 and 11.6.4.

12.4 Create a Handle to the Kernel PlugIn

In your user-mode application or library source code, call `WDC_PciDeviceOpen()` [A.2.8] / `WDC_PcmciaDeviceOpen()` [A.2.9] / `WDC_IsaDeviceOpen()` [A.2.10] (depending on the type of your device) with the name of your Kernel PlugIn driver in order to open a handle to the device using the Kernel PlugIn driver.

The generated **DriverWizard** and the sample **pci_diag** shared library (**xxx_lib.c** / **pci_lib.c**) demonstrate how this should be done – see the generated/sample `XXX_DeviceOpen()/PCI_DeviceOpen()` library function (which is called from the generated/sample **xxx_diag/pci_diag** user-mode application.)

If you are not using the WDC library from your code [A.1], you need to call `WD_KernelPlugInOpen()` [A.8.1] at the beginning of your code in order to open a handle to your Kernel PlugIn driver, and call `WD_KernelPlugInClose()` [A.8.2] before terminating the application or when you no longer wish to use the Kernel PlugIn driver. `WD_KernelPlugInOpen()` returns a handle to the Kernel PlugIn driver within the `hKernelPlugIn` field of the `WD_KERNEL_PLUGIN` structure that was passed to the function.

12.5 Set Interrupt Handling in the Kernel PlugIn

1. When calling `WDC_IntEnable()` [A.2.41] (after having opened a handle to the device using a Kernel PlugIn driver, by calling `WDC_xxxDeviceOpen()` with the name of a Kernel PlugIn driver, as explained in section 12.4), set the `fUseKP` function parameter to `TRUE` to indicate that you wish to enable interrupts in the Kernel PlugIn driver with which the device was opened. The generated DriverWizard and the sample **pci_diag** shared library (**xxx_lib.c** / **pci_lib.c**) demonstrate how this should be done – see the generated/sample `XXX_IntEnable()` / `PCI_IntEnable()` library function (which is called from the generated/sample **xxx_diag/pci_diag** user-mode application.)

If you are not using the `WDC_xxx` API [A.1], in order to enable interrupts in the Kernel PlugIn call `WD_IntEnable()` [A.5.2] or `InterruptEnable()` [A.4.21] (which calls `WD_IntEnable()`), and pass the handle to the Kernel PlugIn driver that you received from `WD_KernelPlugInOpen()` [A.8.1] (within the `hKernelPlugIn` field of the `WD_KERNEL_PLUGIN` structure that was passed to the function.)

2. When calling to `WDC_IntEnable()` / `InterruptEnable()` / `WD_IntEnable()` to enable interrupts in the Kernel PlugIn, WinDriver will activate your Kernel PlugIn's `KP_IntEnable()` callback function [A.9.6]. You can implement this function to set the interrupt context that will be passed to the Kernel PlugIn interrupt handlers (`KP_IntAtIrql()` / `KP_IntAtDpc()`), as well as write to the device to actually enable the interrupts in the hardware, for example, or implement any other code required in order to correctly enable your device's interrupts.
3. Move the implementation of the user-mode interrupt handler, or the relevant portions of this implementation, to the Kernel PlugIn's interrupt handler functions. High-priority code, such as the code for acknowledging (clearing) level sensitive interrupts, should be moved to the `KP_IntAtIrql()` function [A.9.8], which runs at high interrupt request level. Deferred processing of the interrupt can be moved to `KP_IntAtDpc()` [A.9.9], which will be executed once `KP_IntAtIrql()` completes its processing and returns `TRUE`. You can also

modify the code to make it more efficient, due to the advantages of handling the interrupts directly in the kernel, which provides you with greater flexibility (e.g. you can read from a specific register and write back the value that was read, toggle specific register bits, etc.). See Section 11.6.5 for an explanation of how to handle interrupts in the kernel using a Kernel PlugIn driver.

12.6 Set I/O Handling in the Kernel PlugIn

1. Move your I/O handling code (if needed) from the user mode to the Kernel PlugIn message handler – `KP_Call()` [A.9.4].
2. To activate the kernel code that performs the I/O handling from the user mode, call `WDC_CallKerPlug()` [A.2.15] or the lower-level `WD_KernelPlugInCall()` function [A.8.3] with a relevant message for each of the different functionality that you wish to perform in the Kernel PlugIn. Implement a different message for each functionality.
3. Define these messages in a header file that is shared by the user-mode application (which will send the messages) and the Kernel PlugIn driver (that implements the messages).
In the sample/generated DriverWizard Kernel PlugIn projects, the message IDs and other information that should be shared by the user-mode application and Kernel PlugIn drive are defined in the `pci_lib.h/xxx_lib.h` shared library header file.

12.7 Compile Your Kernel PlugIn Driver

12.7.1 On Windows

The sample `WinDriver\samples\pci_diag\kp_pci\` Kernel PlugIn directory and the generated DriverWizard Kernel PlugIn `base_dir\kermode\` directory (where `base_dir` is the directory in which you saved the generated driver project) contain Microsoft Developer Studio (MSDEV) 6.0 and/or MSDEV 7.0 (.NET) Kernel PlugIn project files (`.dsp` / `.vcproj`).

The sample `WinDriver\samples\pci_diag\` directory and the generated `base_dir\` directory have `msdev\` and/or `msdev_net\` sub-directories, which contain MSDEV 6.0 and/or MSDEV 7.0 (.NET) project files (`.dsp` / `.vcproj`), respectively, for the user-mode application that drives the respective Kernel PlugIn driver. The `msdev\` / `msdev_net\` sub-directories also contain MSDEV 6.0 and/or MSDEV 7.0 (.NET) workspace/solution files (`.dsw` / `.sln`), respectively, which include both the Kernel

PlugIn and the user-mode application project files and enable you to easily compile and build both projects and create your SYS Kernel PlugIn driver from the MSDEV 6.0 or 7.0 (.NET) IDE.

To build your Kernel PlugIn driver and respective user-mode application, follow these steps:

1. Verify that the Windows Device Development Kit (DDK) for your target operating system is installed on your PC (see section 11.6.1).
The target operating system is the operating system for which you wish to create your driver. For example, if you are creating a driver for Windows XP, install the Windows XP DDK.
You can install several DDKs, for different operating systems, and then rebuild your Kernel PlugIn driver for each target operating system that you wish to support.
2. Set the **BASEDIR** environment variable to point to the the location of the Windows DDK for your target platform.
For example, to build a Kernel PlugIn driver for Windows XP, set the **BASEDIR** environment variable to point to the directory in which the Windows XP DDK was installed.
3. Start Microsoft Developer Studio (MSDEV) and do the following:
 - (a) From your driver project directory, open the generated workspace/solution file – \base_dir\msdev\xxx_diag.dsw/.sln, where **base_dir** is your driver project directory (**pci_diag** for the sample code / the directory in which you selected to save the generated DriverWizard code; By default location is \WinDriver\wizard\my_projects), and **xxx** is the driver name (**pci** for the sample / the name you selected when generating the code with the wizard.)
Note that when selecting to generate code for the MSDEV IDE with the DriverWizard, the wizard automatically starts MSDEV and opens the generated workspace/solution file after generating the code files, unless you explicitly revoke this behavior by setting the "IDE to Invoke" in the code generation dialog to "None".
 - (b) To build the Kernel PlugIn SYS driver (**kp_pci.sys** / **kp_xxx.sys**):
 - i. Set the Kernel PlugIn project (**kp_pci.dsp/vcproj** / **kp_xxx.dsp/vcproj**) as the active project
 - ii. Select the active configuration for your target platform: From the **Build** menu, choose **Set Active Configuration...**, and select the desired configuration.

NOTE

The active configuration must correspond with the target OS for which you are building the driver. For example, for Windows 2000 select either **Win32 win2k free** (release mode) or **Win32 win2k checked** (debug mode).

- iii. Build your driver: Press the **F7** key or start the build process from the **Build** menu.
- (c) To build the user-mode application that drives the Kernel PlugIn driver (**pci_diag.exe** / **xxx_diag.exe**):
 - i. Set the user-mode project (**pci_diag.dsp/vcproj** / **xxx_diag.dsp/vcproj**) as the active project
 - ii. Build the application: Press the **F7** key or start the build process from the **Build** menu.

NOTE

On **Windows 98/Me**, the generated code cannot be built into a SYS driver using the method described above. You can, however, build a SYS driver for a target Windows 98/Me platform on Windows NT/2000/XP/Server 2003 – i.e. build the code on a PC running Windows NT/2K/XP/Server 2003, but set the **BASEDIR** environment variable to point to the Windows 98/Me DDK and set the build target in the Kernel PlugIn project for Windows 98/Me, and then use the driver that was created on Windows 98/Me.

12.7.2 On Linux

1. Open a shell terminal.
2. Change directory to your Kernel PlugIn directory.

For example, when compiling the sample **KP_PCI** driver, run:

```
cd WinDriver/samples/pci_diag/kp_pci/
```

When compiling the Kernel PlugIn driver for your generated DriverWizard Kernel PlugIn code, run the following command, where **<path>** represents the path to your generated DriverWizard project directory

(e.g. **/home/user/WinDriver/wizard/my_projects/my_kp/**):

```
cd <path>/kermode/linux/
```

3. Generate the **makefile** using the **configure** script:


```
./configure
```

NOTE

The **configure** script creates a **makefile** based on your specific running kernel. You may run the **configure** script based on another kernel source you have installed, by adding a flag **--with-kernel-source=<path>** to the configure script. The <path> is the full path to the kernel source directory.

4. Build the Kernel PlugIn module using the **make** command.
This command creates a new **LINUX.xxx/** directory (where **xxx** depends on the Linux kernel), which contains the created **kp_xxx.o/ko** driver.

5. Move to the directory that holds the makefile for the sample user-mode diagnostics application.

For the **KP_PCI** sample driver:

```
cd ../LINUX/
```

For the generated DriverWizard Kernel PlugIn driver:

```
cd ../../linux/
```

6. Compile the sample diagnostics program using the **make** command.

12.7.3 On Solaris

NOTE

WinDriver generates **makefiles** for GNU **make** utility only.

If you wish to use the standard **make** utility, instead of the GNU **make**, you must modify the **makefile** that WinDriver generates. The GNU **make** package is available from <http://www.sunfreeware.com>.

1. Open a shell terminal.
2. Change directory to your driver's project directory.

For example, when compiling the sample **KP_PCI** driver, run:

```
cd WinDriver/samples/pci_diag/
```

3. Build your Kernel PlugIn module using the **make** command.

For example, to build the sample **KP_PCI** driver, run:

```
make -C kp_pci/SOLARIS
```

To build your generated DriverWizard Kernel PlugIn driver, run:

```
make -C kermode/solaris
```

4. Build your sample diagnostics program by running:
make -C solaris

NOTE

For 64-bit kernels, both the Kernel PlugIn module and the user-mode application that drives it need to be compiled in 64-bit mode. The makefile provided by WinDriver uses the CC and LD environment variables without specifically declaring them. You may therefore need to set these variables to fit your specific compiler and linker with the corresponding flags.

For example, to compile with gcc you may need to set the CC and LD variables as follows:

For the compilation of the Kernel PlugIn module:

```
$ export LD="gcc -m64 -melf64_sparc -nostdlib"
```

```
$ export CC="gcc -m64 -isystem /usr/include/"
```

For the compilation of the user-mode application:

```
$ export LD="gcc -m64"
```

```
$ export CC="gcc -m64"
```

12.8 Install Your Kernel PlugIn Driver

12.8.1 On Win32 Platforms

1. Copy the files to the appropriate locations:
Windows 98/Me/NT/2000/XP/Server 2003: Copy the **MyDrv.sys** driver that was created to the **%windir%\system32\drivers** directory (e.g., **C:\WINNT\system32\drivers** – on WinNT/2000, or **C:\Windows\system32\drivers** – on Windows XP/Server2003).

NOTE

If your Kernel PlugIn driver is intended for Windows 98/Me, develop the driver on a Windows NT/2000/XP/Server 2003 host PC (see note in section 12.7).

2. Register/load your driver, using the **wdreg.exe** (or **wdreg_gui.exe**) utility – on Windows NT/2000/XP/Server 2003, or using the **wdreg16.exe** utility – on Windows 98/Me:

NOTES

- In the following instructions, 'KP_NAME' stands for your Kernel PlugIn driver's name, **without** the .sys extension.
- For Windows 98/Me, replace references to 'wdreg' below with 'wdreg16' (see section 13.2.2 for more information regarding the WDREG utility.)

To install your driver, run:

```
\WinDriver\util> wdreg -name KP_NAME install
```

NOTE

Kernel PlugIn drivers, with the exception of SYS drivers on Windows 98/Me, are dynamically loadable, and thus do not require a reboot in order to load.

12.8.2 On Linux

1. Change directory to your Kernel PlugIn driver directory.

For example, when installing the sample **KP_PCI** driver, run:

```
cd WinDriver/samples/pci_diag/kp_pci/
```

When installing a driver created using the Kernel PlugIn files generated by the DriverWizard, run the following command, where **<path>** represents the path to your generated DriverWizard project directory (e.g. **/home/user/WinDriver/wizard/my_projects/my_kp/**):

```
cd <path>/kermode/
```

2. Run the following command to install your Kernel PlugIn driver:
make install

12.8.3 On Solaris

NOTE

Installation of the Kernel PlugIn Driver should be performed by the system administrator logged in as root, or with root privileges (become a super user).

1. Change directory to your Kernel PlugIn driver's Solaris directory.

For example, when installing the sample **KP_PCI** driver, run:

```
cd WinDriver/samples/pci_diag/kp_pci/SOLARIS
```

When installing a driver created using the Kernel PlugIn files generated by the DriverWizard, run the following command, where **<path>**

represents the path to your generated DriverWizard project directory (e.g. `/home/user/WinDriver/wizard/my_projects/my_kp/`):

```
cd <path>/kermode/solaris
```

2. Copy the newly created driver to the drivers' directory.

For example to copy the sample **KP_PCI** driver, on 64-bit platforms run:

```
WinDriver/samples/pci_diag/kp_pci/solaris#
```

```
cp kp_pci /kernel/drv/sparcv9
```

and on 32-bit platforms run:

```
WinDriver/samples/pci_diag/kp_pci/solaris#
```

```
cp kp_pci /kernel/drv
```

3. Install the driver by running:

```
<Kernel PlugIn directory>/solaris# make install
```

NOTE

The following commands are also useful when installing a driver on Solaris:

- **modinfo** – lists the loaded kernel modules.
- **rem_drv** – removes the kernel module.

Chapter 13

Dynamically Loading Your Driver

13.1 Why Do You Need a Dynamically Loadable Driver?

When adding a new driver, you may be required to reboot the system in order for it to load your new driver into the system. WinDriver is a dynamically loadable driver, which enables your customers to start your application immediately after installing it, without needing to reboot. You can dynamically load your driver whether you have created a user-mode or a kernel-mode driver.

NOTE

In order to successfully UNLOAD your driver, make sure there are no open handles to the driver from WinDriver applications, from a Kernel PlugIn driver, or from connected Plug and Play devices that were registered with WinDriver using an INF file.

13.2 Windows NT/2000/XP/Server 2003 and 98/Me

13.2.1 Windows Driver Types

Windows drivers can be implemented as either of the following types, both of which are supported by the **wdreg** utility:

- WDM (Windows Driver Model) drivers: Files with the extension **.sys** on Win98/Me/2000/XP/Server 2003 (e.g. **windrvr6.sys**).
WDM drivers are installed via the installation of an INF file (see below).
- Non-WDM / Legacy drivers: These include drivers for non-Plug and Play Windows operating systems (Windows NT 4.0), files with the extension **.vxd** on Windows 98/Me, and all Kernel Plugin driver files (e.g. **windrvr6.vxd** ; **MyKPDriver.sys**).

NOTE

Starting from version 6.21 of WinDriver, **.vxd** drivers are no longer supported.

13.2.2 The WDREG Utility

WinDriver provides a utility for dynamically loading and unloading your driver, which replaces the slower manual process using Windows' Device Manager (which can still be used for the device INF). For **Windows 2000/XP/Server 2003**, this utility is provided in two forms: **wdreg** and **wdreg_gui**. Both utilities can be found under the **\WinDriver\util** directory, can be run from the command line, and provide the same functionality. The difference is that **wdreg_gui** displays installation messages graphically, while **wdreg** displays them in console mode.

For **Windows 98/Me** the **wdreg16** utility is provided.

This section describes the usage of **wdreg/ wdreg_gui/wdreg16** on Windows operating systems.

NOTES

(1) The explanations and examples below refer to **wdreg**, but for **Windows 2000/XP/Server 2003** you can replace any references to **wdreg** with **wdreg_gui**. For **Windows 98/Me**, replace the references to **wdreg** with **wdreg16**.

(2) On **Windows 98/Me** you can only use **wdreg16** to install the **windrvr6.sys** WDM driver (by installing **windrvr6.inf**) and Kernel PlugIn drivers, but you **cannot** use **wdreg16** to install any other INF files.

13.2.2.1 WDM Drivers

This section explains how to use the **wdreg** utility to install the WDM **windrvr6.sys** driver on Windows 98/Me/2000/XP/Server 2003, or to install INF files that register Plug and Play devices (such as PCI or PCMCIA) to work with this driver on Windows 2000/XP/Server 2003.

NOTES

(1) As specified above, on **Windows 98/Me** you can only use **wdreg16** to install the **windrvr6.sys** WDM driver, by installing **windrvr6.inf**, but you **cannot** use **wdreg16** to install any other INF files.

(2) This section is **not relevant** for **Kernel PlugIn** drivers, since these are not WDM drivers and are not installed via an INF file – see the previous section [13.2.2.2] for an explanation on how to use **wdreg** to install Kernel PlugIn drivers on Windows 98/Me/NT/2000/XP/Server 2003.

Usage: The **wdreg** utility can be used in two ways as demonstrated here:

1. **wdreg -inf <filename> [-silent] [-log <logfile>] [install | uninstall | enable | disable]**
 2. **wdreg -rescan <enumerator> [-silent] [-log <logfile>]**
- **OPTIONS**
wdreg supports several basic OPTIONS from which you can choose one, some, or none:
 - inf** – The path of the INF file to be dynamically installed.
 - rescan <enumerator>** – Rescan enumerator (ROOT, ACPI, PCI, etc.) for hardware changes. Only one enumerator can be specified.
 - silent** – Suppresses the display of messages of any kind. (Optional)
 - log <logfile>** – Logs all messages to the specified file. (Optional)
 - **ACTIONS**
wdreg supports several basic ACTIONS:
 - install** – Installs the INF file, copies the relevant files to their target locations, dynamically loads the driver specified in the INF file name by replacing the older version (if needed).
 - uninstall** – Removes your driver from the registry so that it will not load on next boot.
 - enable** – Enables your driver.

disable – Disables your driver, i.e. dynamically unloads it, but the driver will reload after system boot.

NOTE

In order to successfully disable/uninstall WinDriver, you must first close any open handles to the **windrvr6.sys** service. This includes closing any open WinDriver applications and uninstalling (from the Device Manager or using **wdreg**) any PCI/PCMCIA devices that are registered to work with the **windrvr6.sys** service (or otherwise removing such devices). **wdreg** will display a relevant warning message if you attempt to stop the **windrvr6.sys** when there are still open handles to the service, and will enable you to select whether to close all open handles and Retry, or Cancel and reboot the PC to complete the command's operation.

13.2.2.2 Non-WDM Drivers

This section explains how to use the **wdreg** utility to install non-WDM drivers, namely **windrvr6.sys** on Windows NT 4.0 and Kernel PlugIn drivers on Windows 98/Me/NT/2000/XP/Server 2003.

Usage: **WDREG** [-file <filename>] [-name <drivername>]
[-startup <level>] [-silent] [-log <logfile>] Action
[Action ...]

- **OPTIONS**

wdreg supports several basic OPTIONS from which you can choose one, some, or none:

- startup** : Specifies when to start the driver. Requires one of the following arguments:
 - **boot**: Indicates a driver started by the operating system loader, and should only be used for drivers that are essential to loading the OS (for example, Atdisk).
 - **system**: Indicates a driver started during OS initialization.
 - **automatic**: Indicates a driver started by the Service Control Manager during system startup.
 - **demand**: Indicates a driver started by the Service Control Manager on demand (i.e., when your device is plugged in).
 - **disabled**: Indicates a driver that cannot be started.

NOTE

The default setting for the **-startup** option is **automatic**.

-name – Relevant only for Kernel PlugIn drivers (by default the **wdreg** commands relate to the **windrvr6** service). Sets the symbolic name of the driver. This name is used by the user-mode application to get a handle to the driver. You must provide the driver's symbolic name (*without* the *.sys extension) as an argument with this option. The argument should be equivalent to the driver name as set in the `KP_Init()` [A.9.1] function of your Kernel PlugIn project: `strcpy(kpInit->cDriverName, XX_DRIVER_NAME)`.

-file – Relevant only for Kernel PlugIn. **wdreg** allows you to install your driver in the registry under a different name than the physical file name. This option sets the file name of the driver. You must provide the driver's file name (*without* the *.sys extension) as an argument.

wdreg looks for the driver in the Windows installation directory (`<WINDIR>\system32\drivers`). Therefore, you should verify that the driver file is located in the correct directory before attempting to install the driver.

Usage:

```
\> wdreg -name <Your new driver name> -file <Your original driver name> install
```

-silent – Suppresses the display of messages of any kind.

-log <logfile> – Logs all messages to the specified file.

- **ACTIONS**

wdreg supports several basic ACTIONS:

create – Instructs Windows to load your driver next time it boots, by adding your driver to the registry.

delete – Removes your driver from the registry so that it will not load on next boot.

start – Dynamically loads your driver into memory for use. You must create your driver before starting it.

stop – Dynamically unloads your driver from memory.

NOTE

In order to successfully stop the **windrvr6.sys** service, you must first close any open handles to the this service (such as closing open WinDriver applications). **wdreg** will display a relevant warning message if you attempt to stop the service when there are still open handles to it.

- **Shortcuts**

wdreg supports a few shortcut operations for your convenience:

install – Creates and starts your driver.

This is the same as first using the **wdreg stop** action and then the **wdreg start** action.

(if an older version exists), or:

The same as using the **wdreg create** action and then the **wdreg start** action.

(otherwise).

uninstall – Unloads your driver from memory and removes it from the registry so that it will not load on next boot.

This is the same as first using the **wdreg stop** action and then the **wdreg delete** action.

NOTE

Remember that in order to successfully stop the WinDriver service, there cannot be any open handles to the **windrvr6.sys** driver (such as open WinDriver applications). This is also true for the **install** and **uninstall** shortcuts, since both commands include stopping the WinDriver service. **wdreg** will display a relevant warning message if you attempt to stop the service when there are still open handles to the **windrvr6.sys** service.

13.2.3 Dynamically Loading/Unloading **windrvr6.sys** INF Files

When using WinDriver, you develop a user-mode application that controls and accesses your hardware by using the generic driver **windrvr6.sys** (WinDriver's kernel module). Therefore, you might want to dynamically load and unload the driver **windrvr6.sys** – which you can do using **wdreg**.

In addition, in WDM-compatible operating systems, you also need to dynamically load INF files for your Plug and Play devices. **wdreg** enables you to do so automatically on Windows 2000, XP and Server 2003.

This section includes example implementations that are based on the detailed description of **wdreg** contained in the previous section.

Example implementations:

- To start **windrvr6.sys** on Windows 98/Me/2000/XP/Server 2003:

```
\> wdreg -inf [path to windrvr6.inf] install
```

which loads the **windrvr6.inf** file and starts the **windrvr6.sys** service.
- To load an INF file named **device.inf**, located under the **c:\tmp** directory, on Windows 2000/XP/Server 2003:

```
\> wdreg -inf c:\tmp\device.inf install
```
- To start **windrvr6.sys** on Windows NT:

```
\> wdreg install  
which is equivalent to:  
\> wdreg create start
```

To unload the driver/INF file, use the same commands, but simply replace *install* in the samples above with *uninstall*.

13.2.4 Dynamically Loading/Unloading Your Kernel PlugIn Driver

If you have used WinDriver to develop a Kernel PlugIn driver, you must load your Kernel PlugIn after loading the WinDriver generic driver **windrvr6.sys**. When uninstalling your driver, you should unload your Kernel PlugIn driver before unloading **windrvr6.sys**.

NOTE

Kernel PlugIn drivers for Windows 98/Me are not dynamically loaded, they require reboot after the initial loading. Kernel PlugIn driver for all other Windows platforms are dynamically loaded, i.e. they do not require reboot.

To load/unload your Kernel PlugIn driver (<**Your driver name**>.sys) use the **wdreg** command as described above for windrvr6, with the addition of the "name" flag, after which you must add the name of your Kernel PlugIn driver.

NOTE

You should **not** add the *.sys extension to the driver name.

Example implementations:

- To load a Kernel PlugIn driver called **KPDriver.sys**, execute:

```
\> wdreg -name KPDriver install
```
- To load a Kernel PlugIn driver called MPEG_Encoder, with file name MPEGENC.sys, execute:

```
\> wdreg -name MPEG_Encoder -file MPEGENC install
```
- To uninstall a Kernel PlugIn driver called **KPDriver.sys**, execute:

```
\> wdreg -name KPDriver uninstall
```
- To uninstall a Kernel PlugIn driver called MPEG_Encoder, with file name MPEGENC.sys, execute:

```
\> wdreg -name MPEG_Encoder -file MPEGENC uninstall
```

13.3 Linux

- To dynamically load WinDriver on Linux, execute:
`/sbin/modprobe windrvr6`
- To dynamically unload WinDriver, execute:
`/sbin/rmmmod windrvr6`
- In addition, you can use the **wdreg** script under Linux to install (load) **windrvr6.o/.ko**.
Example usage: To load your driver, execute:
`\> wdreg <driver name.extension>`

13.4 Solaris

- After the initial installation you can dynamically load WinDriver on Solaris by executing:
`/usr/sbin/add_drv windrvr6`
- To dynamically unload WinDriver, execute:
`/usr/sbin/rem_drv windrvr6`

Chapter 14

Distributing Your Driver

Read this chapter in the final stages of driver development. It will guide you in preparing your driver for distribution.

NOTE

For **Windows 2000/XP/Server 2003**, all references to **wdreg** in this chapter can be replaced with **wdreg_gui**, which offers the same functionality but displays GUI messages instead of console-mode messages.

For **Windows 98/Me**, all references to **wdreg** should be replaced with **wdreg16**. For more information regarding the **wdreg** utility, see Chapter 13.

14.1 Getting a Valid License for WinDriver

To purchase a WinDriver license, complete the order form, found under **\WinDriver\docs\order.txt**, and fax or email it to Jungo. Complete details are included on the order form. Alternatively, you can order WinDriver on-line. Visit <http://www.jungo.com> for more details.

In order to install the registered version of WinDriver and to activate driver code that you have developed during the evaluation period on the development machine, please follow the installation instructions found in Section 3.2 above.

14.2 Windows 98/Me and Windows 2000/XP/Server 2003

Distributing the driver you created is a multi-step process. First, create a distribution package that includes all the files required for the installation of the driver on the target computer. Second, install the driver on the target machine. This involves installing **windrvr6.sys** and **windrvr6.inf**, installing the specific INF file for your device (for Plug and Play hardware – PCI/PCMCIA), and installing your Kernel PlugIn driver (if you have created one). Finally, you need to install and execute the hardware control application that you developed with WinDriver. These steps can be performed using **wdreg** utility.

NOTE

This section refers to distribution of SYS files. Starting from WinDriver version 6.21 **.vxd** drivers are no longer supported.

14.2.1 Preparing the Distribution Package

Your distribution package should include the following files:

- Your hardware control application/DLL.
- **windrvr6.sys** (get this file from the WinDriver package under the **\WinDriver\redist** directory).
- **windrvr6.inf** (get this file from the WinDriver package under the **\WinDriver\redist** directory).
- **wd_utils.dll** (get this file from the WinDriver package under the **\WinDriver\redist** directory).
- An INF file for your device (required for Plug-and-Play devices, such as PCI and PCMCIA).
You can generate this file with the DriverWizard, as explained in Section 4.2.
- Your Kernel PlugIn driver – **<KP driver name>.sys** – if you have created such a driver.

14.2.2 Installing Your Driver on the Target Computer

NOTE

The user must have administrative privileges on the target computer in order to install your driver.

Follow the instructions below in the order specified to properly install your driver on the target computer:

- **Preliminary Steps:**

- To avoid reboot, before attempting to install the driver make sure that there are no open handles to the **windrvr6.sys** service. This includes verifying that there are no open applications that use this service and that there are no connected Plug-and-Play devices that are registered to work with **windrvr6.sys** – i.e., no INF files that point to this driver are currently installed for any of the Plug-and-Play devices connected to the PC, or the INF file is installed but the device is disabled. This may be relevant, for example, when upgrading a driver developed with an earlier version of WinDriver (version 6.0 and later only, since previous versions used a different module name).

You should therefore either disable or uninstall all Plug-and-Play devices that are registered to work with WinDriver from the Device Manager (**Properties** | **Uninstall, Properties** | **Disable** or **Remove** – on Win98/Me), or otherwise disconnect the device(s) from the PC. If you do not do this, attempts to install the new driver using **wdreg** will produce a message that instructs the user to either uninstall all devices currently registered to work with WinDriver, or reboot the PC in order to successfully execute the installation command.

- **Windows 2000:** Due to Windows 2000's INF selection algorithm, if there are Plug-and-Play INF files, developed with an older version of WinDriver installed on the target computer, we also recommend that you delete any old INF files that Windows/WinDriver may have created for the Plug-and-Play devices that you wish to handle with WinDriver, otherwise an older INF file may be installed, causing the older version of WinDriver to become active (see further explanations in Section 14.4). These files are stored in the **%windir%\inf** directory. You can search the INF directory for a device's vendor ID and device/product ID in this INF directory to locate the file(s) associated with the relevant device(s).

- **Install WinDriver's kernel module:**

1. Copy **windrvr6.sys** and **windrvr6.inf** to the same directory.
2. Use the utility **wdreg/wdreg16** to install WinDriver's kernel module on the target computer.

On Windows 2000/XP/Server 2003 type from the command line:

```
\> wdreg -inf <path to windrvr6.inf> install
```

On Windows 98/Me type from the command line:

```
\> wdreg16 -inf <path to windrvr6.inf> install
```

For example, if **windrvr6.inf** and **windrvr6.sys** are in the **d:\MyDevice** directory on the target computer, the command should be:

```
\> wdreg -inf d:\MyDevice\windrvr6.inf install
```

You can find the executable of **wdreg** in the WinDriver package under the **\WinDriver\util** directory. For a general description of this utility and its usage, please refer to Chapter 13.

NOTE

wdreg is an interactive utility. If it fails, it will display a message instructing the user how to overcome the problem. In some cases the user may be asked to reboot the computer.

CAUTION!

When distributing your driver, take care not to overwrite a newer version of **windrvr6.sys** with an older version of the file in Windows drivers directory (**%windir%\system32\drivers**). You should configure your installation program (if you are using one) or your INF file so that the installer automatically compares the time stamp on these two files and does not overwrite a newer version with an older one.

- **Install the INF file for your device** (registering your Plug-and-Play device with **windrvr6.sys**):

- **Windows 2000/XP/Server 2003:** Use the utility **wdreg** to automatically load the INF file.

To automatically install your INF file on **Windows 2000/XP/Server 2003** and update Windows Device Manager, run **wdreg** with the **install** command:

```
\> wdreg -inf <path to your INF file> install
```

NOTE

On **Windows 2000**, if another INF file was previously installed for the device, which registered the device to work with the Plug-and-Play driver used in earlier versions of WinDriver remove any INF file(s) for the device from the **%windir%\inf** directory before installing the new INF file that you created. This will prevent Windows from automatically detecting and installing an obsolete file. You can search the INF directory for the device's vendor ID and device/product ID to locate the file(s) associated with the device.

- **Windows 98/Me:** Install the INF file manually using Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as outlined in detail in Section 14.4 below.
- **Install your Kernel PlugIn driver:** If you have created a Kernel PlugIn driver, install it by following the instructions in Section 14.2.3.
- **Install the wd_utils DLL:** If your hardware control application/DLL uses the **wd_utils** DLL (as is the case for the sample and generated DriverWizard WinDriver projects), copy **wd_utils.dll** to the target's **%windir%\system32** directory.
- **Install your hardware control application/DLL:** Copy your hardware control application/DLL to the target and run it!

14.2.3 Installing Your Kernel PlugIn on the Target Computer

NOTE

The user must have administrative privileges on the target computer in order to install your Kernel PlugIn driver.

If you have created a Kernel PlugIn driver, follow the additional instructions below:

1. Copy your Kernel PlugIn driver (**<KP driver name>.sys**) to Windows drivers directory on the target computer (**%windir%\system32\drivers**).
2. Use the utility **wdreg** to add your Kernel PlugIn driver to the list of device drivers Windows loads on boot. Use the following installation command:
To install a **SYS** Kernel PlugIn Driver:

```
\> wdreg -name <Your driver name, without the *.sys extension> install
```

You can find the executable of **wdreg** in the WinDriver package under the **\WinDriver\util** directory. For a general description of this utility and its

usage, please refer to Chapter 13 (see specifically Section 13.2.4 for Kernel PlugIn installation).

14.3 Windows NT 4.0

Distributing the driver you created is a multi-step process. First, create a distribution package that includes all the files required for installation of the driver on the target computer. Second, install WinDriver's generic driver (**windrvr6.sys**). If you have created a Kernel PlugIn driver, install it on the target computer as well. Finally, you need to install and execute the hardware control application you developed with WinDriver on the target computer. The following subsections describe this process in detail.

14.3.1 Preparing the Distribution Package

Your distribution package should include the following files:

- Your hardware control application/DLL.
- **windrvr6.sys** for Windows NT (get this file from the WinDriver package under the **\WinDriver\redist** directory).
- **wd_utils.dll** (get this file from the WinDriver package under the **\WinDriver\redist** directory).
- Your Kernel PlugIn driver – **<driver name>.sys** – if you have created such a driver.

14.3.2 Installing Your Driver on the Target Computer

NOTE

The user must have administrative privileges on the target computer in order to install your driver.

Follow the instructions below in the order specified to properly install your driver on the target computer:

1. Make sure that there are no open handles to **windrvr6.sys**.
2. Copy the file **windrvr6.sys** to Windows drivers directory on the target computer (**%windir%\system32\drivers**).

3. Use the utility **wdreg** to add **windrvr6.sys** to the list of device drivers Windows loads on boot.
Use the following installation command:

```
\> wdreg install
```

You can find the executable of **wdreg** in the WinDriver package under the **\WinDriver\util** directory. For a general description of this utility and its usage, please refer to Chapter 13.
4. If you have created a Kernel PlugIn driver, install it by following the instructions in Section 14.3.3.
5. If your hardware control application/DLL uses the **wd_utils** DLL (as is the case for the sample and generated DriverWizard WinDriver projects), copy **wd_utils.dll** to the target's **%windir%\system32** directory.
6. Copy your hardware control application/DLL to the target and run it!

14.3.3 Installing Your Kernel PlugIn on the Target Computer

NOTE

The user must have administrative privileges on the target computer in order to install your Kernel PlugIn driver.

If you have created a Kernel PlugIn driver, follow the additional instructions below:

1. Copy your Kernel PlugIn driver (**<driver name>.sys**) to Windows drivers installation directory on the target computer (**%windir%\system32\drivers**)

CAUTION!

When distributing your driver, take care not to overwrite a newer version of **windrvr6.sys** with an older version of the file in the Windows drivers directory (**%windir%\system32\drivers** for **windrvr6.sys**). You should configure your installation program (if you are using one) so that the installer automatically compares the time stamp on these two files and does not overwrite a newer version with an older one.

2. Use the utility **wdreg** to add your Kernel PlugIn driver to the list of device drivers Windows loads on boot.

Use the following installation command:

```
\> wdreg -name [Your driver name] install
```

You can find the executable of **wdreg** in the WinDriver package under the **\WinDriver\util** directory. For a general description of this utility and its usage, please refer to Chapter 13 (see specifically Section 13.2.4).

14.4 Creating an INF File

Device information (INF) files are text files that provide information used by the Plug and Play mechanism in Windows 98/Me/2000/XP/Server 2003 to install software that supports a given hardware device. INF files are required for hardware that identifies itself, such as USB and PCI. An INF file includes all necessary information about a device and the files to be installed. When hardware manufacturers introduce new products, they must create INF files to explicitly define the resources and files required for each class of device.

In some cases, the INF file for your specific device is supplied by the operating system. In other cases, you will need to create an INF file for your device. WinDriver's DriverWizard can generate a specific INF file for your device. The INF file is used to notify the operating system that WinDriver now handles the selected device.

You can use the DriverWizard to generate the INF file on the development machine – as explained in Section 4.2 of the manual – and then install the INF file on any machine to which you distribute the driver, as explained in the following sections.

14.4.1 Why Should I Create an INF File?

- To stop the Windows **Found New Hardware Wizard** from popping up after each boot.
- To ensure that the operating system can initialize the PCI configuration registers on Windows 98/Me/2000/XP/Server 2003.
- To load the new driver created for the device.
An INF file must be created whenever developing a new driver for Plug and Play hardware that will be installed on a Plug and Play system.
- To replace the existing driver with a new one.

14.4.2 How Do I Install an INF File When No Driver Exists?

NOTE

You must have administrative privileges in order to install an INF file on Windows 98, Me, 2000, XP and Server 2003.

- **Windows 2000/XP/Server 2003:**
On Windows 2000/XP/Server 2003 you can use the **wdreg** utility with the **install** command to automatically install the INF file:


```
\> wdg -inf <path to the INF file> install
```

See Section 13.2.2 of the manual for more information.

On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with the DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (see Section 4.2).

It is also possible to install the INF file manually on Windows 2000/XP/Server 2003, using either of the following methods:

- Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.
- Windows **Add/Remove Hardware Wizard**: Right-click the mouse on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard...**
- Windows **Upgrade Device Driver Wizard**: Select the device from the **Device Manager** devices list, select **Properties**, choose the **Driver** tab and click the **Update Driver...** button. On Windows XP and Windows Server 2003 you can choose to upgrade the driver directly from the Properties list.

In all the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation.

We recommend using the **wdg** utility to install the INF file automatically, instead of installing it manually.

- **Windows 98/Me:**

On **Windows 98/Me** you need to install the INF file for your PCI/PCMCIA device manually, either via Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained below:

- Windows **Add New Hardware Wizard**:

NOTE

This method can be used if no other driver is currently installed for the device or if the user first uninstalls (removes) the current driver for the device. Otherwise, Windows **New Hardware Found Wizard**, which activates the **Add New Hardware Wizard**, will not appear for this device.

1. To activate the Windows **Add New Hardware Wizard**, attach the hardware device to the computer or, if the device is already connected, scan for hardware changes (**Refresh**).
2. When Windows **Add New Hardware Wizard** appears, follow its

installation instructions. When asked, point to the location of the INF file in your distribution package.

– Windows **Upgrade Device Driver Wizard**:

1. Open Windows Device Manager: From the **System Properties** window (right-click on **My Computer** and select **Properties**) select the **Device Manager** tab.
2. Select your device from the **Device Manager** devices list, choose the **Driver** tab and click the **Update Driver** button.
To locate your device in the Device Manager, select **View devices by connection**. For PCI devices, navigate to **Standard PC | PCI bus | <your device>**.
3. Follow the instructions of the **Upgrade Device Driver Wizard** that opens. When asked, point to the location of the INF file in your distribution package.

14.4.3 How Do I Replace an Existing Driver Using the INF File?

NOTE

You must have administrative privileges in order to replace a driver on Windows 98, Me, 2000, XP and Server 2003.

1. On **Windows 2000**, if you wish to upgrade the driver for PCI/PCMCIA devices that have been registered to work with earlier versions of WinDriver, we recommend that you first delete from Windows INF directory (`%windir%\inf`) any previous INF files for the device, to prevent Windows from installing an old INF file in place of the new file that you created. Look for files containing your device's vendor and device IDs and delete them.
2. Install your INF file:
 - On **Windows 2000/XP/Server 2003** you can automatically install the INF file:
You can use the **wdreg** utility with the **install** command to automatically install the INF file on Windows 2000/XP/Server 2003:
`> wdreg -inf <path to INF file> install`
See Section 13.2.2 of the manual for more information.
On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with the DriverWizard, by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation window (see Section 4.2).

It is also possible to install the INF file manually on Windows 2000/XP/Server 2003, using either of the following methods:

- Windows **Found New Hardware Wizard**: This wizard is activated when the device is plugged in or, if the device was already connected, when scanning for hardware changes from the Device Manager.
- Windows **Add/Remove Hardware Wizard**: Right-click on **My Computer**, select **Properties**, choose the **Hardware** tab and click on **Hardware Wizard....**
- Windows **Upgrade Device Driver Wizard**: Select the device from the **Device Manager** devices list, select **Properties**, choose the **Driver** tab and click the **Update Driver...** button. On Windows XP and Windows Server 2003 you can choose to upgrade the driver directly from the Properties list.

In the manual installation methods above you will need to point Windows to the location of the relevant INF file during the installation. If the installation wizard offers to install an INF file other than the one you have generated, select **Install one of the other drivers** and choose your specific INF file from the list.

We recommend using the **wdreg** utility to install the INF file automatically, instead of installing it manually.

- On **Windows 98/Me** you need to install the INF file manually via Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained below:

- Windows **Add New Hardware Wizard**:

NOTE

This method can be used if no other driver is currently installed for the device or if the user first uninstalls (removes) the current driver for the device. Otherwise, the Windows **Found New Hardware Wizard**, which activates the **Add New Hardware Wizard**, will not appear for this device.

- (a) To activate the Windows **Add New Hardware Wizard**, attach the hardware device to the computer or, if the device is already connected, scan for hardware changes (Refresh).
 - (b) When Windows **Add New Hardware Wizard** appears, follow its installation instructions. When asked, specify the location of the INF file in your distribution package.
- Windows **Upgrade Device Driver Wizard**:

- (a) Open Windows Device Manager: From the **System Properties** window (right click on **My Computer** and select **Properties**) select the **Device Manager** tab.
- (b) Select your device from the **Device Manager** devices list, open it, choose the **Driver** tab and click the **Update Driver** button. To locate your device in the Device Manager, select **View devices by connection**. For PCI devices, navigate to **Standard PC | PCI bus | <your device>**.
- (c) Follow the instructions of the **Upgrade Device Driver Wizard** that opens. Locate the INF in your distribution package when asked.

14.5 Windows CE

To distribute the driver you developed with WinDriver to a target Windows CE platform, follow these steps:

1. Install WinDriver's kernel DLL (**windrvr6.dll**) on the target computer:
 - For WinDriver applications developed for target CE computers: Copy **windrvr6.dll** from the **\WinDriver\redist\TARGET_CPU** directory to the **Windows** directory on your target Windows CE computer.
 - When building new CE platforms: Copy **windrvr6.dll** from the **\WinDriver\redist\TARGET_CPU** directory to the **%_FLATRELEASEDIR%** directory and then append the contents of the supplied file **PROJECT_WD.BIB** to the file **PROJECT.BIB**. This will make the WinDriver kernel file a permanent part of the Windows CE kernel **NK.BIN**. Then use **MAKEIMG.EXE** to build the new Windows CE kernel **NK.BIN**. This process is similar to the process of installing WinDriver CE with Platform Builder, as described in Section 3.2.2.
2. Add WinDriver to the list of device drivers Windows CE loads on boot:
 - For WinDriver applications developed for target CE computers: Modify the registry according to the entries documented in the file **PROJECT_WD.REG**. This can be done using the Windows CE Pocket Registry Editor on the hand-held CE computer or by using the Remote CE Registry Editor Tool supplied with the Windows CE Platform SDK. You will need to have Windows CE Services installed on your Windows Host System to use the Remote CE Registry Editor Tool.

- When building new CE platforms:
The required registry entries are made by appending the contents of the file **PROJECT_WD.REG** to the Windows CE ETK configuration file **PROJECT.REG** before building the Windows CE image using **MAKEIMG.EXE**.

NOTE

On non-x86 platforms, for PCI only: Make sure you copy the lines specified for PCI from **PROJECT_WD.REG** to **PROJECT.REG**, after removing the comment marks and inserting the card specific information.

3. Install your hardware control application/DLL on the target.

If your hardware control application/DLL uses **wd_utils.dll** (as is the case for the sample and generated DriverWizard WinDriver projects), also copy **wd_utils.dll** from the **WinDriver\redist** directory on the development PC to the target's **Windows** directory.

14.6 Linux

The Linux kernel is continuously under development and kernel data structures are subject to frequent changes. To support such a dynamic development environment and still have kernel stability, the Linux kernel developers decided that kernel modules must be compiled with header files identical to those with which the kernel itself was compiled. They enforce this by including a version number in the kernel header files that is checked against the version number encoded into the kernel. This forces Linux driver developers to facilitate recompilation of their driver based on the target system's kernel version.

14.6.1 WinDriver Kernel Module

Since **windrvr6.o/.ko** is a kernel module, it must be recompiled for every kernel version on which it is loaded. To facilitate this, we supply the following components to insulate the WinDriver kernel module from the Linux kernel:

- **windrvr_gcc_v2.a**, **windrvr_gcc_v3.a** and **windrvr_gcc_v3_regparm.a**: compiled object code for the WinDriver kernel module. **windrvr_gcc_v2.a** is used for kernels compiled with gcc v2.x.x, and **windrvr_gcc_v3.a** is used for kernels compiled with gcc v3.x.x. **windrvr_gcc_v3_regparm.a** is used for kernels compiled with gcc v3.x.x with the **regparm** flag.

- **linux_wrappers.c/h**: wrapper library source code files that bind the WinDriver kernel module to the Linux kernel.
- **linux_common.h**, **windrvr.h**, **wd_ver.h** and **wdusb_interface.h**: header files required for building the WinDriver kernel module on the target. (Note that **wdusb_interface.h** is required also for PCI/PCMCIA/ISA drivers, not just for USB).
- **wdusb_linux.c**: used by WinDriver to utilize the USB stack. Even though this is a USB file, it is also required for PCI/PCMCIA/ISA drivers.
- **configure**: a configuration script that creates a **makefile** that compiles and inserts the module **windrvr6.o/.ko** into the kernel.
- **makefile.in**, **wdreg** and **setup_inst_dir**: the **configure** script uses **makefile.in**, which creates a makefile. This makefile calls the **wdreg** utility shell script and **setup_inst_dir**, which we supply under the **WinDriver/util** directory. All three must be copied to the target.

You need to distribute these components along with your driver source code or object code.

14.6.2 User-Mode Hardware Control Application/Shared Objects

Copy the hardware control application/shared objects that you created with WinDriver to the target.

If your hardware control application/shared objects use the **libwd_utils.so** shared object (as is the case for the sample and generated DriverWizard WinDriver projects), copy **libwd_utils.so** from the **WinDriver/lib** directory on the development PC to the target's library directory (**/usr/lib/** – for 32-bit PowerPC or 32-bit x86 targets; **/user/lib64** – for 64-bit x86 targets).

Since the user-mode hardware control application/shared objects do not have to be matched against the kernel version number, you are free to distribute it as binary code (if you wish to protect your source code from unauthorized copying) or as source code.

CAUTION!

If you select to distribute your source code, make sure you do not distribute your WinDriver license string, which is used in the code.

14.6.3 Kernel PlugIn Modules

Since the Kernel PlugIn module (if you have created such a module) is a kernel module, it also needs to be matched against the active kernel's version number. This means recompilation for the target system. It is advisable to supply the Kernel PlugIn module source code to your customers so that they can recompile it. You can use the **configure** script that the DriverWizard created for you in the code generation of the Kernel PlugIn to build and insert any Kernel PlugIn modules that you distribute.

NOTE

You may have to perform adjustments to the **configure** script, particularly concerning the locations of files (their paths).

To enable re-compilation of your Kernel PlugIn driver on different Linux targets, you are also free to distribute the following files: **kp_linux_gcc_v2.o**, **kp_linux_gcc_v3.o**, **kp_linux_gcc_v3_regparm.o**, **kp_wd_utils_gcc_v2.a**, **kp_wd_utils_gcc_v3.a** and **kp_wd_utils_gcc_v3_regparm.a**.

The **xxx_gcc_v2.o/a** files are used for kernels compiled with gcc v2.x.x, the **xxx_gcc_v3.o/a** files are used for kernels compiled with gcc v3.x.x, and the **xxx_gcc_v3_regparm.o/a** files are used for kernels compiled with gcc v3.x.x with the **regparm** flag.

14.6.4 Installation Script

We suggest that you supply an installation shell script that copies your driver executables/DLL to the correct locations (perhaps **/usr/local/bin**) and then invokes **make** or **gmake** to build and install the WinDriver kernel module and any Kernel PlugIn modules.

14.7 Solaris

For Solaris, you need to supply the following to allow the client to enable target installation of your driver:

- WinDriver's kernel module: The files **windrvr6** and **windrvr6.conf** implement the WinDriver kernel module.
- User-mode hardware control application/shared object: Your user-mode hardware control application/shared object binaries.
- If your hardware control application/shared object uses WinDriver's **libwd_utils.so** shared object (as is the case for the sample and generated

DriverWizard WinDriver projects), copy **libwd_utils.so** from the **WinDriver/lib** directory on the development PC to the target's library directory (**/lib/32** – for 32-bit SPARC or 32-bit x86 targets; **/lib/64** – for 64-bit SPARC targets).

- Kernel PlugIn module: If you used a Kernel PlugIn module, you should supply the relevant files, e.g., **mykp** and **mykp.conf**.
- An installation script: We suggest that you supply an installation shell script that copies your driver executables to the correct locations (perhaps **/usr/local/bin**) and then installs the WinDriver kernel. You may adapt the utility script **install_windrvr6** (found under the **WinDriver** directory on the development machine) to your purposes.

14.8 VxWorks

For VxWorks, you need to supply the following to allow the client to enable target installation of your driver:

- WinDriver's kernel module: The file **windrvr6.o.ko** implements the WinDriver kernel module.
- Your hardware control application/DLL: the source code or the binaries of your hardware control application/DLL (**your_drv.out**, for example).

Your client will need to incorporate all these files into the VxWorks embedded image. There are two steps involved here:

1. **windrvr6.o.ko** and **your_drv.out** have to be built into the VxWorks image.

In the Tornado II Project's build specification for the VxWorks image, specify **windrvr6.o.ko** and **your_drv.out** as **EXTRA_MODULES** under the **MACROS** tab, and copy these files under the appropriate target directory tree. Rebuild the project. These files should now be included in the image.

2. The **drvInit()** routine should be called during startup to initialize **windrvr6.o.ko**. Your driver's startup routine may also need to be called.

Add code to **usrAppInit.c** (found under the Tornado II project directory) so that it will call **drvInit**—WinDriver's initialization routine—and your driver application's startup routine. Of course, you will need to rebuild the VxWorks image after modifying **usrAppInit.c**.

Appendix A

API Reference

A.1 PCI/ISA/PCMCIA/CardBus – WDC Library Overview

The "*WinDriver Card*" – **WDC** – API provides convenient user-mode wrappers to the basic WinDriver PCI/ISA/PCMCIA/CardBus `WD_XXX` API [\[A.4\]](#).

The WDC wrappers are designed to simplify the usage of WinDriver for communicating with PCI/ISA/PCMCIA/CardBus devices. While you can still use the basic `WD_XXX` PCI/PCMCIA/ISA WinDriver API from your code, we recommend that you refrain from doing so and use the high-level WDC API instead.

NOTE: Most of the WDC API can be used both from the user mode and from the kernel mode (from a Kernel PlugIn driver [\[11\]](#).)

The generated DriverWizard PCI/PCMCIA/ISA diagnostics driver code, as well as the PLX sample code, and the **pci_diag**, Kernel PlugIn **pci_diag**, **pcmcia_diag** and **pci_dump** samples, for example, utilize the WDC API.

The WDC API is part of **wd_utils** DLL/shared object: **WinDriver/redist/wd_utils.dll** (Windows 98/Me/NT/2000/XP/Server 2003 and Windows CE) / **WinDriver/lib/libwd_utils.so** (Linux and Solaris). The source code for the WDC API is found in the **WinDriver/src** directory.

The WDC interface is provided in the **wdc_lib.h** and **wdc_defs.h** header files (both found under the **WinDriver/includes** directory.)

wdc_lib.h declares the "high-level" WDC API (type definitions, function declarations, etc.).

wdc_defs.h declares the "low-level" WDC API. This file includes definitions and type information that is encapsulated by the high-level **wdc_lib.h** file.

The WinDriver PCI/PCMCIA/ISA samples and generated DriverWizard code that utilize the WDC API, for example, are comprised of a "library" for the specific device, and a diagnostics application that uses it. The high-level diagnostics code only utilizes the **wdc_lib.h** API, while the library code also uses the lower-level API from the **wdc_defs.h** file, thus maintaining the desired level of encapsulation.

The following sections describe the WDC high-level [A.2] and low-level [A.3] API.

NOTES

- CardBus devices are handled via WinDriver's PCI API, therefore any references to **PCI** in this chapter also include **CardBus**.
- The PCMCIA API – both in the WDC library and in the low-level **WD_XXX** WinDriver API – is supported only on Windows 2000/XP/Server 2003.

A.2 PCI/PCMCIA/ISA – WDC High Level API

This section described the WDC API defined in the **WinDriver/include/wdc_lib.h** header file

A.2.1 Structures, Types and General Definitions

A.2.1.1 WDC_DEVICE_HANDLE

Handle to a WDC device information structure [A.3.3] type

```
typedef void * WDC_DEVICE_HANDLE;
```

A.2.1.2 WDC_DRV_OPEN_OPTIONS Definitions

```
typedef DWORD WDC_DRV_OPEN_OPTIONS;
```

Preprocessor definitions of flags that describe tasks to be performed when opening a handle to the WDC library (see `WDC_DriverOpen()` [A.2.2]):

Name	Description
WDC_DRV_OPEN_CHECK_VER	Compare the version of the WinDriver source files used by the code with the version of the loaded WinDriver kernel
WDC_DRV_OPEN_REG_LIC	Register a WinDriver license registration string

The following preprocessor definitions provide convenient WDC driver open options, which can be passed to `WDC_DriverOpen()` [A.2.2]:

Name	Description
WDC_DRV_OPEN_BASIC	Instructs <code>WDC_Driveropen()</code> [A.2.2] to perform only the basic WDC open tasks, mainly open a handle to WinDriver's kernel module. NOTE: The value of this option is zero (0) (<=> no driver open flags), therefore this option cannot be combined with any of the other WDC driver open options

Name	Description
WDC_DRV_OPEN_KP	Convenience option when calling WDC_DriverOpen() [A.2.2] from the Kernel PlugIn. This option is equivalent to setting the WDC_DRV_OPEN_BASIC flag, which is the recommended option to set when opening a handle to the WDC library from the Kernel PlugIn.
WDC_DRV_OPEN_ALL	A convenience mask of all the basic WDC driver open flags – WDC_DRV_OPEN_CHECK_VER and WDC_DRV_OPEN_REG_REG_LIC. (The basic functionality of opening a handle to WinDriver's kernel module is always performed by WDC_DriverOpen() [A.2.2], so there is no need to also set the WDC_DRV_OPEN_BASIC flag)
WDC_DRV_OPEN_DEFAULT	Use the default WDC open options: <ul style="list-style-type: none"> • For user-mode applications: equivalent to setting WDC_DRV_OPEN_ALL ; • For a Kernel PlugIn: equivalent to setting WDC_DRV_OPEN_KP

A.2.1.3 WDC_DIRECTION Enumeration

Enumeration of a device's address/register access directions:

Enum Value	Description
WDC_READ	Read from the address
WDC_WRITE	Write to the address

A.2.1.4 WDC_ADDR_MODE Enumeration

Enumeration of memory or I/O addresses/registers read/write modes.

The enumeration values are used to determine whether a memory or I/O address/register is read/written in multiples of 8, 16, 32 or 64 bits (i.e. 1, 2, 4 or 8 bytes).

Enum Value	Description
WDC_MODE_8	8 bits (1 byte) mode

Enum Value	Description
WDC_MODE_16	16 bits (2 bytes) mode
WDC_MODE_32	32 bits (4 bytes) mode
WDC_MODE_64	64 bits (8 bytes) mode

A.2.1.5 WDC_ADDR_RW_OPTIONS Enumeration

Enumeration of flags that are used to determine how a memory or I/O address will be read/written:

Enum Value	Description
WDC_RW_BLOCK	Read/write a block of addresses. This flag is set automatically by the <code>WDC_ReadAddrBlock()</code> [A.2.20] and <code>WDC_WriteAddrBlock()</code> [A.2.21] functions.
WDC_RW_MEM_INDIRECT	Use WinDriver to read/write a memory address in the kernel (using <code>WD_Transfer()</code> [A.4.14]). This option is relevant only for memory addresses, which by default are read/written by the WDC API directly, using the user-mode mapping of the address when accessing the memory from the user mode, or the kernel-mode mapping when accessing memory from a Kernel PlugIn driver. I/O addresses are always read/written in the kernel using <code>WD_Transfer()</code> .
WDC_BLOCK_AUTOINC	Automatically increment the read/write address after each block of memory or I/O addresses that is read/written. (If this flag is not set, each block is read/written from the same address.) This option is only relevant for block transfers (i.e. when the <code>WDC_RW_BLOCK</code> flag is also set.)

The following enumeration value provides a convenience option for using the default WDC read/write address options:

Enum Value	Description
WDC_RW_OPT_DEFAULT	<p>Instructs WDC to use the default read/write options: Read/write memory addresses directly from the user mode (WDC_RW_MEM_INDIRECT is <i>not</i> set) using single (non-block) transfers (i.e. WDC_RW_BLOCK is <i>not</i> set). NOTE: The value of this option is zero (0) (<=> no read/write flags), therefore this option cannot be combined with any of the other WDC_ADDR_RW_OPTIONS options.</p> <p>This option is used by the WDC_ReadAddr8/16/32/64() [A.2.18] and WDC_WriteAddr8/16/32/64() [A.2.19] functions.</p>

A.2.1.6 WDC_ADDR_SIZE Definitions

```
typedef DWORD WDC_ADDR_SIZE;
```

Preprocessor definitions that depict memory or I/O address/register sizes:

Name	Description
WDC_SIZE_8	8 bits (1 byte)
WDC_SIZE_16	16 bits (2 bytes)
WDC_SIZE_32	32 bits (4 bytes)
WDC_SIZE_64	64 bits (8 bytes)

A.2.1.7 WDC_SLEEP_OPTIONS Definitions

```
typedef DWORD WDC_SLEEP_OPTIONS;
```

Preprocessor definitions depict the sleep options that can be passed to WDC_Sleep() [A.2.53]:

Name	Description
WDC_SLEEP_BUSY	Perform busy sleep (consumes the CPU)
WDC_SLEEP_NON_BUSY	Perform non-busy sleep (does not consume the CPU)

A.2.1.8 WDC_DBG_OPTIONS Definitions

```
typedef DWORD WDC_DBG_OPTIONS;
```

Preprocessor definitions that depict the possible debug options for the WDC library, which are passed to `WDC_SetDebugOptions()` [A.2.47].

The following flags determine the output file for the WDC library's debug messages:

Name	Description
WDC_DBG_OUT_DBM	Send debug messages from the WDC library to the Debug Monitor [6.2]
WDC_DBG_OUT_FILE	Send debug messages from the WDC library to a debug file. By default, the debug file will be stderr , unless a different file is set in the <code>sDbgFile</code> parameter of the <code>WDC_SetDebugOptions()</code> function [A.2.47]. This option is only supported from the user mode (as opposed to the Kernel PlugIn.)

The following flags determine the debug level – i.e. what type of WDC debug messages to display, if at all:

Name	Description
WDC_DBG_LEVEL_ERR	Display only WDC error debug messages
WDC_DBG_LEVEL_TRACE	Display both error and trace WDC debug messages
WDC_DBG_NONE	Do not display WDC debug messages

The following preprocessor definitions provide convenient debug flags combinations, which can be passed to `WDC_SetDebugOptions()` [A.2.47]:

- User-mode and Kernel PlugIn convenience debug options:

Name	Description
WDC_DBG_DEFAULT	WDC_DBG_OUT_DBM WDC_DBG_LEVEL_TRACE : Use the default debug options – send WDC error and trace messages to the Debug Monitor [6.2]
WDC_DBG_DBM_ERR	WDC_DBG_OUT_DBM WDC_DBG_LEVEL_ERR : Send WDC error debug messages to the Debug Monitor [6.2]

Name	Description
WDC_DBG_DBM_TRACE	WDC_DBG_OUT_DBM WDC_DBG_LEVEL_TRACE : Send WDC error and trace debug messages to the Debug Monitor [6.2]
WDC_DBG_FULL	Full WDC debugging: <ul style="list-style-type: none"> • From the user mode: WDC_DBG_OUT_DBM WDC_DBG_OUT_FILE WDC_DBG_LEVEL_TRACE : Send WDC error and trace debug messages both to the Debug Monitor [6.2] and to a debug output file (default file: stderr) • From the Kernel PlugIn: WDC_DBG_OUT_DBM WDC_DBG_LEVEL_TRACE : Send WDC error and trace messages to the Debug Monitor [6.2]

- User-mode only convenience debug options:

Name	Description
WDC_DBG_FILE_ERR	WDC_DBG_OUT_FILE WDC_DBG_LEVEL_ERR : Send WDC error debug messages to a debug file (default file: stderr)
WDC_DBG_FILE_TRACE	WDC_DBG_OUT_FILE WDC_DBG_LEVEL_TRACE : Send WDC error and trace debug messages to a debug file (default file: stderr)
WDC_DBG_DBM_FILE_ERR	WDC_DBG_OUT_DBM WDC_DBG_OUT_FILE WDC_DBG_LEVEL_ERR : Send WDC error debug messages both to the Debug Monitor [6.2] and to a debug file (default file: stderr)
WDC_DBG_DBM_FILE_TRACE	WDC_DBG_OUT_DBM WDC_DBG_OUT_FILE WDC_DBG_LEVEL_TRACE : Send WDC error and trace debug messages both to the Debug Monitor [6.2] and to a debug file (default file: stderr)

A.2.1.9 WDC_SLOT_U Union

WDC PCI/PCMCIA device location information union type:

Name	Type	Description
➤ pciSlot	WD_PCI_SLOT	PCI device location information structure
❑ dwBus	DWORD	Bus number
❑ dwSlot	DWORD	Slot number
❑ dwFunction	DWORD	Function number
➤ pcmciaSlot	WD_PCMCIA_SLOT	PCMCIA device location information structure
❑ uBus	BYTE	Bus number
❑ uSocket	BYTE	Socket number
❑ uFunction	BYTE	Function number

A.2.1.10 WDC_PCI_SCAN_RESULT

Structure type for holding the results of a PCI bus scan (see `WDC_PciScanDevices()` [A.2.4]):

Name	Type	Description
➤ dwNumDevices	DWORD	Number of devices found on the PCI bus that match the search criteria (vendor & device IDs)
➤ deviceId	WD_PCI_ID[WD_PCI_CARDS]	Array of matching vendor and device IDs found on the PCI bus
❑ dwVendorId	DWORD	Vendor ID
❑ dwDeviceId	DWORD	Device ID
➤ deviceSlot	WD_PCI_SLOT[WD_PCI_CARDS]	Array of structures that hold information regarding the location of the matching devices that were found
❑ dwBus	DWORD	Bus number
❑ dwSlot	DWORD	Slot number
❑ dwFunction	DWORD	Function number

A.2.1.11 WDC_PCMCIA_SCAN_RESULT

Structure type for holding the results of a PCMCIA bus scan (see

WDC_PcmciaScanDevices() [A.2.5]):

Name	Type	Description
➤ dwNumDevices	DWORD	Number of devices found on the PCMCIA bus that match the search criteria (manufacturer & device IDs)
➤ deviceId	WD_PCMCIA_ID[WD_PCMCIA_CARDS]	Array of matching vendor and device IDs found on the PCMCIA bus
❑ wManufacturerId	WORD	Manufacturer ID
❑ wCardId	WORD	Device ID
➤ deviceSlot	WD_PCMCIA_SLOT[WD_PCMCIA_CARDS]	Array of structures that hold information regarding the location of the matching devices that were found
❑ uBus	BYTE	Bus number
❑ uSocket	BYTE	Socket number
❑ uFunction	BYTE	Function number

A.2.2 WDC_DriverOpen()

PURPOSE

- Opens and stores a handle to WinDriver's kernel module and initializes the WDC library according to the open options passed to it.
- This function should be called once before calling any other WDC API.

PROTOTYPE

```
DWORD WINAPI WDC_DriverOpen(WDC_DRV_OPEN_OPTIONS openOptions ,
    const CHAR *sLicense);
```

PARAMETERS

Name	Type	Input/Output
> openOptions	WDC_DRV_OPEN_OPTIONS	Input
> sLicense	const CHAR*	Input

DESCRIPTION

Name	Description
openOptions	A mask of any of the supported open flags [A.2.1.2], which determines the initialization actions that will be performed by the function.
sLicense	WinDriver license registration string. This argument is ignored if the WDC_DRV_OPEN_REG_LIC flag is not [A.2.1.2] set in the openOptions argument. If this parameter is a NULL pointer or an empty string, the function will attempt to register the demo WinDriver evaluation license. Therefore, when evaluating WinDriver pass NULL as this parameter. After registering your WinDriver toolkit, modify the code to pass your WinDriver license registration string.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.3 WDC_DriverClose()

- Closes the WDC WinDriver handle (acquired and stored by a previous call to `WDC_DriverOpen()` [A.2.2]) and un-initializes the WDC library.

Every `WDC_DriverOpen()` call should have a matching `WDC_DriverClose()` call, which should be issued when you no longer need to use the WDC library.

PROTOTYPE

```
DWORD WINAPI WDC_DriverClose ( void );
```

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.4 WDC_PciScanDevices()

PURPOSE

• Scans the PCI bus for all devices with the specified vendor and device ID combination and returns information regarding the matching devices that were found and their location.

PROTOTYPE

```
DWORD WINAPI WDC_PciScanDevices(DWORD dwVendorId, DWORD dwDeviceId,  
                                WDC_PCI_SCAN_RESULT *pPciScanResult);
```

PARAMETERS

Name	Type	Input/Output
> dwVendorId	DWORD	Input
> dwDeviceId	DWORD	Input
> pPciScanResult	WDC_PCI_SCAN_RESULT*	Output

DESCRIPTION

Name	Description
dwVendorId	Vendor ID to search for (hexadecimal). Zero (0) – all vendor IDs.
dwDeviceId	Device ID to search for (hexadecimal). Zero (0) – all device IDs.
pPciScanResult	A pointer to a structure that will be updated by the function with the results of the PCI bus scan [A.2.1.10]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- If you set both the vendor and device IDs to zero (0) will return, the function will return information regarding all connected PCI devices.

A.2.5 WDC_PcmciaScanDevices()

PURPOSE

- Scans the PCMCIA bus for all devices with the specified manufacturer and device ID combination and returns information regarding the matching devices that were found and their location.

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaScanDevices(WORD wManufacturerId, WORD wDeviceId,
    WDC_PCMCIA_SCAN_RESULT *pPcmciaScanResult);
```

PARAMETERS

Name	Type	Input/Output
> wManufacturerId	WORD	Input
> wDeviceId	WORD	Input
> pPcmciaScanResult	WDC_PCMCIA_SCAN_RESULT*	Output

DESCRIPTION

Name	Description
wManufacturerId	Manufacturer ID to search for (hexadecimal). Zero (0) – all manufacturer IDs.
wDeviceId	Device ID to search for (hexadecimal). Zero (0) – all device IDs.
pPcmciaScanResult	A pointer to a structure that will be updated by the function with the results of the PCMCIA bus scan [A.2.1.11]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- If you set both the vendor and device IDs to zero (0) will return, the function will return information regarding all connected PCI devices.

A.2.6 WDC_PciGetDeviceInfo()

PURPOSE

- Retrieves a PCI device's resources information (memory and I/O ranges and interrupt information).

PROTOTYPE

```
DWORD WINAPI WDC_PciGetDeviceInfo(WD_PCI_CARD_INFO *pDeviceInfo);
```

PARAMETERS

Name	Type	Input/Output
> pDeviceInfo	WD_PCI_CARD_INFO*	Input/Output
❑ pciSlot	WD_PCI_SLOT	Input
◆ dwBus	DWORD	Input
◆ dwSlot	DWORD	Input
◆ dwFunction	DWORD	Input
❑ Card	WD_CARD	Output
◆ dwItems	DWORD	Output
◆ Item	WD_ITEMS[WD_CARD_ITEMS]	Output
◇ item	DWORD	Output
◇ fNotSharable	DWORD	Output
◇ I	union	Output
◆ Mem	struct	Output
→ dwPhysicalAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwTransAddr	DWORD	N/A
→ dwUserDirectAddr	DWORD	N/A
→ dwCpuPhysicalAddr	DWORD	N/A
→ dwBar	DWORD	Output
◆ IO	struct	Output
→ dwAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwBar	DWORD	Output
◆ Int	struct	Output
→ dwInterrupt	DWORD	Output
→ dwOptions	DWORD	N/A

Name	Type	Input/Output
→ hInterrupt	DWORD	N/A
♦ Bus	struct	Output
→ dwBusType	WD_BUS_TYPE	Output
→ dwBusNum	DWORD	Output
→ dwSlotFunc	DWORD	Output

DESCRIPTION

Name	Description
pDeviceInfo	Pointer to a PCI device information structure
pciSlot	Pointer to device location information structure, which can be acquired by calling WDC_PciScanDevices() [A.2.4]
dwBus	Bus number
dwSlot	Slot number
dwFunction	Function number
Card	PCI device resources information structure
dwItems	Number of items (resources) on the device
Item	Array of device resources (items) information structures
Item	Item type: ITEM_NONE / ITEM_INTERRUPT / ITEM_MEMORY / ITEM_IO / ITEM_BUS
fNotSharable	If true, only one application at a time can access the memory or I/O range, or monitor the device's interrupts
I	Union of resources data based on the item type (Item)
I.Mem	Memory item (ITEM_MEMORY) information
I.Mem.dwPhysicalAddr	First address of the physical memory range
I.Mem.dwBytes	Length (in bytes) of the memory range
I.Mem.dwBar	Base Address Register number
I.IO	I/O item (ITEM_IO) information
I.IO.dwAddr	First address of the I/O range
I.IO.dwBytes	Length (in bytes) of the I/O range
I.IO.dwBar	Base Address Register number
I.Int	Interrupt item (ITEM_INTERRUPT) information
I.Int.dwInterrupt	Physical interrupt request (IRQ) number
I.Bus	Bus item (ITEM_BUS) information
I.Bus.dwBusType	Device's bus type. For PCI devices the bus type should be WD_BUS_PCI
I.Bus.dwBusNum	The device's bus number

Name	Description
I.Bus.dwSlotFunc	Slot and function information for the device: The lower three bits represent the function number and the remaining bits represent the slot number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot number of 0x10 (remaining bits: 10000).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

REMARKS

- The resources information is obtained from the operating system's Plug-and-Play manager, unless the information is not available, in which case it is read directly from the PCI configuration registers.
Note: On Windows, you must install an **INF file** file, which registers your device with WinDriver, before calling this function (see section [14.4](#) regarding creation of INF files with WinDriver).
- If the Interrupt Request (IRQ) number is obtained from the Plug-and-Play manager, it is mapped, and therefore may differ from the physical IRQ number.

A.2.7 WDC_PcmciaGetDeviceInfo()

PURPOSE

- Retrieves a PCMCIA device's resources information (memory and I/O ranges and interrupt information).

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaGetDeviceInfo (WD_PCMCIA_CARD_INFO *pDeviceInfo);
```

PARAMETERS

Name	Type	Input/Output
> pDeviceInfo	WD_PCMCIA_CARD_INFO*	Input/Output
□ pcmciaSlot	WD_PCMCIA_SLOT	Input
◆ uBus	BYTE	Input
◆ uSocket	BYTE	Input
◆ uFunction	BYTE	Input
□ Card	WD_CARD	Output
◆ dwItems	DWORD	Output
◆ Item	WD_ITEMS[WD_CARD_ITEMS]	Output
◇ item	DWORD	Output
◇ fNotSharable	DWORD	Output
◇ I	union	Output
◆ Mem	struct	Output
→ dwPhysicalAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwTransAddr	DWORD	N/A
→ dwUserDirectAddr	DWORD	N/A
→ dwCpuPhysicalAddr	DWORD	N/A
→ dwBar	DWORD	Output
◆ IO	struct	Output
→ dwAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwBar	DWORD	Output
◆ Int	struct	Output
→ dwInterrupt	DWORD	Output
→ dwOptions	DWORD	N/A

Name	Type	Input/Output
→ hInterrupt	DWORD	N/A
♦ Bus	struct	Output
→ dwBusType	WD_BUS_TYPE	Output
→ dwBusNum	DWORD	Output
→ dwSlotFunc	DWORD	Output
□ cVersion	CHAR [WD_PCMCIA_VERSION_LEN]	Output
□ cManufacturer	CHAR [WD_PCMCIA_ MANUFACTURER_LEN]	Output
□ cProductName	CHAR [WD_PCMCIA_ PRODUCTNAME_LEN]	Output
□ wManufacturerId	WORD	Output
□ wCardId	WORD	Output
□ wFuncId	WORD	Output

DESCRIPTION

Name	Description
pDeviceInfo	Pointer to a PCMCIA device information structure
pcmciaSlot	A PCMCIA device location information structure, which can be acquired by calling WDC_PcmciaScanDevices () [A.2.5]
uBus	Bus number
uSocket	Socket number
uFunction	Function number
Card	PCMCIA device resources information structure
dwItems	Number of items (resources) on the device
Item	Array of device resources (items) information structures
Item	Item type: ITEM_NONE / ITEM_INTERRUPT / ITEM_MEMORY / ITEM_IO / ITEM_BUS
fNotSharable	If true, only one application at a time can access the memory or I/O range, or monitor the device's interrupts
I	Union of resources data based on the item type (Item)
I.Mem	Memory item (ITEM_MEMORY) information
I.Mem.dwPhysicalAddr	First address of the physical memory range
I.Mem.dwBytes	Length (in bytes) of the memory range
I.Mem.dwBar	Base Address Register number
I.IO	I/O item (ITEM_IO) information
I.IO.dwAddr	Firs address of the I/O range
I.IO.dwBytes	Length (in bytes) of the I/O range

Name	Description
I.IO.dwBar	Base Address Register number
I.Int	Interrupt item (ITEM_INTERRUPT) information
I.Int.dwInterrupt	Physical interrupt request (IRQ) number
I.Bus	Bus item (ITEM_BUS) information
I.Bus.dwBusType	Device's bus type. For PCMCIA devices the bus type should be WD_BUS_PCMCIA
I.Bus.dwBusNum	The device's bus number
I.Bus.dwSlotFunc	Slot and function information for the device: The lower three bits represent the function number and the remaining bits represent the slot number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot number of 0x10 (remaining bits: 10000).
cVersion	Device version string
cManufacturer	Device manufacturer string
cProductName	Device product string
wManufacturerId	The device's manufacturer ID
wCardId	The device's device ID
wFuncId	The device's function ID

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- The resources information is obtained from the operating system's Plug-and-Play manager, unless the information is not available, in which case it is read directly from the PCMCIA configuration registers.
Note: On Windows, you must install an **INF file**, which registers your device with WinDriver, before calling this function (see section 14.4 regarding creation of INF files with WinDriver).
- If the Interrupt Request (IRQ) number is obtained from the Plug-and-Play manager, it is mapped, and therefore may differ from the physical IRQ number.

A.2.8 WDC_PciDeviceOpen()

PURPOSE

- Allocates and initializes a WDC PCI device structure, registers the device with WinDriver, and returns a handle to the device.

Among the operations performed by this function:

- Verifies that a non-shareable memory or I/O resource on the device has not already been registered exclusively
- Maps the physical memory ranges found on the device both to kernel-mode and user-mode address space, and stores the mapped addresses in the device structure for future use
- Saves device resources information required for supporting the communication with the device.
For example, the function saves the Interrupt Request (IRQ) number and the interrupt type (should be level sensitive for PCI), as well as retrieves and saves an interrupt handle, which are later used when the user calls functions to handle the device's interrupts.
- If the caller selects to use a Kernel PlugIn driver to communicate with the device, the function opens a handle to this driver and stores it for future use

PROTOTYPE

```
DWORD WINAPI WDC_PciDeviceOpen(WDC_DEVICE_HANDLE *phDev ,
    const WD_PCI_CARD_INFO *pDeviceInfo , const PVOID pDevCtx ,
    PVOID reserved , const CHAR *pcKPDriverName , PVOID pKPOpenData ) ;
```

PARAMETERS

Name	Type	Input/Output
➤ phDev	WDC_DEVICE_HANDLE*	Output
➤ pDeviceInfo	const WD_PCI_CARD_INFO*	Input
➤ pDevCtx	const PVOID	Input
➤ reserved	PVOID	
➤ pcKPDriverName	const CHAR *	Input
➤ pKPOpenData	PVOID	Input

DESCRIPTION

Name	Description
phDev	Pointer to a handle to the WDC device allocated by the function
pDeviceInfo	Pointer to a PCI device resources information structure – see WDC_PciGetDeviceInfo() [A.2.6]
pDevCtx	Pointer to device context information, which will be stored in the device structure
reserved	Reserved for future use
pcKPDriverName	Kernel PlugIn driver name. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.
pKPOpenData	Kernel PlugIn driver open data to be passed to WD_KernelPlugInOpen() [A.8.1]. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.
- If your card has a large memory range, which cannot be mapped entirely to the kernel virtual address space, you can modify the relevant item for this resource in the card resources information structure that you received from WDC_PciGetDeviceInfo() [A.2.6], and set the WD_ITEM_DO_NOT_MAP_KERNEL flag in the item's dwOptions field (pDeviceInfo->Card.Item[i].dwOptions) before passing the information structure (pDeviceInfo) to WDC_PciDeviceOpen(). The WD_ITEM_DO_NOT_MAP_KERNEL flag instructs the function to map the relevant memory range only to the user-mode virtual address space but not the kernel address space.

NOTE that if you select to set this flag, the device information structure that will be created by the function will not hold a kernel-mapped address for this resource (pAddrDesc[i].kptAddr in the WDC_DEVICE structure [A.3.3] for the relevant memory range will not be updated) and you will therefore not be able to rely on this mapping in calls to WinDriver's API or when accessing the memory from a Kernel PlugIn driver.

A.2.9 WDC_PcmciaDeviceOpen()

PURPOSE

- Allocates and initializes a WDC PCMCIA device structure, registers the device with WinDriver, and returns a handle to the device.

Among the operations performed by this function:

- Verifies that a non-shareable memory or I/O resource on the device has not already been registered exclusively
- Maps the device's physical memory ranges device both to kernel-mode and user-mode address space, and stores the mapped addresses in the device structure for future use
- Saves device resources information required for supporting future communication with the device.
For example, the function saves the Interrupt Request (IRQ) number and the interrupt type (edge-triggered / level sensitive), as well as retrieves and saves an interrupt handle, which are later used when the user calls functions to handle the device's interrupts.
- If the caller selects to use a Kernel PlugIn driver to communicate with the device, the function opens a handle to this driver and stores it for future use

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaDeviceOpen (WDC_DEVICE_HANDLE *phDev ,
    const WD_PCMCIA_CARD_INFO *pDeviceInfo , const PVOID pDevCtx ,
    PVOID reserved , const CHAR *pcKPDriverName , PVOID pKPOpenData) ;
```

PARAMETERS

Name	Type	Input/Output
➤ phDev	WDC_DEVICE_HANDLE*	Output
➤ pDeviceInfo	const WD_PCMCIA_CARD_INFO*	Input
➤ pDevCtx	const PVOID	Input
➤ reserved	PVOID	
➤ pcKPDriverName	const CHAR *	Input
➤ pKPOpenData	PVOID	Input

DESCRIPTION

Name	Description
phDev	Pointer to a handle to the WDC device allocated by the function
pDeviceInfo	Pointer to a PCMCIA device resources information structure – see WDC_PcmciaGetDeviceInfo() [A.2.7]
pDevCtx	Pointer to device context information, which will be stored in the device structure
reserved	Reserved for future use
pcKPDriverName	Kernel PlugIn driver name. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.
pKPOpenData	Kernel PlugIn driver open data to be passed to WD_KernelPlugInOpen() [A.8.1]. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.
- If your card has a large memory range, which cannot be mapped entirely to the kernel virtual address space, you can modify the relevant item for this resource in the card resources information structure that you received from WDC_PcmciaGetDeviceInfo() [A.2.7], and set the WD_ITEM_DO_NOT_MAP_KERNEL flag in the item's dwOptions field (pDeviceInfo->Card.Item[i].dwOptions) before passing the information structure (pDeviceInfo) to WDC_PcmciaDeviceOpen(). The WD_ITEM_DO_NOT_MAP_KERNEL flag instructs the function to map the relevant memory range only to the user-mode virtual address space but not the kernel address space.

NOTE that if you select to set this flag, the device information structure that will be created by the function will not hold a kernel-mapped address for this resource (pAddrDesc[i].kptAddr in the WDC_DEVICE structure [A.3.3] for the relevant memory range will not be updated) and you will therefore not be able to rely on this mapping in calls to WinDriver's API or when accessing the memory from a Kernel PlugIn driver.

A.2.10 WDC_IsaDeviceOpen()

PURPOSE

- Allocates and initializes a WDC ISA device structure, registers the device with WinDriver, and returns a handle to the device.

Among the operations performed by this function:

- Verifies that a non-shareable memory or I/O resource on the device has not already been registered exclusively
- Maps the device's physical memory ranges device both to kernel-mode and user-mode address space, and stores the mapped addresses in the device structure for future use
- Saves device resources information required for supporting future communication with the device.

For example, the function saves the Interrupt Request (IRQ) number and the interrupt type (edge-triggered / level sensitive), as well as retrieves and saves an interrupt handle, which are later used when the user calls functions to handle the device's interrupts.

- If the caller selects to use a Kernel PlugIn driver to communicate with the device, the function opens a handle to this driver and stores it for future use

PROTOTYPE

```
DWORD WINAPI WDC_IsaDeviceOpen(WDC_DEVICE_HANDLE *phDev,
    const WD_CARD *pDeviceInfo, const PVOID pDevCtx,
    PVOID reserved, const CHAR *pcKPDriverName, PVOID pKPOpenData);
```

PARAMETERS

Name	Type	Input/Output
➤ phDev	WDC_DEVICE_HANDLE*	Output
➤ pDeviceInfo	const WD_CARD*	Input
☐ dwItems	DWORD	Input
☐ Item	WD_ITEMS[WD_CARD_ITEMS]	Input
◆ item	DWORD	Input
◆ fNotSharable	DWORD	Input
◆ dwOptions	DWORD	Input

Name	Type	Input/Output
◆ I	union	Input
◇ Mem	struct	
◆ dwPhysicalAddr	DWORD	Input
◆ dwBytes	DWORD	Input
◆ dwTransAddr	DWORD	N/A
◆ dwUserDirectAddr	DWORD	N/A
◆ dwCpuPhysicalAddr	DWORD	N/A
◆ dwBar	DWORD	Input
◇ IO	struct	
◆ dwAddr	DWORD	Input
◆ dwBytes	DWORD	Input
◆ dwBar	DWORD	Input
◇ Int	struct	
◆ dwInterrupt	DWORD	Input
◆ dwOptions	DWORD	Input
◆ hInterrupt	DWORD	N/A
◇ Bus	struct	
◆ dwBusType	WD_BUS_TYPE	Input
◆ dwBusNum	DWORD	Input
◆ dwSlotFunc	DWORD	Input
➤ pDevCtx	const PVOID	Input
➤ reserved	PVOID	
➤ pcKPDriverName	const CHAR *	Input
➤ pKPOpenData	PVOID	Input

DESCRIPTION

Name	Description
phDev	Pointer to a handle to the WDC device allocated by the function
pDeviceInfo	Pointer to a structure that holds the device's resources information
dwItems	Number of items (resources) on the device
Item	Array of device resources (items) information structures
Item	Item type: ITEM_NONE / ITEM_INTERRUPT / ITEM_MEMORY / ITEM_IO / ITEM_BUS
fNotSharable	If true, only one application at a time can access the memory or I/O range, or monitor the device's interrupts

Name	Description
dwOptions	Any of the following WD_ITEM_OPTIONS flags: <ul style="list-style-type: none"> • WD_ITEM_DO_NOT_MAP_KERNEL: This flag instructs the function to avoid mapping a memory address range to the kernel virtual address space and map the memory only to the user-mode virtual address space. See the Remarks to this function for more information. NOTE: This flag is applicable only to memory items. • WD_ITEM_ALLOW_CACHE (Windows NT/2k/XP/ Server 2003 and CE only): Map the item's physical memory (I.Mem.dwPhysicalAddr) as cached. NOTE: This flag is applicable only to memory items that pertain to the host's RAM, as opposed to local memory on the card.
I	Union of resources data based on the item type (Item)
I.Mem	Memory item (ITEM_MEMORY) information
I.Mem.dwPhysicalAddr	First address of the physical memory range
I.Mem.dwBytes	Length (in bytes) of the memory range
I.Mem.dwBar	Base Address Register number
I.IO	I/O item (ITEM_IO) information
I.IO.dwAddr	First address of the I/O range
I.IO.dwBytes	Length (in bytes) of the I/O range
I.IO.dwBar	Base Address Register number
I.Int	Interrupt item (ITEM_INTERRUPT) information
I.Int.dwInterrupt	Physical interrupt request (IRQ) number
I.Int.dwOptions	Interrupt information bit mask, which can consist of a combination of any of the following flags (or zero (0) for not option): <ul style="list-style-type: none"> • INTERRUPT_LEVEL_SENSITIVE – Level Sensitive interrupt. Default - Interrupt is Edge-Triggered. ISA interrupts are normally Edge-Triggered. • INTERRUPT_CE_INT_ID – On Windows CE (unlike other operating systems), there is an abstraction of the physical interrupt number to a logical one. Setting this bit will instruct WinDriver to refer to the IRQ number (dwInterrupt) as a logical interrupt number and convert it to a physical interrupt number.
I.Bus	Bus item (ITEM_BUS) information
I.Bus.dwBusType	Device's bus type. The bus type in the call to this function should always be WD_BUS_ISA
I.Bus.dwBusNum	The device's bus number

Name	Description
I.Bus.dwSlotFunc	Slot and function information for the device: The lower three bits represent the function number and the remaining bits represent the slot number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot number of 0x10 (remaining bits: 10000).
pDevCtx	Pointer to device context information, which will be stored in the device structure
reserved	Reserved for future use
pcKPDriverName	Kernel PlugIn driver name. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.
pKPOpenData	Kernel PlugIn driver open data to be passed to WD_KernelPlugInOpen() [A.8.1]. If your application does not use a Kernel PlugIn driver, pass a NULL pointer for this argument.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.
- If your card has a large memory range, which cannot be mapped entirely to the kernel virtual address space, you can set the WD_ITEM_OPTIONS flag for the relevant memory WD_ITEMS structure (pDeviceInfo->Card.Item[i].dwOptions) in order to instruct the function to map this memory range only to the user-mode virtual address space but not the kernel address space.

NOTE that if you select to set this flag, the device information structure that will be created by the function will not hold a kernel-mapped address for this resource (pAddrDesc[i]kptAddr in the WDC_DEVICE structure [A.3.3] for the relevant memory range will not be updated) and you will therefore not be able to rely on this mapping in calls to WinDriver's API or when accessing the memory from a Kernel PlugIn driver.

A.2.11 WDC_PciDeviceClose()

PURPOSE

- Un-initializes a WDC PCI device structure and frees the memory allocated for it.

PROTOTYPE

```
DWORD WINAPI WDC_PciDeviceClose (WDC_DEVICE_HANDLE hDev) ;
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen() [A.2.8]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.

A.2.12 WDC_PcmciaDeviceClose()

PURPOSE

- Un-initializes a WDC PCMCIA device structure and frees the memory allocated for it.

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaDeviceClose(WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCMCIA device structure, returned by WDC_PcmciaDeviceOpen() [A.2.9]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.

A.2.13 WDC_IsaDeviceClose()

PURPOSE

- Un-initializes a WDC ISA device structure and frees the memory allocated for it.

PROTOTYPE

```
DWORD WINAPI WDC_IsaDeviceClose(WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC ISA device structure, returned by WDC_IsaDeviceOpen() [A.2.10]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.

A.2.14 WDC_CardCleanupSetup()

PURPOSE

• Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:

- The application exists abnormally.
- The application exits normally but without closing the specified card.
- If the `bForceCleanup` parameter is set to `TRUE`, the cleanup commands will also be performed when the specified card is closed.

PROTOTYPE

```
DWORD WDC_CardCleanupSetup(WDC_DEVICE_HANDLE hDev ,
    WD_TRANSFER *Cmd, DWORD dwCmds, BOOL bForceCleanup);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ Cmd	WD_TRANSFER*	Input
➤ dwCmds	DWORD	Input
➤ bForceCleanup	BOOL	Input

DESCRIPTION

Name	Description
➤ hDev	Handle to a WDC device, returned by <code>WDC_xxxDeviceOpen()</code> (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
➤ Cmd	Pointer to an array of cleanup transfer commands to be performed
➤ dwCmds	Number of cleanup commands in the <code>Cmd</code> array

Name	Description
➤ bForceCleanup	<p>If FALSE: The cleanup transfer commands (Cmd) will be performed in either of the following cases:</p> <ul style="list-style-type: none">• When the application exist abnormally.• When the application exits normally without closing the card by calling the relevant WDC_xxxDeviceClose() function (PCI [A.2.11] / PCMCIA [A.2.12] / ISA [A.2.13]). <p>If TRUE: The cleanup transfer commands will be performed both in the two cases described above, as well as in the following case:</p> <ul style="list-style-type: none">• When the relevant WD_xxxDeviceClose() function is called for the card.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.2.15 WDC_CallKerPlug()

PURPOSE

- Sends a message from a user-mode application to a Kernel PlugIn driver. The function passes a message ID from the application to the Kernel PlugIn's `KP_Call()` [A.9.4] function, which should be implemented to handle the specified message ID, and returns the result from the Kernel PlugIn to the user-mode application.

PROTOTYPE

```
DWORD WINAPI WDC_CallKerPlug (WDC_DEVICE_HANDLE hDev, DWORD dwMsg,
                              PVOID pData, PDWORD pdwResult);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwMsg	DWORD	Input
> pData	PVOID	Input
> pdwResult	pdwResult	Output

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by <code>WDC_xxxDeviceOpen()</code> (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwMsg	A message ID to pass to the Kernel PlugIn driver (specifically to <code>KP_Call()</code> [A.9.4])
pData	Pointer to data to pass to the Kernel PlugIn driver
pdwResult	Result returned by the Kernel PlugIn driver (<code>KP_Call()</code>) for the operation performed in the kernel as a result of the message that was sent

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.16 WDC_ReadMemXXX()

PURPOSE

• WDC_ReadMem8/16/32/64() reads 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, from a specified memory address. The address is read directly in the calling context (user mode / kernel mode).

PROTOTYPE

```
BYTE *WDC_ReadMem8( addr , off )
WORD *WDC_ReadMem16( addr , off )
UINT32 *WDC_ReadMem32( addr , off )
UINT64 *WDC_ReadMem64( addr , off )
```

PARAMETERS

Name	Type	Input/Output
➤ addr	DWORD	Input
➤ off	DWORD	Input

DESCRIPTION

Name	Description
addr	The memory address space to read from
off	The offset from the beginning of the specified address space (addr) to read from

RETURN VALUE

Returns the data that was read from the specified address.

A.2.17 WDC_WriteMemXXX()**PURPOSE**

• WDC_WriteMem8/16/32/64() writes 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, to a specified memory address. The address is written to directly in the calling context (user mode / kernel mode).

PROTOTYPE

```
void WDC_WriteMem8(addr, off, val)
void WDC_WriteMem16(addr, off, val)
void WDC_WriteMem32(addr, off, val)
void WDC_WriteMem64(addr, off, val)
```

PARAMETERS

Name	Type	Input/Output
➤ addr	DWORD	Input
➤ off	DWORD	Input
➤ val	BYTE / WORD / UINT32 / UINT64	Input

DESCRIPTION

Name	Description
addr	The memory address space to read from
off	The offset from the beginning of the specified address space (addr) to read from
val	The data to write to the specified address

RETURN VALUE

None

A.2.18 WDC_ReadAddrXXX()**PURPOSE**

• WDC_ReadAddr8/16/32/64() reads 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, from a specified memory or I/O address.

PROTOTYPE

```
DWORD WINAPI WDC_ReadAddr8(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, BYTE *val);

DWORD WINAPI WDC_ReadAddr16(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, WORD *val);

DWORD WINAPI WDC_ReadAddr32(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, UINT32 *val);

DWORD WINAPI WDC_ReadAddr64(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, UINT64 *val);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ dwAddrSpace	DWORD	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE* / WORD* / UINT32* / UINT64*	Output

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwAddrSpace	The memory or I/O address space to read from
dwOffset	The offset from the beginning of the specified address space (dwAddrSpace) to read from

Name	Description
val	Pointer to a buffer to be filled with the data that is read from the specified address

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.19 WDC_WriteAddrXXX()**PURPOSE**

• WDC_WriteAddr8/16/32/64() writes 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, to a specified memory or I/O address.

PROTOTYPE

```

DWORD WINAPI WDC_WriteAddr8(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, BYTE val)

DWORD WINAPI WDC_WriteAddr16(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, WORD val);

DWORD WINAPI WDC_WriteAddr32(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, UINT32 val);

DWORD WINAPI WDC_WriteAddr64(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, UINT64 val);

```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ dwAddrSpace	DWORD	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE / WORD / UINT32 / UINT64	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwAddrSpace	The memory or I/O address space to write to
dwOffset	The offset from the beginning of the specified address space (dwAddrSpace) to write to
val	The data to write to the specified address

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.20 WDC_ReadAddrBlock()**PURPOSE**

- Reads a block of data from the device.

PROTOTYPE

```
DWORD WINAPI WDC_ReadAddrBlock(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, DWORD dwBytes, PVOID pData,
    WDC_ADDR_MODE mode, WDC_ADDR_RW_OPTIONS options)
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwAddrSpace	DWORD	Input
> dwOffset	DWORD	Input
> dwBytes	DWORD	Input
> pData	PVOID	Output
> mode	WDC_ADDR_MODE	Input
> options	WDC_ADDR_RW_OPTIONS	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwAddrSpace	The memory or I/O address space to read from
dwOffset	The offset from the beginning of the specified address space (dwAddrSpace) to read from
dwBytes	The number of bytes to read
pData	Pointer to a buffer to be filled with the data that is read from the device
mode	The read access mode – see WDC_ADDR_MODE [A.2.1.4]
options	A bit mask that determines how the data will be read – see WDC_ADDR_RW_OPTIONS [A.2.1.5]. The function automatically sets the WDC_RW_BLOCK flag.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.21 WDC_WriteAddrBlock()**PURPOSE**

- Writes a block of data to the device.

PROTOTYPE

```
DWORD WINAPI WDC_WriteAddrBlock(WDC_DEVICE_HANDLE hDev,
    DWORD dwAddrSpace, DWORD dwOffset, DWORD dwBytes, PVOID pData,
    WDC_ADDR_MODE mode, WDC_ADDR_RW_OPTIONS options)
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwAddrSpace	DWORD	Input
> dwOffset	DWORD	Input
> dwBytes	DWORD	Input
> pData	PVOID	Input
> mode	WDC_ADDR_MODE	Input
> options	WDC_ADDR_RW_OPTIONS	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwAddrSpace	The memory or I/O address space to write to
dwOffset	The offset from the beginning of the specified address space (dwAddrSpace) to write to
dwBytes	The number of bytes to write
pData	Pointer to a buffer that holds the data to write to the device
mode	The write access mode – see WDC_ADDR_MODE [A.2.1.4]
options	A bit mask that determines how the data will be written – see WDC_ADDR_RW_OPTIONS [A.2.1.5]. The function automatically sets the WDC_RW_BLOCK flag.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.22 WDC_MultiTransfer()**PURPOSE**

- Performs a group of memory and/or I/O read/write transfers.

PROTOTYPE

```
DWORD WINAPI WDC_MultiTransfer (WD_TRANSFER *pTrans , DWORD dwNumTrans ) ;
```

PARAMETERS

Name	Type	Input/Output
> pTrans	WD_TRANSFER*	
□ cmdTrans	DWORD	Input
□ dwPort	DWORD	Input
□ dwBytes	DWORD	Input
□ fAutoinc	DWORD	Input
□ dwOptions	DWORD	Input
□ Data	union	
□ Data.Byte	BYTE	Input/Output
□ Data.Word	WORD	Input/Output
□ Data.Dword	DWORD	Input/Output
□ Data.Qword	QWORD	Input/Output
□ Data.pBuffer	PVOID	Input/Output
> dwNumTrans	DWORD	Input

DESCRIPTION

Name	Description
pTrans	Pointer to an array of transfer commands information structures

Name	Description
cmdTrans	<p>A value indicating the type of transfer command to perform – see definition of the WD_TRANSFER_CMD enumeration in windrvr.h.</p> <p>The transfer commands conform to the following format: <dir><p><string><size></p> <ul style="list-style-type: none"> • <i>dir</i>: <i>R</i> to read, <i>W</i> to write • <i>p</i>: <i>P</i> for an I/O port (address), <i>M</i> for a memory port (address) • <i>string</i>: <i>S</i> for a string (block) transfer, otherwise a single transfer • <i>size</i>: BYTE, WORD, DWORD or QWORD
dwPort	The I/O port address / kernel-mapped virtual memory address, which has been stored in the relevant device (WDC_DEVICE [A.3.3]): <code>dev.pAddrDesc[i].kptAddr</code> (where <i>i</i> is the index of the desired address space).
fAutoinc	<p>Relevant only for string (block) transfers:</p> <p>If TRUE, the I/O or memory read/write port/address will be incremented after each block that is transferred;</p> <p>If FALSE, all data is transferred to/from the same port/address.</p>
dwOptions	Must be 0
Data	The data buffer for the transfer
Data.Byte	Used for 8-bit transfers
Data.Word	Used for 16-bit transfers
Data.Dword	Used for 32-bit transfers
Data.Qword	Used for 64-bit transfers
Data.pBuffer	Used for string (block) transfers – a pointer to the data buffer for the transfer
dwNumTrans	Number of transfer commands in the pTrans array

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- The transfers are performed using the lower-level WD_MultiTransfer() [A.4.15] WinDriver function, which reads/writes the specified addresses in the kernel.
- Memory addresses are read/written in the kernel (like I/O addresses) and NOT

directly in the user mode, therefore the port addresses passed to this function, for both memory and I/O addresses, must be the kernel-mode mappings of the physical addresses, which are stored in the device structure [\[A.3.3\]](#).

A.2.23 WDC_AddrSpaceIsActive()

PURPOSE

- Checks if the specified memory or I/O address space is active – i.e. if its size is not zero (0).

PROTOTYPE

```
BOOL WINAPI WDC_AddrSpaceIsActive(WDC_DEVICE_HANDLE hDev, DWORD dwAddrSpace);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ dwAddrSpace	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
dwAddrSpace	The memory or I/O address space to look for

RETURN VALUE

Returns TRUE if the specified address space is active; otherwise returns FALSE.

A.2.24 WDC_PciReadCfgBySlot()

PURPOSE

- Reads data from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).
The device is identified by its location on the PCI bus.

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciReadCfgBySlot(WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , PVOID pData , DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
> pPciSlot	WD_PCI_SLOT	Input
□ dwBus	DWORD	Input
□ dwSlot	DWORD	Input
□ dwFunction	DWORD	Input
> dwOffset	DWORD	Input
> pData	PVOID	Output
> dwBytes	DWORD	Input

DESCRIPTION

Name	Description
pPciSlot	Pointer to device location information structure, which can be acquired by calling WDC_PciScanDevices() [A.2.4]
dwBus	Bus number
dwSlot	Slot number
dwFunction	Function number
dwOffset	The offset from the beginning of the PCI configuration space to read from
pData	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Name	Description
dwBytes	The number of bytes to read

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.2.25 WDC_PciWriteCfgBySlot()

PURPOSE

- Write data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).
The device is identified by its location on the PCI bus.

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciWriteCfgBySlot(WD_PCI_SLOT *pPciSlot, DWORD dwOffset,
    UINT64 val, DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
➤ pPciSlot	WD_PCI_SLOT	Input
☐ dwBus	DWORD	Input
☐ dwSlot	DWORD	Input
☐ dwFunction	DWORD	Input
➤ dwOffset	DWORD	Input
➤ pData	PVOID	Input
➤ dwBytes	DWORD	Input

DESCRIPTION

Name	Description
pPciSlot	Pointer to device location information structure, which can be acquired by calling WDC_PciScanDevices() [A.2.4]
dwBus	Bus number
dwSlot	Slot number
dwFunction	Function number
dwOffset	The offset from the beginning of the PCI configuration space to write to
pData	Pointer to a data buffer that holds the data to write
dwBytes	The number of bytes to write

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.26 WDC_PciReadCfgBySlotXXX()

PURPOSE

• WDC_PciReadCfgBySlot8/16/32/64() reads 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux). The device is identified by its location on the PCI bus.

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciReadCfgRegBySlot8(WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , BYTE *val )

DWORD WINAPI WDC_PciReadCfgReg1BySlot6(WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , WORD *val )

DWORD WINAPI WDC_PciReadCfgReg32BySlot(WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , UINT32 *val )

DWORD WINAPI WDC_PciReadCfgReg64BySlot(WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , UINT64 *val )
```

PARAMETERS

Name	Type	Input/Output
➤ pPciSlot	WD_PCI_SLOT	Input
☐ dwBus	DWORD	Input
☐ dwSlot	DWORD	Input
☐ dwFunction	DWORD	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE* / WORD* / UINT32* / UINT64*	Output

DESCRIPTION

Name	Description
pPciSlot	Pointer to device location information structure, which can be acquired by calling <code>WDC_PciScanDevices()</code> [A.2.4]
dwBus	Bus number
dwSlot	Slot number
dwFunction	Function number
dwOffset	The offset from the beginning of the PCI configuration space to read from
val	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.27 WDC_PciWriteCfgBySlotXXX()

PURPOSE

•WDC_PciWriteCfgBySlot8/16/32/64() writes 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux). The device is identified by its location on the PCI bus.

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciWriteCfgRegBySlot8 (WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , BYTE val)

DWORD WINAPI WDC_PciWriteCfgRegBySlot16 (WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , WORD val)

DWORD WINAPI WDC_PciWriteCfgRegBySlot32 (WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , UINT32 val)

DWORD WINAPI WDC_PciWriteCfgRegBySlot64 (WD_PCI_SLOT *pPciSlot ,
    DWORD dwOffset , UINT64 val)
```

PARAMETERS

Name	Type	Input/Output
➤ pPciSlot	WD_PCI_SLOT	Input
☐ dwBus	DWORD	Input
☐ dwSlot	DWORD	Input
☐ dwFunction	DWORD	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE / WORD / UINT32 / UINT64	Input

DESCRIPTION

Name	Description
pPciSlot	Pointer to device location information structure, which can be acquired by calling <code>WDC_PciScanDevices()</code> [A.2.4]
dwBus	Bus number
dwSlot	Slot number
dwFunction	Function number
dwOffset	The offset from the beginning of the PCI configuration space to read from
val	The data to write to the PCI configuration space

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.28 WDC_PciReadCfg()

PURPOSE

- Reads data from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciReadCfg(WDC_DEVICE_HANDLE hDev ,
    DWORD dwOffset , PVOID pData , DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwOffset	DWORD	Input
> pData	PVOID	Output
> dwBytes	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCI device structure, returned by <code>WDC_PciDeviceOpen()</code> [A.2.8]
dwOffset	The offset from the beginning of the PCI configuration space to read from
pData	Pointer to a buffer to be filled with the data that is read from the PCI configuration space
dwBytes	The number of bytes to read

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.29 WDC_PciWriteCfg()

PURPOSE

- Writes data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciWriteCfg(WDC_DEVICE_HANDLE hDev,
                             DWORD dwOffset, PVOID pData, DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwOffset	DWORD	Input
> pData	PVOID	Input
> dwBytes	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCI device structure, returned by <code>WDC_PciDeviceOpen()</code> [A.2.8]
dwOffset	The offset from the beginning of the PCI configuration space to write to
pData	Pointer to a data buffer that holds the data to write
dwBytes	The number of bytes to write

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.30 WDC_PciReadCfgXXX()

PURPOSE

• WDC_PciReadCfg8/16/32/64() reads 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciReadCfgReg8(WDC_DEVICE_HANDLE hDev,
                                DWORD dwOffset, BYTE *val)

DWORD WINAPI WDC_PciReadCfgReg16(WDC_DEVICE_HANDLE hDev,
                                 DWORD dwOffset, WORD *val)

DWORD WINAPI WDC_PciReadCfgReg32(WDC_DEVICE_HANDLE hDev,
                                 DWORD dwOffset, UINT32 *val)

DWORD WINAPI WDC_PciReadCfgReg64(WDC_DEVICE_HANDLE hDev,
                                 DWORD dwOffset, UINT64 *val)
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE* / WORD* / UINT32* / UINT64*	Output

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen() [A.2.8]

Name	Description
dwOffset	The offset from the beginning of the PCI configuration space to read from
val	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.2.31 WDC_PciWriteCfgXXX()

PURPOSE

•WDC_PciWriteCfg8/16/32/64() writes 1 byte (8 bits) / 2 bytes (16 bits) / 4 bytes (32 bits) / 8 bytes (64 bits), respectively, to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space (on Windows/Linux).

For Windows/Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WINAPI WDC_PciWriteCfgReg8 (WDC_DEVICE_HANDLE hDev ,
    DWORD dwOffset , BYTE val)

DWORD WINAPI WDC_PciWriteCfgReg16 (WDC_DEVICE_HANDLE hDev ,
    DWORD dwOffset , WORD val)

DWORD WINAPI WDC_PciWriteCfgReg32 (WDC_DEVICE_HANDLE hDev ,
    DWORD dwOffset , UINT32 val)

DWORD WINAPI WDC_PciWriteCfgReg64 (WDC_DEVICE_HANDLE hDev ,
    DWORD dwOffset , UINT64 val)
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ dwOffset	DWORD	Input
➤ val	BYTE / WORD / UINT32 / UINT64	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen() [A.2.8]

Name	Description
dwOffset	The offset from the beginning of the PCI configuration space to read from
val	The data to write to the PCI configuration space

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.32 WDC_PcmciaReadAttribSpace()

PURPOSE

- Reads data from a specified offset in a PCMCIA device's attribute space.

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaReadAttribSpace(WDC_DEVICE_HANDLE hDev,  
    DWORD dwOffset, PVOID pData, DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwOffset	DWORD	Input
> pData	PVOID	Output
> dwBytes	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCMCIA device structure, returned by <code>WDC_PcmciaDeviceOpen()</code> [A.2.9]
dwOffset	The offset from the beginning of the PCMCIA attribute space to read from
pData	Pointer to a buffer to be filled with the data that is read from the PCMCIA attribute space
dwBytes	The number of bytes to read

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.33 WDC_PcmciaWriteAttribSpace()

PURPOSE

- Writes data to a specified offset in a PCMCIA device's attribute space.

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaWriteAttribSpace(WDC_DEVICE_HANDLE hDev ,  
    DWORD dwOffset , PVOID pData , DWORD dwBytes);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwOffset	DWORD	Input
> pData	PVOID	Input
> dwBytes	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCMCIA device structure, returned by <code>WDC_PcmciaDeviceOpen()</code> [A.2.9]
dwOffset	The offset from the beginning of the PCMCIA attribute space to write to
pData	Pointer to a data buffer that holds the data to write
dwBytes	The number of bytes to write

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.2.34 WDC_PcmciaSetWindow()**PURPOSE**

- Modifies the settings of the PCMCIA bus controller's memory window.

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaSetWindow (WDC_DEVICE_HANDLE hDev ,
                                   WD_PCMCIA_ACC_SPEED speed , WD_PCMCIA_ACC_WIDTH width , DWORD dwCardBase );
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> speed	WD_PCMCIA_ACC_SPEED	Input
> width	WD_PCMCIA_ACC_WIDTH	Input
> dwCardBase	DWORD	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCMCIA device structure, returned by WDC_PcmciaDeviceOpen() [A.2.9]
speed	The access speed to the PCMCIA bus. Can be any of the following WD_PCMCIA_ACC_SPEED enumeration values: <ul style="list-style-type: none"> • WD_PCMCIA_ACC_SPEED_DEFAULT: Use the default access speed • WD_PCMCIA_ACC_SPEED_250NS: 250 ns • WD_PCMCIA_ACC_SPEED_200NS: 200 ns • WD_PCMCIA_ACC_SPEED_150NS: 150 ns • WD_PCMCIA_ACC_SPEED_100NS: 100 ns
width	The PCMCIA bus width. Can be any of the following WD_PCMCIA_ACC_WIDTH enumeration values: <ul style="list-style-type: none"> • WD_PCMCIA_ACC_WIDTH_DEFAULT: Use the default bus width • WD_PCMCIA_ACC_WIDTH_8BIT: 8 bit • WD_PCMCIA_ACC_WIDTH_16BIT: 16 bit

Name	Description
dwCardBase	The offset in the PCMCIA device's memory from which the memory mapping begins

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

A.2.35 WDC_PcmciaSetVpp()

PURPOSE

- Modifies the power level of the PCMCIA bus controller's Voltage Power Pin (Vpp).

PROTOTYPE

```
DWORD WINAPI WDC_PcmciaSetVpp(WDC_DEVICE_HANDLE hDev, WD_PCMCIA_VPP vpp);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ vpp	WD_PCMCIA_VPP	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC PCMCIA device structure, returned by WDC_PcmciaDeviceOpen() [A.2.9]
vpp	The power level of the PCMCIA controller's Voltage Power Pin (Vpp). Can be any of the following WD_PCMCIA_VPP enumeration values: <ul style="list-style-type: none">• WD_PCMCIA_VPP_DEFAULT: Use the default power level of the PCMCIA Vpp pin• WD_PCMCIA_VPP_OFF: Set the voltage on the Vpp pin to zero (disable)• WD_PCMCIA_VPP_ON: Set the voltage on the Vpp pin to 12V (enable)• WD_PCMCIA_VPP_AS_VCC: Set the voltage on the Vpp pin to equal that of the Vcc pin

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.2.36 WDC_DMAContigBufLock()**PURPOSE**

• Locks a contiguous physical memory buffer, which can be used safely for Direct Memory Access (DMA), and maps the physical memory to both kernel- and user-mode virtual address spaces.

PROTOTYPE

```
DWORD WINAPI WDC_DMAContigBufLock(WDC_DEVICE_HANDLE hDev, PVOID *ppBuf,
    DWORD dwOptions, DWORD dwDMABufSize, WD_DMA **ppDma);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> ppBuf	PVOID*	Output
> dwOptions	DWORD	Input
> dwDMABufSize	DWORD	Input
> ppDma	WD_DMA**	Output
□ hDma	DWORD	Output
□ pUserAddr	PVOID	Output
□ pKernelAddr	KPTR	Output
□ dwBytes	DWORD	Output
□ dwOptions	DWORD	Output
□ dwPages	DWORD	Output
□ hCard	DWORD	Output
□ Page	WD_DMA_PAGE [WD_DMA_PAGES]	Output
◆ pPhysicalAddr	KPTR	Output
◆ dwBytes	DWORD	Output

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10]). NOTE: This field can also be set to NULL in order to lock a contiguous physical memory buffer with no relation to a specific device.
ppBuf	Pointer to a pointer to be filled by the function with the user-mode mapped address of the allocated DMA buffer
dwOptions	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): <ul style="list-style-type: none"> •DMA_READ_FROM_DEVICE: Memory is locked to be read from the device and written to the host ; OR <ul style="list-style-type: none"> DMA_WRITE_TO_DEVICE: Memory is locked to be read from the host and written to the device. Note: You must set either the DMA_READ_FROM_DEVICE or the DMA_WRITE_TO_DEVICE flag, and you cannot set both flags together. <ul style="list-style-type: none"> •DMA_ALLOW_CACHE: Allow caching of the memory. •DMA_CTG_KBUF_BELOW_16M: Allocate the physical DMA buffer within the lower 16MB of the main memory
dwDMABufSize	The size (in bytes) of the DMA buffer
ppDma	Pointer to a pointer to a DMA buffer information structure, which is allocated by the function. The pointer to this structure (*ppDma) should be passed to WDC_DMABufUnlock() [A.2.38] when the DMA buffer is no longer needed.
hDma	Handle to the allocated DMA buffer or 0 if the allocation failed.
pUserAddr	User-mode mapped address of the DMA buffer
pKernelAddr	Kernel-mode mapped address of the DMA buffer
dwBytes	The size of the DMA buffer (in bytes) (<=> dwDMABufSize parameter)
dwOptions	A bit mask of the options used for the DMA buffer allocation (<=> options parameter)
dwPages	Number of physical memory pages used for the allocated buffer <=> the number of elements in the Page array (see below.) The number of pages is always one for contiguous buffer DMA.

Name	Description
hCard	Low-level WinDriver card handle, acquired by WDC_xxxDeviceOpen() (by calling WD_CardRegister() [A.4.11]) and stored in the WDC device structure
Page	Array of physical memory pages information structures. For contiguous buffer DMA this array always holds only one element (see dwPages).
pPhysicalAddr	The page's physical address
dwBytes	The page's size (in bytes)

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- When calling this function you do **not** need to set the DMA_KERNEL_BUFFER_ALLOC flag, since the function sets this flag automatically.
- This function is currently only supported from the user mode.

A.2.37 WDC_DMASGBufLock()**PURPOSE**

- Locks a pre-allocated user-mode memory buffer and returns a list of the corresponding physical pages that were allocated.
- Maps the allocated buffer to the kernel-mode address space and returns the kernel-mapped address.

PROTOTYPE

```
DWORD WINAPI WDC_DMASGBufLock(WDC_DEVICE_HANDLE hDev, PVOID pBuf,
    DWORD dwOptions, DWORD dwDMABufSize, WD_DMA **ppDma);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> pBuf	PVOID	Input
> dwOptions	DWORD	Input
> dwDMABufSize	DWORD	Input
> ppDma	WD_DMA**	Output
□ hDma	DWORD	Output
□ pUserAddr	PVOID	Output
□ pKernelAddr	KPTR	Output
□ dwBytes	DWORD	Output
□ dwOptions	DWORD	Output
□ dwPages	DWORD	Output
□ hCard	DWORD	Output
□ Page	WD_DMA_PAGE [WD_DMA_PAGES]	Output
◆ pPhysicalAddr	KPTR	Output
◆ dwBytes	DWORD	Output

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])
pBuf	Pointer to a user-mode buffer to be mapped to the allocated physical DMA buffer(s)
dwOptions	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): <ul style="list-style-type: none"> • DMA_READ_FROM_DEVICE: Memory is locked to be read from the device and written to the host ; OR <ul style="list-style-type: none"> • DMA_WRITE_TO_DEVICE: Memory is locked to be read from the host and written to the device. Note: You must set either the DMA_READ_FROM_DEVICE or the DMA_WRITE_TO_DEVICE flag, and you cannot set both flags together. <ul style="list-style-type: none"> • DMA_ALLOW_CACHE: Allow caching of the memory.
dwDMABufSize	The size (in bytes) of the DMA buffer
ppDma	Pointer to a pointer to a DMA buffer information structure, which is allocated by the function. The pointer to this structure (*ppDma) should be passed to WDC_DMABufUnlock() [A.2.38] when the DMA buffer is no longer needed.
hDma	Handle to the allocated DMA buffer or 0 if the allocation failed.
pUserAddr	User-mode mapped address of the DMA buffer (<=> pBuf parameter)
pKernelAddr	Kernel-mode mapped address of the DMA buffer
dwBytes	The size of the DMA buffer (in bytes) (<=> dwDMABufSize parameter)
dwOptions	A bit mask of the options used for the DMA buffer allocation (<=> options parameter)
dwPages	Number of physical memory pages used for the allocated buffer <=> the number of elements in the Page array (see below)
hCard	Low-level WinDriver card handle, acquired by WDC_xxxDeviceOpen() (by calling WD_CardRegister() [A.4.11]) and stored in the WDC device structure
Page	Array of physical memory pages information structures
pPhysicalAddr	The page's physical address
dwBytes	The page's size (in bytes)

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

REMARKS

- When calling the function to allocate large buffers (> 1MB) you do **not** need to set the DMA_LARGE_BUFFER flag, which is used for allocation of large Scatter/Gather DMA buffers using the WD_DMALock() function [[A.4.16](#)], since WDC_DMASGBufLock() handles this for you.
- This function is currently only supported from the user mode.

A.2.38 WDC_DMABufUnlock()

PURPOSE

- Unlocks and frees the memory allocated for a DMA buffer by a previous call to WDC_DMAContigBufLock() [A.2.36] or WDC_DMASGBufLock() [A.2.37].

PROTOTYPE

```
DWORD WINAPI WDC_DMABufUnlock(WD_DMA *pDma)
```

PARAMETERS

Name	Type	Input/Output
> pDma	WD_DMA*	Input

DESCRIPTION

Name	Description
pDma	Pointer to a DMA information structure, received from a previous call to WDC_DMAContigBufLock() [A.2.36] (for a Contiguous DMA buffer) or WDC_DMASGBufLock() [A.2.37] (for a Scatter/Gather DMA buffer) – *ppDma returned by these functions

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function is currently only supported from the user mode.

A.2.39 WDC_DMASyncCpu()**PURPOSE**

- Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.

NOTE: This function should be called before performing a DMA transfer (see Remarks below).

PROTOTYPE

```
DWORD DLLCALLCONV WDC_DMASyncCpu(WD_DMA *pDma) ;
```

PARAMETERS

Name	Type	Input/Output
> pDma	WD_DMA*	Input

DESCRIPTION

Name	Description
pDma	Pointer to a DMA information structure, received from a previous call to WDC_DMAContigBufLock() [A.2.36] (for a Contiguous DMA buffer) or WDC_DMASGBufLock() [A.2.37] (for a Scatter/Gather DMA buffer) – *ppDma returned by these functions

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- An asynchronous DMA read or write operation accesses data in memory, not in the processor (CPU) cache, which resides between the CPU and the host's physical memory. Unless the CPU cache has been flushed, by calling WDC_DMASyncCpu(), just before a read transfer, the data transferred into system memory by the DMA operation could be overwritten with stale data if the

CPU cache is flushed later. Unless the CPU cache has been flushed by calling `WDC_DMASyncCpu()` just before a write transfer, the data in the CPU cache might be more up-to-date than the copy in memory.

- This function is currently only supported from the user mode.

A.2.40 WDC_DMASyncIo()**PURPOSE**

- Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.

NOTE: This function should be called after performing a DMA transfer (see Remarks below).

PROTOTYPE

```
DWORD WINAPI WDC_DMASyncIo(WD_DMA *pDma) ;
```

PARAMETERS

Name	Type	Input/Output
> pDma	WD_DMA*	Input

DESCRIPTION

Name	Description
pDma	Pointer to a DMA information structure, received from a previous call to WDC_DMAContigBufLock() [A.2.36] (for a Contiguous DMA buffer) or WDC_DMASGBufLock() [A.2.37] (for a Scatter/Gather DMA buffer) – *ppDma returned by these functions

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- After a DMA transfer has been completed, the data can still be in the I/O cache, which resides between the host's physical memory and the bus-master DMA device, but not yet in the host's main memory. If the CPU accesses the memory, it might read the wrong data from the CPU cache. To ensure a consistent view of the memory for the CPU, you should call WDC_DMASyncIo()

after a DMA transfer in order to flush the data from the I/O cache and update the CPU cache with the new data. The function also flushes additional caches and buffers between the device and memory, such as caches associated with bus extenders or bridges.

- This function is currently only supported from the user mode.

A.2.41 WDC_IntEnable()

PURPOSE

- Enable interrupt handling for the device.
- If the caller selects to handle the interrupts in the kernel, using a **Kernel PlugIn driver**, the Kernel PlugIn `KP_IntAtIrql()` function [A.9.8], which runs at high IRQ (Interrupt Request) level, will be invoked immediately when an interrupt is received.
- The function can receive transfer commands information, which will be performed by WinDriver at the kernel, at high IRQ level, when an interrupt is received. If a **Kernel PlugIn driver** is used to handle the interrupts, any transfer commands set by the caller will be executed by WinDriver after the Kernel PlugIn `KP_IntAtIrql()` function [A.9.8] completes its execution.

When handling level sensitive interrupts (such as PCI interrupts) from the **user mode**, without a Kernel PlugIn driver, you must prepare and pass to the function transfer commands for acknowledging the interrupt. When using a **Kernel PlugIn driver**, the information for acknowledging the interrupts should be implemented in the Kernel PlugIn `KP_IntAtIrql()` function [A.9.8], so the transfer commands are not required.

- The function receives a user-mode interrupt handler routine, which will be called by WinDriver after the kernel-mode interrupt processing is completed.

If the interrupts are handled using a **Kernel PlugIn driver**, the return value of the Kernel PlugIn deferred interrupt handler function (`KP_IntAtDpc()` [A.9.9]) will determine how many times (if at all) the user-mode interrupt handler will be called (provided `KP_IntAtDpc()` itself is executed - which is determined by the return value of the Kernel PlugIn `KP_IntAtIrql()` [A.9.8] function.)

PROTOTYPE

```
DWORD WINAPI WDC_IntEnable(
    WDC_DEVICE_HANDLE hDev ,
    WD_TRANSFER *pTransCmds ,
    DWORD dwNumCmds ,
    DWORD dwOptions ,
    INT_HANDLER funcIntHandler ,
    PVOID pData ,
    BOOL fUseKP);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input
➤ pTransCmds	WD_TRANSFER*	Input
❑ cmdTrans	DWORD	Input
❑ dwPort	DWORD	Input
❑ dwBytes	DWORD	Input
❑ fAutoinc	DWORD	Input
❑ dwOptions	DWORD	Input
❑ Data	union	
❑ Data.Byte	BYTE	Input/Output
❑ Data.Word	WORD	Input/Output
❑ Data.Dword	DWORD	Input/Output
❑ Data.Qword	QWORD	Input/Output
❑ Data.pBuffer	PVOID	Input/Output
➤ dwNumCmds	DWORD	Input
➤ dwOptions	DWORD	Input
➤ funcIntHandler	INT_HANDLER	Input
➤ pData	PVOID	Input
➤ fUseKP	BOOL	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])

Name	Description
pTransCmds	<p>An array of transfer commands information structures (WD_TRANSFER *) that define the operations to be performed at the kernel level upon the detection of an interrupt, or NULL if no transfer commands are required.</p> <p>NOTE: When handling level sensitive interrupts (such as PCI interrupts) without a Kernel PlugIn driver, you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received – see section 9.2 for details.</p> <p>The commands in the array can be either of the following:</p> <ul style="list-style-type: none"> • A read/write transfer command that conforms to the following format: <dir><p>_[S]<size> – see the description of the cmdTrans field of the WD_TRANSFER structure in section A.4.14 for details. • CMD_MASK: Interrupt mask command for determining the source of the interrupt: When this command is set, upon the arrival of an interrupt in the kernel WinDriver compares the value of the previous read command in the pTransCmds array with the mask that is set in the relevant Data field union member of the mask transfer command. For example, if pTransCmds[i-1].cmdTrans is RM_BYTE, WinDriver performs the following check: pTransCmds[i-1].Data.Byte & pTransCmds[i].Data.Byte. If the values match, the driver claims ownership of the interrupt and invokes your interrupt handler routine (funcIntHandler) when the control is returned to the user mode; otherwise, the driver rejects ownership of the interrupt, the interrupt handler routine is not invoked and the subsequent transfer commands in the pTransCmds array are not executed. <p>NOTE: A CMD_MASK command must be preceded by a read transfer command (RM_XXX / RP_XXX).</p>
dwNumCmds	Number of transfer commands in the pTransCmds array
dwOptions	<p>A bit mask of interrupt handling flags.</p> <p>Can be zero (0) for no option, or:</p> <ul style="list-style-type: none"> • INTERRUPT_CMD_COPY: If set, WinDriver will copy any data read in the kernel as a result of a read transfer command, and return it to the user within the relevant transfer command structure

Name	Description
funcIntHandler	A user-mode interrupt handler callback function, which will be executed after an interrupt is received and processed in the kernel. (The prototype of the interrupt handler – INT_HANDLER – is defined in windrvr_int_thread.h .)
pData	Data for the user-mode interrupt handler callback routine (funcIntHandler)
fUseKP	<p>If TRUE – The device's Kernel PlugIn driver's KP_IntAtIrql() function [A.9.8], which runs at high IRQ (Interrupt Request) level, will be executed immediately when an interrupt is received. (The Kernel PlugIn driver to be used for the device is passed to WDC_xxxDeviceOpen() and stored in the WDC device structure.)</p> <p>If the caller also passes transfer commands to the function (pTransCmds), these commands will be executed by WinDriver at the kernel, at high IRQ level, after KP_IntAtIrql() completes its execution.</p> <p>If KP_IntAtIrql() returns TRUE, the Kernel PlugIn deferred interrupt processing routine – KP_IntAtDpc() [A.9.9] – will be invoked. The return value of this function determines how many times (if at all) the user-mode interrupt handler (funcIntHandler) will be executed once the control returns to the user mode.</p> <p>If FALSE – When an interrupt is received, any transfer commands set by the user in pTransCmds will be executed by WinDriver at the kernel, at high IRQ level, and the user-mode interrupt handler routine (funcIntHandler) will be executed when the control returns to the user mode.</p>

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.
- The function enables interrupt handling in the software. After a successful call to this function you must physically enable generation of interrupts in the hardware (which you should be able to do by writing to the device from your code.)
- A successful call to this function must be followed with a call to

`WDC_IntDisable()` later on in the code, in order to disable the interrupts.
The `WDC_xxxDriverClose()` functions (PCI: [\[A.2.11\]](#), PCMCIA: [\[A.2.12\]](#),
ISA: [\[A.2.13\]](#)) call `WDC_IntDisable()` if the device's interrupts are enabled.

A.2.42 WDC_IntDisable()

PURPOSE

- Disables interrupt handling for the device, pursuant to a previous call to WDC_IntEnable() [A.2.41].

PROTOTYPE

```
DWORD WINAPI WDC_IntDisable(WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.

A.2.43 WDC_IntIsEnabled()

PURPOSE

- Checks if a device's interrupts are currently enabled.

PROTOTYPE

```
BOOL DLLCALLCONV WDC_IntIsEnabled (WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])

RETURN VALUE

Returns TRUE if the device's interrupts are enabled; otherwise returns FALSE.

A.2.44 WDC_EventRegister()

PURPOSE

- Registers the application to receive Plug-and-Play and power management events notifications for the device.

PROTOTYPE

```
DWORD WINAPI WDC_EventRegister(WDC_DEVICE_HANDLE hDev, DWORD dwActions,
    EVENT_HANDLER funcEventHandler, PVOID pData, BOOL fUseKP);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input
> dwActions	DWORD	Input
> funcEventHandler	WDC_EVENT_HANDLER	Input
> pData	PVOID	Input
> fUseKP	BOOL	Input

DESCRIPTION

Name	Description
hDev	Handle to a Plug and Play WDC device, returned by WDC_PciDeviceOpen() [A.2.8] or WDC_PcmciaDeviceOpen() [A.2.9]

Name	Description
dwActions	<p>A bit mask of flags indicating which events to register to:</p> <p>Plug and Play events:</p> <ul style="list-style-type: none"> • WD_INSERT - Device inserted • WD_REMOVE - Device removed <p>Device power state change events:</p> <ul style="list-style-type: none"> • WD_POWER_CHANGED_D0 - Full power • WD_POWER_CHANGED_D1 - Low sleep • WD_POWER_CHANGED_D2 - Medium sleep • WD_POWER_CHANGED_D3 - Full sleep • WD_POWER_SYSTEM_WORKING - Fully on <p>Systems power state:</p> <ul style="list-style-type: none"> • WD_POWER_SYSTEM_SLEEPING1 - Fully on but sleeping • WD_POWER_SYSTEM_SLEEPING2 - CPU off, memory on, PCI/PCMCIA on • WD_POWER_SYSTEM_SLEEPING3 - CPU off, Memory is in refresh, PCI/PCMCIA on aux power • WD_POWER_SYSTEM_HIBERNATE - OS saves context before shutdown • WD_POWER_SYSTEM_SHUTDOWN - No context saved
funcEventHandler	<p>A user-mode event handler callback function, which will be called when an event for which the caller registered to receive notifications (see dwActions) occurs . (The prototype of the event handler – EVENT_HANDLER – is defined in windrvr_events.h.)</p>
pData	<p>Data for the user-mode event handler callback routine (funcEventHandler)</p>
fUseKP	<p>If TRUE – When an event for which the caller registered to receive notifications (dwActions) occurs, the device's Kernel PlugIn driver's KP_Event () function [A.9.5] will be called. (The Kernel PlugIn driver to be used for the device is passed to WDC_xxxDeviceOpen () and stored in the WDC device structure.)</p> <p>If this function returns TRUE, the user-mode events handler callback function (funcEventHandler) will be called when the kernel-mode event processing is completed.</p> <p>If FALSE – When an event for which the caller registered to receive notifications (dwActions) occurs, the user-mode events handler callback function will be called.</p>

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [[A.11](#)].

REMARKS

- This function can be called from the **user mode only**.
- A successful call to this function must be followed with a call to `WDC_EventUnregister()` [[A.2.45](#)] later on in the code, in order to un-register from receiving Plug-and-play and power management notifications from the device.

A.2.45 WDC_EventUnregister()

PURPOSE

- Un-registers an application from a receiving Plug-and-Play and power management notifications for a device, pursuant to a previous call to WDC_EventRegister() [A.2.44].

PROTOTYPE

```
DWORD WINAPI WDC_EventUnregister(WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
> hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a Plug and Play WDC device, returned by WDC_PciDeviceOpen() [A.2.8] or WDC_PcmciaDeviceOpen() [A.2.9]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- This function can be called from the **user mode only**.

A.2.46 WDC_EventIsRegistered()

PURPOSE

- Checks if the application is currently registered to receive Plug-and-Play and power management notifications for the device.

PROTOTYPE

```
BOOL WINAPI WDC_EventIsRegistered (WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a Plug and Play WDC device, returned by WDC_PciDeviceOpen() [A.2.8] or WDC_PcmciaDeviceOpen() [A.2.9]

RETURN VALUE

Returns TRUE if the application is currently registered to receive Plug-and-Play and power management notifications for the device; otherwise returns FALSE.

A.2.47 WDC_SetDebugOptions()

PURPOSE

- Sets debug options for the WDC library – see the description of WDC_DBG_OPTIONS [A.2.1.8] for details regarding the possible debug options to set.
- This function is typically called at the beginning of the application, after the call to WDC_DriverOpen() [A.2.2], and can be re-called at any time while the WDC library is in use (i.e. WDC_DriverClose() [A.2.3] has not been called) in order to change the debug settings.
- Until the function is called, the WDC library uses the default debug options – see WDC_DBG_DEFAULT [A.2.1.8].

When the function is recalled, it performs any required cleanup for the previous debug settings and sets the default debug options before attempting to set the new options specified by the caller.

PROTOTYPE

```
DWORD WINAPI WDC_SetDebugOptions (WDC_DBG_OPTIONS dbgOptions ,
    const CHAR *sDbgFile );
```

PARAMETERS

Name	Type	Input/Output
➤ dbgOptions	WDC_DBG_OPTIONS	Input
➤ sDbgFile	const CHAR*	Input

DESCRIPTION

Name	Description
dbgOptions	A bit mask of flags indicating the desired debug settings – see WDC_DBG_OPTIONS [A.2.1.8]. If this parameter is set to zero (0), the default debug options will be used – see WDC_DBG_DEFAULT [A.2.1.8]

Name	Description
sDbgFile	WDC debug output file. This parameter is relevant only if the WDC_DBG_OUT_FILE flag is set in the debug options (dbgOptions) (either directly or via one of the convenience debug options combinations – see WDC_DBG_OPTIONS [A.2.1.8]). If the WDC_DBG_OUT_FILE debug flag is set and sDbgFile is NULL, WDC debug messages will be logged to the default debug file – stderr .

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.2.48 WDC_Err()

PURPOSE

- Displays debug error messages according to the WDC debug options – see WDC_DBG_OPTIONS [A.2.1.8] and WDC_SetDebugOptions() [A.2.47].

PROTOTYPE

```
void DLLCALLCONV WDC_Err( const CHAR *format[ , argument ] ... );
```

PARAMETERS

Name	Type	Input/Output
➤ format	const CHAR*	Input
➤ argument		Input

DESCRIPTION

Name	Description
format	Format-control string, which contains the error message to display. The string is limited to 256 characters (CHAR)
argument	Optional arguments for the format string

RETURN VALUE

None

A.2.49 WDC_Trace()

PURPOSE

- Displays debug trace messages according to the WDC debug options – see WDC_DBG_OPTIONS [A.2.1.8] and WDC_SetDebugOptions() [A.2.47].

```
void DLLCALLCONV WDC_Trace(const CHAR *format[, argument] ...);
```

PARAMETERS

Name	Type	Input/Output
➤ format	const CHAR*	Input
➤ argument		Input

DESCRIPTION

Name	Description
format	Format-control string, which contains the trace message to display. The string is limited to 256 characters (CHAR)
argument	Optional arguments for the format string

RETURN VALUE

None

A.2.50 WDC_GetWDHandle()

PURPOSE

- Returns a handle to WinDriver's kernel module, which is required by the basic WD_xxx WinDriver PCI/PCMCIA/ISA API [\[A.4\]](#) (see Remarks below).

PROTOTYPE

```
HANDLE DLLCALLCONV WDC_GetWDHandle( void );
```

RETURN VALUE

Returns a handle to WinDriver's kernel module, or INVALID_HANDLE_VALUE in case of a failure

REMARKS

- When using only the WDC API, you do not need to get a handle to WinDriver, since the WDC library encapsulates this for you. This function enables you to get the WinDriver handles used by the WDC library so you can pass it to lower-level WD_xxx API, if such APIs are used from your code. In such cases, take care **not** to close the handle you received (using WD_Close() [\[A.7.4\]](#)). The handle will be closed by the WDC library when it is closed, using WDC_DriverClose() [\[A.2.3\]](#).

A.2.51 WDC_GetDevContext()

PURPOSE

- Returns the device's user context information.

```
PVOID DLLCALLCONV WDC_GetDevContext(WDC_DEVICE_HANDLE hDev);
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])

RETURN VALUE

Returns a pointer to the device's user context, or NULL if not context has been set.

A.2.52 WDC_GetBusType()

PURPOSE

• Returns the device's bus type: WD_BUS_PCI, WD_BUS_PCMCIA, WD_BYIS_ISA or WD_BUS_UNKNOWN.

PROTOTYPE

```
WD_BUS_TYPE DLLCALLCONV WDC_GetBusType(WDC_DEVICE_HANDLE hDev) ;
```

PARAMETERS

Name	Type	Input/Output
➤ hDev	WDC_DEVICE_HANDLE	Input

DESCRIPTION

Name	Description
hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen() (PCI [A.2.8] / PCMCIA [A.2.9] / ISA [A.2.10])

RETURN VALUE

Returns the device's bus type.

A.2.53 WDC_Sleep()

PURPOSE

- Delays execution for the specified duration of time (in microseconds).
By default the function performs a busy sleep (consumes the CPU).

PROTOTYPE

```
DWORD WINAPI WDC_Sleep(DWORD dwMicroSecs , WDC_SLEEP_OPTIONS options);
```

PARAMETERS

Name	Type	Input/Output
➤ dwMicroSecs	DWORD	Input
➤ options	WDC_SLEEP_OPTIONS	Input

DESCRIPTION

Name	Description
dwMicroSecs	The number of microseconds to sleep
options	Sleep options [A.2.1.7]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.3 PCI/PCMCIA/ISA – WDC Low Level API

This section described the WDC types and preprocessor definitions defined in the **WinDriver/include/wdc_defs.h** header file

A.3.1 WDC_ID_U Union

WDC device ID information union type (used for PCI and PCMCIA devices):

Name	Type	Description
➤ pciId	WD_PCI_ID	PCI device ID information structure
❑ dwVendorId	DWORD	The device's vendor ID
❑ dwProductId	DWORD	The device's product ID
➤ pcmciaId	WD_PCMCIA_ID	PCMCIA device ID information structure
❑ wManufacturerId	WORD	The device's manufacturer ID
❑ wCardId	WORD	The device's device ID

A.3.2 WDC_ADDR_DESC

PCI/PCMCIA/ISA device memory or I/O address space information structure type:

Name	Type	Description
➤ flsMemory	BOOL	TRUE – memory address space ; FALSE – I/O address space
➤ dwAddrSpace	DWORD	The address space number
➤ dwItemIndex	DWORD	The index of the WD_ITEMS structure for the address space, which is retrieved and stored by WDC_xxxDeviceOpen() in the CardReg.Card.Item array in the relevant WDC device information structure [A.3.3]
➤ dwBytes	DWORD	The address space's size (in bytes)
➤ kptAddr	KPTR	The kernel-mode mapping of the address space's physical base address. This address is used by the WDC API for accessing a memory or I/O region using the WD_Transfer() [A.4.14] or WD_MultiTransfer() [A.4.15] APIs, or when accessing memory address directly in the kernel

Name	Type	Description
➤ dwUserDirectMemAddr	DWORD	The user-mode mapping of a memory address space's physical base address. This address is used for accessing memory addresses directly from the user mode

A.3.3 WDC_DEVICE

PCI/PCMCIA/ISA device information structure type.

The WDC_xxxDeviceOpen() functions (PCI: [A.2.8] / PCMCIA: [A.2.9] / ISA: [A.2.10]) allocate and return device structures of this type.

Name	Type	Description
➤ id	WDC_ID_U	Device ID information union (relevant for PCI and PCMCIA devices) – see [A.3.1]
➤ slot	WDC_SLOT_U	Device location information structure – see description of WDC_SLOT_U in section [A.2.1.9]
➤ dwNumAddrSpaces	DWORD	Number of address spaces found on the device
➤ pAddrDesc	WDC_ADDR_DESC*	Array of memory and I/O address spaces information structures – see [A.3.2]
➤ cardReg	WD_CARD_REGISTER	WinDriver device resources information structure, returned by WD_CardRegister() [A.4.11], which is called by the WDC_xxxDeviceOpen() functions
➤ kerPlug	WD_KERNEL_PLUGIN	Kernel PlugIn driver information structure [A.10.1]. This structure is filled by the WDC_xxxDeviceOpen() functions if the caller selects to use a Kernel PlugIn driver (otherwise this structure is not used) and is maintained by the WDC library.
➤ Int	WD_INTERRUPT	Interrupt information structure. This structure is filled by the WDC_xxxDeviceOpen() functions for devices that have interrupts, and is maintained by the WDC library.
☐ hInterrupt	DWORD	Handle to an internal WinDriver interrupt structure, required by the low-level WD_xxx() WinDriver interrupt APIs.

Name	Type	Description
<input type="checkbox"/> dwOptions	DWORD	A bit mask of interrupt information flags, which is passed by the WDC_xxxDeviceOpen() functions to the lower-level WD_CardRegister() [A.4.11] WinDriver function
➤ hIntThread	DWORD	Handle to the interrupt thread that is spawn when interrupts are enabled. This handle is passed by WDC to the low-level WinDriver interrupt APIs. When using the WDC API you do not need to access this handle directly.
➤ Event	WD_EVENT	WinDriver Plug-and-Play and power management events information structure – see [A.6.2] for details
hEvent	HANDLE	Handle used by the WinDriver EventRegister() [A.6.2] / EventUnregister() [A.6.3] functions. When using the WDC API you do not need to access this handle directly.
➤ pCtx	PVOID	Device context information. This information is received as a parameter by the WDC_xxxDeviceOpen() functions and stored in the device structure for future use by the calling application (optional)

A.3.4 PWDC_DEVICE

Pointer to a WDC_DEVICE structure [A.3.3] type:

```
typedef WDC_DEVICE* PWDC_DEVICE
```

A.3.5 WDC_MEM_DIRECT_ADDR Macro

PURPOSE

- Utility macro that returns a pointer that can be used for direct access to a specified memory address space from the context of the calling process.

PROTOTYPE**WDC_MEM_DIRECT_ADDR (pAddrDesc)****PARAMETERS**

Name	Type	Input/Output
➤ pAddrDesc	WDC_ADDR_DESC*	Input

DESCRIPTION

Name	Description
pAddrDesc	Pointer to a WDC memory address space information structure – see [A.3.2]

RETURN VALUE

When called from the user mode, returns the user-mode mapping of the physical memory address (pAddrDesc->dwUserDirectMemAddr);

When called from the kernel mode, returns the kernel-mode mapping of the physical memory address (pAddrDesc->kptAddr).

The returned pointer can be used for accessing the memory directly from the user mode or kernel mode, respectively.

A.3.6 WDC_ADDR_IS_MEM Macro

PURPOSE

- Utility macro that checks if a given address space is a memory or I/O address space.

PROTOTYPE

```
WDC_ADDR_IS_MEM( pAddrDesc )
```

PARAMETERS

Name	Type	Input/Output
➤ pAddrDesc	WDC_ADDR_DESC*	Input

DESCRIPTION

Name	Description
pAddrDesc	Pointer to a WDC memory address space information structure – see [A.3.2]

RETURN VALUE

Returns pAddrDesc->fIsMemory – TRUE for a memory address space; FALSE otherwise.

A.3.7 WDC_ADDR_SPACE_IS_ACTIVE Macro

PURPOSE

- Utility macro that checks if a given address space is a memory or I/O address space.

PROTOTYPE

```
WDC_ADDR_SPACE_IS_ACTIVE( pAddrDesc )
```

PARAMETERS

Name	Type	Input/Output
➤ pAddrDesc	WDC_ADDR_DESC*	Input

DESCRIPTION

Name	Description
pAddrDesc	Pointer to a WDC memory address space information structure – see [A.3.2]

RETURN VALUE

Returns TRUE if the specified address space is active – i.e. its size (pAddrDesc->dwBytes) is not zero (0); otherwise returns FALSE.

A.3.8 WDC_GET_ADDR_DESC Macro

PURPOSE

- Utility macro that retrieves a WDC address space information structure (WDC_ADDR_DESC [A.3.2]), which complies to the specified address space number.

PROTOTYPE

```
WDC_GET_ADDR_DESC(pDev , dwAddrSpace)
```

PARAMETERS

Name	Type	Input/Output
➤ pDev	PWDC_DEVICE	Input
➤ dwAddrSpace	DWORD	Input

DESCRIPTION

Name	Description
pDev	Pointer to a WDC device information structure – see [A.3.4]
dwAddrSpace	Address space number

RETURN VALUE

Returns a
pointer to the device's address information structure (WDC_ADDR_DESC [A.3.2]) for
the specified address space number – pDev->pAddrDesc[dwAddrSpace].

A.3.9 WDC_IS_KP Macro

PURPOSE

- Utility macro that checks if a WDC device uses a Kernel PlugIn driver.

PROTOTYPE

```
WDC_IS_KP( pDev )
```

PARAMETERS

Name	Type	Input/Output
➤ pDev	PWDC_DEVICE	Input

DESCRIPTION

Name	Description
pDev	Pointer to a WDC device information structure – see [A.3.4]

RETURN VALUE

Returns TRUE if the device uses a Kernel PlugIn driver; otherwise returns FALSE.

A.4 PCI/PCMCIA/ISA - WD_xxx Functions

This section describes the basic WD_xxx() PCI/PCMCIA/ISA WinDriver API.

NOTE

It is recommended to use the API from WinDriver's **WDC** library, which provides convenient wrappers to the basic WD_xxx PCI/PCMCIA/ISA API [\[A.1\]](#), instead of using the WD_xxx functions described in this section directly.

A.4.1 Calling Sequence WinDriver - PCI/PCMCIA/ISA

The following is a typical calling sequence for the PCI/PCMCIA/ISA drivers.

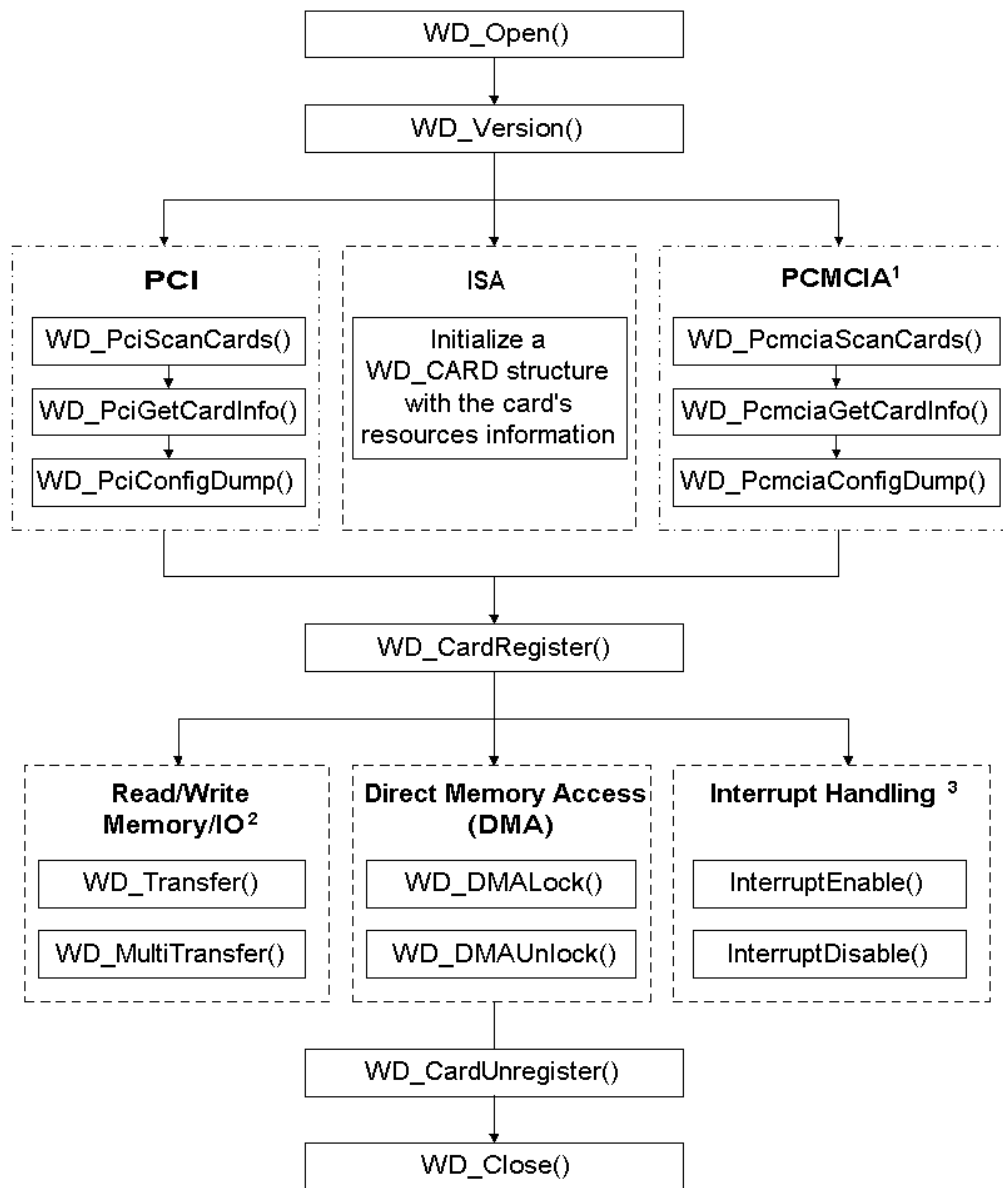


Figure A.1: WinDriver PCI/PCMCIA/ISA Calling Sequence

NOTES

1. PCMCIA is supported only on Windows 2000/XP/Server 2003.
2. Memory addresses can be accessed directly, using the user-mode mapping of the address returned by `WD_CardRegister()` [A.4.11] (or the kernel mapping, when accessing memory from the Kernel PlugIn [11]). Direct memory access is more efficient than using `WD_Transfer()`.
3. It is possible (although not recommended) to replace the use of the high-level `InterruptEnable()` convenience function [A.4.21] with the low-level `WD_IntEnable()` [A.5.2], `WD_IntWait()` [A.5.3] and `WD_IntCount()` [A.5.4] functions, and replace the call to `InterruptDisable()` [A.4.22] with a call to the low-level `WD_IntDisable()` function [A.5.5].
For more information on the low-level WinDriver interrupt handling API, refer to Section A.5 of the manual.
4. WinDriver's general-use APIs, such as `WD_DebugAdd()` [A.7.6] or `WD_Sleep()` [A.7.8], can be called anywhere between the calls to `WD_Open()` and `WD_Close()`. For more details, refer to Section A.7.

A.4.2 WD_PciScanCards()

PURPOSE

- Detects PCI devices installed on the PCI bus, which conform to the input criteria (VendorID and/or DeviceID), and returns the number and location (bus, slot and function) of the detected devices.

PROTOTYPE

```
DWORD WD_PciScanCards(HANDLE hWD, WD_PCI_SCAN_CARDS *pPciScan);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPciScan	WD_PCI_SCAN_CARDS *	
□ searchId	WD_PCI_ID	
◆ dwVendorId	DWORD	Input
◆ dwDeviceId	DWORD	Input
□ dwCards	DWORD	Output
□ cardId	Array of WD_PCI_ID	
◆ dwVendorId	DWORD	Output
◆ dwDeviceId	DWORD	Output
□ cardSlot	Array of WD_PCI_SLOT	
◆ dwBus	DWORD	Output
◆ dwSlot	DWORD	Output
◆ dwFunction	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pPciScan	WD_PCI_SCAN_CARDS elements:
searchId	WD_PCI_ID elements:
searchId.dwVendorId	Required PCI Vendor ID to detect. If 0, detects devices from all vendors.
searchId.dwDeviceId	Required PCI Device ID to detect. If 0, detects all devices.

Name	Description
dwCards	Number of devices detected.
cardId	WD_PCI_ID elements:
cardId.dwVendorId	Vendor IDs of the detected devices (corresponding to the required Vendor ID defined in searchId.dwVendorId).
cardId.dwDeviceId	Device IDs of the detected devices (corresponding to the required Device ID defined in searchId.dwDeviceId).
cardSlot	WD_PCI_SLOT elements:
cardSlot.dwBus	Bus number of detected device.
cardSlot.dwSlot	Slot number of detected device.
cardSlot.dwFunction	Function number of detected device.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_PCI_SCAN_CARDS pciScan;
DWORD cards_found;
WD_PCI_SLOT pciSlot;

BZERO(pciScan);
pciScan.searchId.dwVendorId = 0x12bc;
pciScan.searchId.dwDeviceId = 0x1;
WD_PciScanCards(hWD, &pciScan);
if (pciScan.dwCards>0) /* Found at least one device */
    pciSlot = pciScan.cardSlot[0]; /* use the first card found */
else
    printf("No matching PCI devices found\n");
```

A.4.3 WD_PciGetCardInfo()

PURPOSE

- Retrieves PCI device's resource information (i.e., Memory ranges, I/O ranges, Interrupt lines).

PROTOTYPE

```
DWORD WD_PciGetCardInfo (HANDLE hWD,  
    WD_PCI_CARD_INFO *pPciCard );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPciCard	WD_PCI_CARD_INFO *	
□ pciSlot	WD_PCI_SLOT	
◆ dwBus	DWORD	Input
◆ dwSlot	DWORD	Input
◆ dwFunction	DWORD	Input
□ Card	WD_CARD	
◆ dwItems	DWORD	Output
◆ Item	Array of WD_ITEMS	
◇ item	DWORD	Output
◇ fNotSharable	DWORD	Output
◇ I	union	
◆ Mem	struct	
→ dwPhysicalAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwTransAddr	DWORD	N/A
→ dwUserDirectAddr	DWORD	N/A
→ dwCpuPhysicalAddr	DWORD	N/A
→ dwBar	DWORD	Output
◆ IO	struct	
→ dwAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwBar	DWORD	Output
◆ Int	struct	

Name	Type	Input/Output
→ dwInterrupt	DWORD	Output
→ dwOptions	DWORD	N/A
→ hInterrupt	DWORD	N/A
♦ Bus	struct	
→ dwBusType	WD_BUS_TYPE	Output
→ dwBusNum	DWORD	Output
→ dwSlotFunc	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pPciCard	WD_PCI_CARD_INFO elements:
pciSlot	WD_PCI_SLOT elements:
pciSlot.dwBus	PCI bus number of device.
pciSlot.dwSlot	PCI slot number of device.
pciSlot.dwFunction	PCI function num of device.
Card	WD_CARD elements:
dwItems	Number of items detected on device.
Item	WD_ITEMS elements:
item	Type of item. Can be ITEM_MEMORY, ITEM_IO, ITEM_INTERRUPT or ITEM_BUS.
fNotSharable	If true, only one application at a time can access the mapped memory range, or monitor this card's interrupts.
I	Specific data according to "Item".
I.Mem	Describes ITEM_MEMORY.
I.Mem.dwPhysicalAddr	First address of physical memory range.
I.Mem.dwBytes	Length of range in bytes.
I.Mem.dwBar	Base Address Register number of PCI card.
I.IO	Describes ITEM_IO.
I.IO.dwAddr	First address of I/O range.
I.IO.dwBytes	Length of range in bytes.
I.IO.dwBar	Base Address Register number of PCI card.
I.Int	Describes ITEM_INTERRUPT.
I.Int.dwInterrupt	Physical number of interrupt request (IRQ).
I.Bus	Describes ITEM_BUS.
I.Bus.dwBusType	Used to save type of device from the WD_BUS_TYPE options (i.e., ISA/ISAPnP/PCI/PCMCIA) and in this case - WD_BUS_PCI.

Name	Description
I.Bus.dwBusNum	Bus number of the specific PCI device.
I.Bus.dwSlotFunc	Slot and Function. This value is a combination of the slot number and the function number: The lower three bits represent the function number and the remaining bits represent the slot number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot number of 0x10 (remaining bits: 10000).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- For PCI devices, if the I/O, memory and IRQ information is available from the Plug-and-Play manager, the information is obtained from there. Otherwise, the information is read directly from the PCI configuration registers. Note: for Windows, you must have an **inf** file installed.
- If the IRQ is obtained from the Plug-and-Play manager, it is mapped and therefore may differ from the physical IRQ.

EXAMPLE

```
WD_PCI_CARD_INFO pciCardInfo;
WD_CARD Card;

BZERO(pciCardInfo);
pciCardInfo.pciSlot = pciSlot;
WD_PciGetCardInfo(hWD, &pciCardInfo);
if (pciCardInfo.Card.dwItems!=0) /* At least one item was found */
{
    Card = pciCardInfo.Card;
}
else
{
    printf("Failed fetching PCI card information\n");
}
```

A.4.4 WD_PciConfigDump()

PURPOSE

- Reads/Writes from/to the PCI configuration space of a selected PCI device or the extended configuration space of a selected PCI Express device (on Windows/Linux).

For Windows and Linux, all references to "PCI" in the description below also include PCI Express.

PROTOTYPE

```
DWORD WD_PciConfigDump(HANDLE hWD, WD_PCI_CONFIG_DUMP *pConfig);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pConfig	WD_PCI_CONFIG_DUMP *	
□ pciSlot	WD_PCI_SLOT	
◆ dwBus	DWORD	Input
◆ dwSlot	DWORD	Input
◆ dwFunction	DWORD	Input
□ pBuffer	PVOID	Input/Output
□ dwOffset	DWORD	Input
□ dwBytes	DWORD	Input
□ flsRead	DWORD	Input
□ dwResult	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pConfig	WD_PCI_CONFIG_DUMP elements:
pciSlot	WD_PCI_SLOT elements:
pciSlot.dwBus	PCI bus number of card.
pciSlot.dwSlot	PCI slot number of card.
pciSlot.dwFunction	PCI function number of card.

Name	Description
pBuffer	A pointer to the data that will either: 1. Be written to the PCI configuration registers. 2. Be read from the PCI configuration registers.
dwOffset	The offset of the specific register/s in PCI configuration space to read/write from/to.
dwBytes	Number of bytes read/written from/to buffer.
fIsRead	If TRUE - read from PCI configuration registers. If FALSE - write to PCI configuration registers.
dwResult	1. PCI_ACCESS_OK - read/write ok. 2. PCI_ACCESS_ERROR - failed reading/writing. 3. PCI_BAD_BUS - bus does not exist. 4. PCI_BAD_SLOT - slot or Function does not exist.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

EXAMPLE

```
WD_PCI_CONFIG_DUMP pciConfig;
DWORD dwStatus;
WORD aBuffer[2];

BZERO(pciConfig);
pciConfig.pciSlot.dwBus = 0;
pciConfig.pciSlot.dwSlot = 3;
pciConfig.pciSlot.dwFunction = 0;
pciConfig.pBuffer = aBuffer;
pciConfig.dwOffset = 0;
pciConfig.dwBytes = sizeof(aBuffer);
pciConfig.fIsRead = TRUE;

dwStatus = WD_PciConfigDump(hWD, &pciConfig);
if (dwStatus)
{
    printf("WD_PciConfigDump failed: %s\n", Stat2Str(dwStatus));
}
else
{
    printf("Card in Bus 0, Slot 3, Function 0 has Vendor ID %x "
           "Device ID %x\n", aBuffer[0], aBuffer[1]);
}
```

A.4.5 WD_PcmciaScanCards()

PURPOSE

- Detects PCMCIA devices attached to PCMCIA sockets, which conform to the input criteria (ManufacturerId and/or CardId), and returns the number and location (bus, socket and function) of the detected devices.

PROTOTYPE

```
DWORD WD_PcmciaScanCards(HANDLE hWD,
                          WD_PCMCIA_SCAN_CARDS *pPcmciaScan);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPcmciaScan	WD_PCMCIA_SCAN_CARDS *	
□ searchId	WD_PCMCIA_ID	
◆ wManufacturerId	WORD	Input
◆ wCardId	WORD	Input
□ dwCards	DWORD	Output
□ cardId	Array of WD_PCMCIA_ID	
◆ wManufacturerId	DWORD	Output
◆ wCardId	DWORD	Output
□ cardSlot	Array of WD_PCMCIA_SLOT	
◆ uBus	BYTE	Output
◆ uSocket	BYTE	Output
◆ uFunction	BYTE	Output
◆ uPadding	BYTE	N/A
□ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pPcmciaScan	WD_PCMCIA_SCAN_CARDS elements:
searchId	WD_PCMCIA_ID elements:

Name	Description
searchId.wManufacturerId	Required PCMCIA card manufacturer ID to detect. If 0, detects devices from all manufacturers.
searchId.wCardId	Required PCMCIA card ID to detect. If 0, detects all cards.
dwCards	Number of cards detected.
cardId	WD_PCMCIA_ID elements:
cardId.wManufacturerId	Manufacturer IDs of the detected devices (corresponding to the required Manufacturer ID defined in searchId.dwManufacturerId).
cardId.wCardId	Card IDs of the detected devices (corresponding to the required Card ID defined in searchId.wCardId).
cardSlot	WD_PCMCIA_SLOT elements:
cardSlot.uBus	Bus number of detected device (first bus is 0).
cardSlot.uSocket	Socket number of detected device (first socket is 0).
cardSlot.uFunction	Function number of detected device (first function is 0).
cardSlot.uPadding	Padding of 1 byte (reserved).
dwOptions	Reserved for future use, set to 0.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_PCMCIA_SCAN_CARDS pcmciaScan;
DWORD cards_found;
WD_PCMCIA_SLOT pcmciaSlot;

BZERO(pcmciaScan);
pcmciaScan.searchId.wManufacturerId = 0x1234;
pcmciaScan.searchId.wCardId = 0x5678;
WD_PcmciaScanCards(hWD, &pcmciaScan);
if (pcmciaScan.dwCards>0) /* Found at least one device */
{
    /* use the first card found */
    pcmciaSlot = pcmciaScan.cardSlot[0];
}
else
{
    printf("No matching PCMCIA devices found\n");
}
```

A.4.6 WD_PcmciaGetCardInfo()

PURPOSE

- Retrieves PCMCIA device's resource information (i.e. memory ranges, version, manufacturer and model strings, type of device, interrupt information).

PROTOTYPE

```
DWORD WD_PcmciaGetCardInfo(HANDLE hWD, WD_PCMCIA_CARD_INFO *pPcmciaCard);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPcmciaCard	WD_PCMCIA_CARD_INFO *	
□ pcmciaSlot	WD_PCMCIA_SLOT	
◆ uBus	BYTE	Input
◆ uSocket	BYTE	Input
◆ uFunction	BYTE	Input
◆ uPadding	BYTE	N/A
□ Card	WD_CARD	
◆ dwItems	DWORD	Output
◆ Item	Array of WD_ITEMS	
◇ item	DWORD	Output
◇ fNotSharable	DWORD	Output
◇ I	union	
◆ Mem	struct	
→ dwPhysicalAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwTransAddr	DWORD	N/A
→ dwUserDirectAddr	DWORD	N/A
→ dwCpuPhysicalAddr	DWORD	N/A
→ dwBar	DWORD	Output
◆ IO	struct	
→ dwAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwBar	DWORD	Output
◆ Int	struct	

Name	Type	Input/Output
→ dwInterrupt	DWORD	Output
→ dwOptions	DWORD	N/A
→ hInterrupt	DWORD	N/A
♦ Bus	struct	
→ dwBusType	WD_BUS_TYPE	Output
→ dwBusNum	DWORD	Output
→ dwSlotFunc	DWORD	Output
□ cVersion[4]	CHAR	Output
□ cManufacturer[48]	CHAR	Output
□ cProductName[48]	CHAR	Output
□ wManufacturerId	WORD	Input
□ wCardId	WORD	Input
□ wFuncId	WORD	Input
□ dwOptions	DWORD	N/A

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pPcmciaCard	WD_PCMCIA_CARD_INFO elements:
PcmciaSlot	WD_PCMCIA_SLOT elements:
cardSlot.uBus	Bus number of device (first bus is 0).
cardSlot.uSocket	Socket number of device (first socket is 0).
cardSlot.uFunction	Function number of device (first function is 0).
cardSlot.uPadding	Padding of 1 byte (reserved).
Card	WD_CARD elements:
dwItems	Number of items detected on device.
Item	WD_ITEMS elements:
item	Type of item. Can be ITEM_MEMORY, ITEM_IO, ITEM_INTERRUPT or ITEM_BUS.
fNotSharable	If true, only one application at a time can access the mapped memory range, or monitor this card's interrupts.
I	Specific data according to "Item".
I.Mem	Describes ITEM_MEMORY.
I.Mem.dwPhysicalAddr	First address of physical memory range.
I.Mem.dwBytes	Length of range in bytes.
I.Mem.dwBar	Base Address Register number of PCMCIA card.
I.IO	Describes ITEM_IO.
I.IO.dwAddr	First address of I/O range.

Name	Description
I.IO.dwBytes	Length of range in bytes.
I.IO.dwBar	Base Address Register number of PCMCIA card.
I.Int	Describes ITEM_INTERRUPT.
I.Int.dwInterrupt	Physical number of interrupt request (IRQ).
I.Bus	Describes ITEM_BUS.
I.Bus.dwBusType	Used to save type of device from the WD_BUS_TYPE options (i.e., ISA/ISAPnP/PCI/PCMCIA) and in this case - WD_BUS_PCMCIA.
I.Bus.dwBusNum	Bus number of the specific PCMCIA device.
I.Bus.dwSlotFunc	Socket and Function. This value is a combination of the socket number and the function number: The lower three bits represent the function number and the remaining bits represent the socket number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a socket number of 0x10 (remaining bits: 10000).
cVersion[4]	Version string
cManufacturer[48]	Manufacturer string
cProductName[48]	Product name string
wManufacturerId	Card manufacturer ID.
wCardId	Card type and model ID.
wFuncId	Card function ID.
dwOptions	Reserved for future use, set to 0.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- Resource information for PCMCIA cards is obtained by WinDriver from the Windows Plug-and-Play manager (On Windows 2000 and above).

EXAMPLE

```
WD_PCMCIA_CARD_INFO pcmciaCardInfo;
WD_CARD Card;

BZERO(pcmciaCardInfo);
pcmciaCardInfo.pcmciaSlot = pcmciaSlot;
WD_PcmciaGetCardInfo(hWD, &pcmciaCardInfo);
```

```
if (pcmciaCardInfo.Card.dwItems!=0) /* At least one item was found */
{
    Card = pcmciaCardInfo.Card;
}
else
{
    printf("Failed fetching PCMCIA card information\n");
}
```


A.4.7 WD_PcmciaConfigDump()

PURPOSE

• Reads/Writes from/to the PC Card attribute space of a selected PCMCIA device, this is where the Card Information Structure (CIS) is stored.

PROTOTYPE

```
DWORD WD_PcmciaConfigDump(HANDLE hWD, WD_PCMCIA_CONFIG_DUMP *pPcmciaConfig);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPcmciaConfig	WD_PCMCIA_CONFIG_DUMP *	
□ pcmciaSlot	WD_PCMCIA_SLOT	
◆ uBus	BYTE	Input
◆ uSocket	BYTE	Input
◆ uFunction	BYTE	Input
◆ uPadding	BYTE	N/A
□ pBuffer	PVOID	Input/Output
□ dwOffset	DWORD	Input
□ dwBytes	DWORD	Input
□ flsRead	DWORD	Input
□ dwResult	DWORD	Output
□ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pPcmciaConfig	WD_PCMCIA_CONFIG_DUMP elements:
pcmciaSlot	WD_PCMCIA_SLOT elements:
cardSlot.uBus	Bus number of device (first bus is 0).
cardSlot.uSocket	Socket number of device (first socket is 0).
cardSlot.uFunction	Function number of device (first function is 0).

Name	Description
cardSlot.uPadding	Padding of 1 byte (reserved).
pBuffer	A pointer to the data that will either: 1. Be written to the PCMCIA configuration registers. 2. Be read from the PCMCIA configuration registers.
dwOffset	The offset of the specific register/s in PCMCIA configuration space to read/write from/to.
dwBytes	Number of bytes read/written from/to buffer.
fIsRead	If TRUE - read from PCMCIA configuration registers. If FALSE - write to PCMCIA configuration registers.
dwResult	1. PCMCIA_ACCESS_OK - read/write ok. 2. PCMCIA_BAD_SOCKET - socket does not exist. 3. PCMCIA_BAD_OFFSET - incorrect offset. 4. PCMCIA_ACCESS_ERROR - failed reading/writing.
dwOptions	Reserved for future use, set to 0.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_PCMCIA_CONFIG_DUMP pcmciaConfig;
DWORD dwStatus;
WORD aBuffer[2];

BZERO(pcmciaConfig);
pcmciaConfig.pcmciaSlot.uBus = 0;
pcmciaConfig.pcmciaSlot.uSocket = 0;
pcmciaConfig.pcmciaSlot.uFunction = 0;
pcmciaConfig.pcmciaSlot.uPadding = 0;
pcmciaConfig.pBuffer = aBuffer;
pcmciaConfig.dwOffset = 0;
pcmciaConfig.dwBytes = sizeof(aBuffer);
pcmciaConfig.fIsRead = TRUE;

dwStatus = WD_PcmciaConfigDump(hWD, &pcmciaConfig);
if (dwStatus)
{
    printf("WD_PcmciaConfigDump failed: %s\n", Stat2Str(dwStatus));
}
else
{

```

```
    printf("Card in Bus 0, Socket 0: the code of the first tuple in"
           " the CIS is %x\n", (UINT32)aBuffer[0]);
}
```

A.4.8 WD_IsapnpScanCards()

PURPOSE

- Detects ISA PnP devices installed on the ISA PnP bus that conform to the input criteria (VendorID and/or Serial Device Number), and returns the number and location (bus, slot and function) of the detected devices.

PROTOTYPE

```
DWORD WD_IsapnpScanCards(HANDLE hWD,
    WD_ISAPNP_SCAN_CARDS *pIsapnpScan );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pIsapnpScan	WD_ISAPNP_SCAN_CARDS *	
□ searchId	WD_ISAPNP_CARD_ID	
◆ cVendor[8]	CHAR	Input
◆ dwSerial	DWORD	Input
□ dwCards	DWORD	Output
□ Card	Array of WD_ISAPNP_CARD	
◆ cardId	WD_ISAPNP_CARD_ID	
◇ cVendor[8]	CHAR	Output
◇ dwSerial	DWORD	Output
◆ dwLogicalDevices	DWORD	Output
◆ bPnPVersionMajor	BYTE	Output
◆ bPnPVersionMinor	BYTE	Output
◆ bVendorVersionMajor	BYTE	Output
◆ bVendorVersionMinor	BYTE	Output
◆ cIdent	CHAR [36]	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pIsapnpScan	WD_ISAPNP_SCAN_CARDS elements:

Name	Description
searchId	WD_ISAPNP_CARD_ID elements:
searchId.cVendor[8]	Required ISA PnP Vendor ID to detect. If 0, detects devices from all vendors.
searchId.dwSerial	Required ISA PnP serial device number to detect. If 0, detects all devices.
dwCards	Number of devices detected.
Card	WD_ISAPNP_CARD elements.
cardId	WD_ISAPNP_CARD_ID elements - vendor ID and serial number of device found.
cardId.cVendor[8]	Vendor ID.
cardId.dwSerial	Serial number of device.
dwLogicalDevices	Number of logical devices on device.
bPnPVersionMajor	ISA PnP version major.
bPnPVersionMinor	ISA PnP version minor.
bVendorVersionMajor	Vendor version major.
bVendorVersionMinor	Vendor version minor.
cIdent	WD_ISAPNP_ANSI - the ASCII device identification string.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_ISAPNP_SCAN_CARDS isapnpScan;
DWORD Cards_found
WD_ISAPNP_CARD isapnpCard;

BZERO(isapnpScan);
/* CTL009e - Sound Blaster ISA PnP Card */
strcpy(isapnpScan.searchId.cVendorId, "CTL009e");
isapnpScan.searchId.dwSerial = 0;
WD_IsapnpScanCards(hWD, &isapnpScan);
if (isapnpScan.dwCards>0) /* Found at least one device */
{
    /* Take the first card found */
    isapnpCard = isapnpScan.Card[0];
}
else
```

```
{  
    printf("No matching ISA PnP devices found\n");  
}
```

A.4.9 WD_IsapnpGetCardInfo()

PURPOSE

- Retrieves ISA PnP device resources information (i.e., Memory ranges, IO ranges, Interrupts).

PROTOTYPE

```
DWORD WD_IsapnpGetCardInfo(HANDLE hWD,  
    WD_ISAPNP_CARD_INFO *pIsapnpCard );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pIsapnpCard	WD_ISAPNP_CARD_INFO *	
❑ cardId	WD_ISAPNP_CARD_ID	
◆ cVendor	CHAR[8]	Input
◆ dwSerial	DWORD	Input
❑ dwLogicalDevice	DWORD	Input
❑ clogicalDevice	CHAR [8]	Output
❑ dwCompatibleDevices	DWORD	Output
❑ CompatibleDevices	CHAR [10][8]	Output
❑ cIdent	CHAR [36]	Output
❑ Card	WD_CARD	
◆ dwItems	DWORD	Output
◆ Item	Array of WD_ITEMS	
◇ item	DWORD	Output
◇ fNotSharable	DWORD	Output
◇ I	union	
◆ Mem	struct	
→ dwPhysicalAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwTransAddr	DWORD	N/A
→ dwUserDirectAddr	DWORD	N/A
→ dwCpuPhysicalAddr	DWORD	N/A
→ dwBar	DWORD	Output
◆ IO	struct	

Name	Type	Input/Output
→ dwAddr	DWORD	Output
→ dwBytes	DWORD	Output
→ dwBar	DWORD	Output
♦ Int	struct	
→ dwInterrupt	DWORD	Output
→ hInterrupt	DWORD	N/A
→ dwOptions	DWORD	N/A
♦ Bus	struct	
→ dwBusType	WD_BUS_TYPE	Output
→ dwBusNum	DWORD	Output
→ dwSlotFunc	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pIsapnpCard	WD_ISAPNP_CARD_INFO elements:
cardId	WD_ISAPNP_CARD_ID elements:
cardId.cVendor	Required ISA Plug-and-Play Vendor ID for which information is required.
cardId.dwSerial	Required ISA Plug-and-Play serial device number for which information is required.
dwLogicalDevice	Number of the logical device for which information is required.
cLogicalDevice	WD_ISAPNP_COMP_ID - a string of 8 characters for the ASCII code of the logical device ID found.
dwCompatibleDevices	Number of compatible devices found.
CompatibleDevices	WD_ISAPNP_COMP_ID - an array of the compatible devices' IDs.
cIdent	WD_ISAPNP_ANSI - the ASCII device identification string.
Card	WD_CARD elements:
dwItems	Number of items detected on device.
Item	WD_ITEMS elements:
item	Type of item. Can be ITEM_MEMORY, ITEM_IO, ITEM_INTERRUPT or ITEM_BUS.
fNotSharable	If true, only one application at a time can access the mapped memory range or monitor this card's interrupts.
I	Specific data according to "Item".

Name	Description
I.Mem	Describes ITEM_MEMORY.
I.Mem.dwPhysicalAddr	First address of physical memory range.
I.Mem.dwBytes	Length of range in bytes.
I.Mem.dwBar	Base Address Register number of PCI card.
I.IO	Describes ITEM_IO.
I.IO.dwAddr	First address of I/O range.
I.IO.dwBytes	Length of range in bytes.
I.IO.dwBar	Base Address Register number of PCI card.
I.Int	Describes ITEM_INTERRUPT.
I.Int.dwInterrupt	Physical number of interrupt request (IRQ).
I.Bus	Describes ITEM_BUS.
I.Bus.dwBusType	Used to save type of device from the WD_BUS_TYPE options (i.e., ISA/ISAPnP/PCI/PCMCIA) and in this case - WD_BUS_EISA.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_ISAPNP_CARD_INFO isapnpCardInfo;
WD_CARD Card;

BZERO(isapnpCardInfo);
/* from WD_IsapnpScanCard(): */
isapnpCardInfo.CardId = isapnpCard;
isapnpCardInfo.dwLogicalDevice = 0;

WD_IsapnpGetCardInfo(hWD, &isapnpCardInfo);
/* At least one item was found */
if (isapnpCardInfo.Card.dwItems!=0)
    Card = isapnpCardInfo.Card;
else
    printf("Failed fetching ISA PnP card information\n");
```

A.4.10 WD_IsapnpConfigDump()

PURPOSE

- Reads/Writes from/to the ISA PnP configuration registers of a selected ISA PnP device.

PROTOTYPE

```
DWORD WD_IsapnpConfigDump(HANDLE hWD, WD_ISAPNP_CONFIG_DUMP *pConfig);
```

PARAMETERS

Name	Type	Input/Output
➤ hWD	HANDLE	Input
➤ pConfig	WD_ISAPNP_CONFIG_DUMP *	
❑ cardId	WD_ISAPNP_CARD_ID	
◆ cVendor	CHAR[8]	Input
◆ dwSerial	DWORD	Input
❑ dwOffset	DWORD	Input
❑ flsRead	DWORD	Input
❑ bData	BYTE	Input/Output
❑ dwResult	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pConfig	WD_ISAPNP_CONFIG_DUMP elements.
cardId	WD_ISAPNP_CARD_ID elements:
cardId.cVendor	Required ISA Plug-and-Play Vendor ID for the required device.
cardId.dwSerial	Required ISA Plug-and-Play serial device number for the required device.
dwOffset	The offset of the specific register/s in ISA PnP configuration space to read/write from/to.
flsRead	If TRUE - read from ISA PnP configuration registers. If FALSE - write to ISA PnP configuration registers.

Name	Description
bData	The data that will either: 1. Be written to the ISA PnP configuration registers 2. Be read from the ISA PnP configuration registers.
dwResult	0 - ISAPNP_ACCESS_OK - read/write ok. 1 - ISAPNP_ACCESS_ERROR - failed reading/writing. 2 - ISAPNP_BAD_ID - device does not exist.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_ISAPNP_CONFIG_DUMP isapnpConfig;

BZERO(isapnpConfig);
/* from WD_IsapnpScanCard(): */
isapnpConfig.CardId = isapnpCard;
isapnpConfig.dwOffset = 0;
isapnpConfig.fIsRead = TRUE;
WD_IsapnpConfigDump(hWD, &isapnpConfig);
if (isapnpConfig.dwResult!=ISAPNP_ACCESS_OK)
{
    printf("No ISA PnP device specified slot\n");
}
else
{
    printf("ISA PnP config in offset 0 =\%x\n",
        isapnpConfig.bData);
}
```

A.4.11 WD_CardRegister()

PURPOSE

- Maps the physical memory ranges to be accessed by kernel-mode processes and user-mode applications.
- Checks whether an I/O or Memory resource was previously exclusively registered.
- Saves data regarding interrupt request (IRQ) number and interrupt type (edge triggered or level sensitive) in internal data structures to be used by InterruptEnable() [A.4.21] or WD_IntEnable() [A.5.2].

PROTOTYPE

```
DWORD WD_CardRegister(HANDLE hWD, WD_CARD_REGISTER *pCardReg);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pCardReg	WD_CARD_REGISTER *	
□ Card	WD_CARD	
◆ dwItems	DWORD	Input
◆ Item	Array of WD_ITEMS	
◇ item	DWORD	Input
◇ fNotSharable	DWORD	Input
◇ dwOptions	DWORD	Input
◇ I	union	
◆ Mem	struct	
→ dwPhysicalAddr	DWORD	Input
→ dwBytes	DWORD	Input
→ dwTransAddr	DWORD	Output
→ dwUserDirectAddr	DWORD	Output
→ dwCpuPhysicalAddr	DWORD	Output
→ dwBar	DWORD	Input
◆ IO	struct	
→ dwAddr	DWORD	Input
→ dwBytes	DWORD	Input
→ dwBar	DWORD	Input

Name	Type	Input/Output
♦ Int	struct	
→ dwInterrupt	DWORD	Input
→ dwOptions	DWORD	Input
→ hInterrupt	DWORD	Output
♦ Bus	WD_BUS	
→ dwBusType	WD_BUS_TYPE	Input
→ dwBusNum	DWORD	Input
→ dwSlotFunc	DWORD	Input
❑ fCheckLockOnly	DWORD	Input
❑ hCard	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pCardReg	WD_CARD_REGISTER elements:
Card	WD_CARD elements:
dwItems	Number of items detected on device.
Item	WD_ITEMS elements:
item	Can be ITEM_MEMORY, ITEM_IO, ITEM_INTERRUPT or ITEM_BUS.
fNotSharable	If true, only one application at a time can access the mapped memory range, or monitor this card's interrupts.
dwOptions	Any of the following WD_ITEM_OPTIONS flags: <ul style="list-style-type: none"> • WD_ITEM_DO_NOT_MAP_KERNEL: This flag instructs the function to avoid mapping a memory address range to the kernel virtual address space and map the memory only to the user-mode virtual address space. See the Remarks to this function for more information. NOTE: This flag is applicable only to memory items. • WD_ITEM_ALLOW_CACHE (Windows NT/2k/XP/ Server 2003 and CE only): Map the item's physical memory (I.Mem.dwPhysicalAddr) as cached. NOTE: This flag is applicable only to memory items that pertain to the host's RAM, as opposed to local memory on the card.
I	Specific data according to "item".
I.Mem	Describes ITEM_MEMORY.
I.Mem.dwPhysicalAddr	First address of physical memory range.

Name	Description
I.Mem.dwBytes	Length of range in bytes.
I.Mem.dwTransAddr	Maps the physical memory address received by dwPhysicalAddr and dwBytes (in WD_XxxGetCardInfo) for kernel-mode processes. Used by WD_Transfer() [A.4.14].
I.Mem.dwUserDirectAddr	Maps the physical memory address received by dwPhysicalAddr and dwBytes (in WD_XxxGetCardInfo) for user-mode applications (enabling direct access from user mode).
I.Mem.dwCpuPhysicalAddr	Translates device's memory address from bus specific values into CPU values.
I.Mem.dwBar	Base Address Register number of PCI card.
I.IO	Describes ITEM_IO.
I.IO.dwAddr	First address of I/O range.
I.IO.dwBytes	Length of range in bytes.
I.IO.dwBar	Base Address Register number of PCI card.
I.Int	Describes ITEM_INTERRUPT.
I.Int.dwInterrupt	Physical number of interrupt request (IRQ).
I.Int.dwOptions	A bit mask flag: <ul style="list-style-type: none"> • INTERRUPT_LEVEL_SENSITIVE - If set, the interrupt is Level Sensitive. Default - Interrupt is Edge-Triggered (Received from WD_XxxGetCardInfo). <ul style="list-style-type: none"> • INTERRUPT_CE_INT_ID - On Windows CE (unlike other operating systems), there is an abstraction of the physical interrupt number to a logical one. Setting this bit will instruct WinDriver to refer to the interrupt in dwInterrupt as a logical interrupt number and convert it to a physical interrupt number.
I.Int.hInterrupt	Returns an interrupt handle to use with InterruptEnable() [A.4.21] or WD_IntEnable() [A.5.2].
I.Bus	Describes ITEM_BUS.
I.Bus.dwBusType	Used to save type of device from the WD_BUS_TYPE options (i.e., ISA/ISAPnP/PCI/PCMCIA) 1 = ISA; 2 = EISA; 5 = PCI; 8 = PCMCIA.
I.Bus.dwBusNum	Bus number of the specific device.
I.Bus.dwSlotFunc	Slot and Function.

Name	Description
fCheckLockOnly	When set to TRUE - checks whether certain resources were already locked when asking for an exclusive resource.
hCard	Handle to card used by WD_CardUnregister() [A.4.13]. 0 when card registration fails.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- For PCI/PCMCIA the cardReg.Card input resources information should be retrieved from the Plug-and-Play manager via WD_PciGetCardInfo() [A.4.3] and WD_PcmciaGetCardInfo() [A.4.6].
- If your card has a large memory range, which cannot be mapped entirely to the kernel virtual address space, you can set the WD_ITEM_DO_NOT_MAP_KERNEL flag for the relevant memory WD_ITEMS structure (pCardReg->Card.Item[i].dwOptions) in order to instruct the function to map this memory range only to the user-mode virtual address space but not the kernel address space.
(For PCI/PCMCIA devices, you can modify the relevant item in the card information structure (pCard) that you received from WD_PciGetCardInfo() [A.4.3] / WD_PcmciaGetCardInfo() [A.4.6] before passing this structure to WD_CardRegister()).

NOTE that if you select to set this flag, WD_CardRegister() will not update the item's dwTransAddr field with a kernel-mapping of the memory and you will therefore not be able to call WD_Transfer() [A.4.14] or WD_MultiTransfer() [A.4.15] in order to access the memory from the kernel, nor will you be able to access the memory directly from a Kernel PlugIn driver using the memory item's dwTransAddr field.

- WD_CardRegister() enables the user to map the card memory resources into virtual memory and access them as regular pointers. In **Windows CE** there is a security mechanism that prevents processes from accessing each other's memory space, without explicitly requesting it. This request is made using the SetProcPermissions() function, which sets the desired permissions for the current thread. Since the card resources are mapped inside WinDriver and these mappings belong to the Device Manager process, any other thread should call this function before accessing these mappings. Every call to WD_Open() calls SetProcPermissions() (see **windrvr.h**), so any thread that accesses

mapped regions and does not call `WD_Open()`, such as a user thread, Windows message thread etc, should explicitly call `SetProcPermissions()`. For more information, please refer to Microsoft's documentation.

WinDriver's `ThreadStart()` function [A.12.3] performs the required call to `SetProcPermissions()`, therefore when using `ThreadStart()` to create new threads you do not need to call `SetProcPermissions()` yourself.

EXAMPLE

```
WD_CARD_REGISTER cardReg;
BZERO(cardReg);
cardReg.Card.dwItems = 1;
cardReg.Card.Item[0].item = ITEM_IO;
cardReg.Card.Item[0].fNotSharable = TRUE;
cardReg.Card.Item[0].I.IO.dwAddr = 0x378;
cardReg.Card.Item[0].I.IO.dwBytes = 8;
WD_CardRegister(hWD, &cardReg);
if (cardReg.hCard==0)
{
    printf("Failed locking device\n");
    return FALSE;
}
```


A.4.12 WD_CardCleanupSetup()

PURPOSE

- Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:
 - The application exists abnormally.
 - The application exits normally but without un-registering the specified card.
 - If the WD_FORCE_CLEANUP flag is set in the dwOptions parameter, the cleanup commands will also be performed when the specified card is un-registered.

PROTOTYPE

```
DWORD WD_CardCleanupSetup (HANDLE hWD, WD_CARD_CLEANUP *pCardCleanup )
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pCardCleanup	WD_CARD_CLEANUP *	
□ hCard	DWORD	Input
□ Cmds	WD_TRANSFER *	Input
□ dwCmds	DWORD	Input
□ dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pCardCleanup	Pointer to the WD_CARD_CLEANUP structure.
hCard	Handle to the relevant card as received from WD_CardRegister() [A.4.11].
Cmds	Pointer to an array of cleanup transfer commands to be performed
dwCmds	Number of cleanup commands in the Cmds array

Name	Description
bForceCleanup	<p>If 0: The cleanup transfer commands (Cmd) will be performed in either of the following cases:</p> <ul style="list-style-type: none">• When the application exist abnormally.• When the application exits normally but without calling <code>WD_CardUnregister()</code> [A.4.13] to un-register the card. <p>If the WD_FORCE_CLEANUP flag is set: The cleanup transfer commands will be performed both in the two cases described above, as well as in the following case:</p> <ul style="list-style-type: none">• When <code>WD_CardUnregister()</code> [A.4.13] is called to un-register the card.

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

You should call this function right after calling `WD_CardRegister()` [A.4.11].

EXAMPLE

```
WD_CARD_CLEANUP cleanup;
BZERO(cleanup);

/* Set-up the cleanup struct with the cleanup information */

dwStatus = WD_CardCleanupSetup(hWD, &cleanup);
if (dwStatus)
{
    printf("WD_CardCleanupSetup failed: %s\n", Stat2Str(dwStatus));
}
```

A.4.13 WD_CardUnregister()

PURPOSE

- Unregisters a device and frees the resources allocated to it.

PROTOTYPE

```
DWORD WD_CardUnregister(HANDLE hWD, WD_CARD_REGISTER *pCardReg);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pCardReg	WD_CARD_REGISTER *	
□ Card	WD_CARD	N/A
□ fCheckLockOnly	DWORD	N/A
□ hCard	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
hCard	Handle of device to unregister received from WD_CardRegister() [A.4.11].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_CardUnregister(hWD, &cardReg);
```

A.4.14 WD_Transfer()

PURPOSE

- Executes a single read/write instruction to an I/O port or to a memory address.

PROTOTYPE

```
DWORD WD_Transfer(HANDLE hWD, WD_TRANSFER *pTrans );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pTrans	WD_TRANSFER *	
❑ cmdTrans	DWORD	Input
❑ dwPort	DWORD	Input
❑ dwBytes	DWORD	Input
❑ fAutoinc	DWORD	Input
❑ dwOptions	DWORD	Input
❑ Data	union	
❑ Data.Byte	BYTE	Input/Output
❑ Data.Word	WORD	Input/Output
❑ Data.Dword	DWORD	Input/Output
❑ Data.Qword	QWORD	Input/Output
❑ Data.pBuffer	PVOID	Input/Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pTrans	WD_TRANSFER elements:

Name	Description
cmdTrans	A value indicating the type of transfer command to perform – see definition of the WD_TRANSFER_CMD enumeration in windrvr.h . The transfer command should conform to the following format: <dir><p>[_S]<size> Explanation: <dir> : R for read, W for write <p> : P for I/O, M for memory <S> : signifies a string (block) transfer, as opposed to a single transfer <size> : BYTE, WORD, DWORD or QWORD
dwPort	For an I/O transfer - port address received from I.IO.dwAddr in WD_CardRegister() [A.4.11]. For a memory transfer - kernel-mode virtual memory address received from I.Mem.dwTransAddr in WD_CardRegister().
dwBytes	Used in string transfers - number of bytes to transfer.
fAutoinc	fAutoinc Used in string transfers - If TRUE, I/O or memory address should be incremented for transfer. If FALSE, all data is transferred to the same port/address.
dwOptions	Must be 0.
Data	The data to be translated.
Data.Byte	Used for 8-bit transfers.
Data.Word	Used for 16-bit transfers.
Data.Dword	Used for 32-bit transfers
Data.Qword	Used for 64-bit transfers
Data.pBuffer	Used in string (block) transfers – the pointer to the buffer with the data to read/write from/to.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- 64-bit data transfers (QWORD) are available only for memory read/write string operations.

64-bit data transfers (QWORD) require 64-bit enabled PCI device, 64-bit PCI bus, and an x86 CPU running under any of the operating systems supported by WinDriver. (Note: 64-bit data transfers performed with WD_Transfer() do not

require 64-bit operating system/CPU).

- When using `WD_Transfer()`, it is important to align the base address according to the size of the data type, especially when issuing string transfer commands. Otherwise, the transfers are split into smaller portions. **/ The easiest way to align data is to use basic types when defining a buffer, i.e.*

`BYTE buf[len];` */* for BYTE transfers - not aligned */* `WORD buf[len];` */* for WORD transfers - aligned on 2-byte boundary */* `UINT32 buf[len];` */* for DWORD transfers - aligned on 4-byte boundary */* `UINT64 buf[len];` */* for QWORD transfers - aligned on 8-byte boundary */*

EXAMPLE

```
WD_TRANSFER Trans;
BYTE read_data;

BZERO(Trans);
Trans.cmdTrans = RP_BYTE; /* Read Port BYTE */
Trans.dwPort = 0x210;
WD_Transfer(hWD, &Trans);
read_data = Trans.Data.Byte;
```

A.4.15 WD_MultiTransfer()

PURPOSE

- Executes a multiple read/write instruction to an I/O port or a memory address.

PROTOTYPE

```
DWORD WD_MultiTransfer(HANDLE hWD,
    WD_TRANSFER *pTransArray, DWORD dwNumTransfers);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pTransArray	Array of WD_TRANSFER *	
❑ cmdTrans	DWORD	Input
❑ dwPort	DWORD	Input
❑ dwBytes	DWORD	Input
❑ fAutoinc	DWORD	Input
❑ dwOptions	DWORD	Input
❑ Data	union	
❑ Data.Byte	BYTE	Input/Output
❑ Data.Word	WORD	Input/Output
❑ Data.Dword	DWORD	Input/Output
❑ Data.Qword	QWORD	Input/Output
❑ Data.pBuffer	PVOID	Input/Output
> dwNumTransfers	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pTransArray	WD_TRANSFER elements:

Name	Description
cmdTrans	Command of operation (WD_TRANSFER_CMD; please refer to windrvr.h for implementation). Should be typed in the following format: <dir><p>_<string><size> <ul style="list-style-type: none"> • dir - R for read, W for write. • p - P for I/O port, M for memory. • String - S for string, none for single transfer. • Size - BYTE, WORD, DWORD or QWORD.
dwPort	For an I/O transfer - port address received from I.IO.dwAddr in WD_CardRegister() [A.4.11]. For a memory transfer - kernel-mode virtual memory address received from I.Mem.dwTransAddr in WD_CardRegister().
dwBytes	Used in string transfers - number of bytes to transfer.
fAutoinc	fAutoinc Used in string transfers: If TRUE, I/O or memory address should be incremented for transfer. If FALSE, all data is transferred to the same port/address.
dwOptions	Must be 0
Data	The data buffer for the transfer
Data.Byte	Used for 8-bit transfers
Data.Word	Used for 16-bit transfers
Data.Dword	Used for 32-bit transfers
Data.Qword	Used for 64-bit transfers
Data.pBuffer	Used in string transfers - the pointer to the buffer with the data to read/write from/to
dwNumTransfers	Number of commands in array

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

See WD_Transfer() [A.4.14] remarks.

NOTE

This function is not supported in Visual Basic.

EXAMPLE

```
WD_TRANSFER Trans[4];
DWORD dwResult;
char *cData = "Message to send\n";

BZERO(Trans);
Trans[0].cmdTrans = WP_WORD; /* Write Port WORD */
Trans[0].dwPort = 0x1e0;
Trans[0].Data.Word = 0x1023;

Trans[1].cmdTrans = WP_WORD;
Trans[1].dwPort = 0x1e0;
Trans[1].Data.Word = 0x1022;

Trans[2].cmdTrans = WP_SBYTE; /* Write Port String BYTE */
Trans[2].dwPort = 0x1f0;
Trans[2].dwBytes = strlen(cData);
Trans[2].fAutoinc = FALSE;
Trans[2].dwOptions = 0;
Trans[2].Data.pBuffer = cData;

Trans[3].cmdTrans = RP_DWORD; /* Read Port Dword */
Trans[3].dwPort = 0x1e4;

WD_MultiTransfer(hWD, &Trans, 4);
dwResult = Trans[3].Data.Dword;
```

A.4.16 WD_DMALock()**PURPOSE**

- Enables Contiguous Buffer or Scatter Gather DMA.
- Locks a physical memory region and returns a list of the corresponding physical addresses.
- Returns a mapping of the physical address of the allocated buffer to the kernel virtual address space.
- For Contiguous Buffer DMA, returns a mapping of the physical address of the allocated buffer to the user-mode virtual address space. (In the case of Scatter/Gather DMA, the virtual user-mode address is provided by the caller as input to the function).

PROTOTYPE

```
DWORD WD_DMALock (HANDLE hWD, WD_DMA *pDma) ;
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDma	WD_DMA *	
□ hDma	DWORD	Output
□ pUserAddr	PVOID	Input/Output
□ pKernelAddr	KPTR	Output
□ dwBytes	DWORD	Input
□ dwOptions	DWORD	Input
□ dwPages	DWORD	Input/Output
□ hCard	DWORD	Input
□ Page	Array of WD_DMA_PAGE	
◆ pPhysicalAddr	KPTR	Output
◆ dwBytes	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].

Name	Description
pDma	WD_DMA elements.
hDma	Handle of DMA buffer to be used by WD_DMAUnlock() [A.4.17]. Returns 0 if failed.
pUserAddr	Pointer to the user-mode virtual memory. Input in the case of Scatter Gather and output in the case of contiguous buffer DMA.
pKernelAddr	Kernel mapping of kernel allocated buffer. Relevant only for Contiguous Buffer DMA (dwOptions = DMA_KERNEL_BUFFER_ALLOC).
dwBytes	Size of buffer.
dwOptions	<p>A bit mask flag:</p> <ul style="list-style-type: none"> • DMA_KERNEL_BUFFER_ALLOC: If set - Allocates contiguous buffer in physical memory. Default - Scatter Gather. • DMA_KBUF_BELOW_16M: Relevant only if DMA was set to contiguous (above). If set - Physical address will be allocated within the first 16MB of the main memory. • DMA_LARGE_BUFFER: Relevant only if DMA is set to Scatter Gather (above). If set - Enables locking more than 1MB. • DMA_ALLOW_CACHE: Allow caching of the memory. • DMA_KERNEL_ONLY_MAP: Relevant only if DMA was set to contiguous option with DMA_KERNEL_BUFFER_ALLOC flag. If set - the mapping of the allocated buffer is to the kernel only, not to the user mode. • DMA_READ_FROM_DEVICE: When set, the memory pages are locked to be read from, indicating writing to device. • DMA_WRITE_TO_DEVICE: When set, the memory pages are locked to be written to, indicating reading from device. <p>Note: Set either DMA_READ_FROM_DEVICE or DMA_WRITE_TO_DEVICE but not both.</p>
dwPages	<p>Number of pages.</p> <p>Returns 1 if DMA is set to contiguous.</p> <p>In case of DMA_LARGE_BUFFER it is used as an input and an output parameter (otherwise only output), describing the size of the page array; Please refer to the remark regarding dwPages in this section.</p>

Name	Description
hCard	Handle of relevant card as received from <code>WD_CardRegister()</code> [A.4.11]. You have the option of setting this field to 0, enabling locking of the kernel buffer without a device.
Page	<code>WD_DMA_PAGE</code> - Array of pages.
pPhysicalAddr	Pointer to the physical address.
dwBytes	Size of page.

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- WinDriver supports both Scatter/Gather and Contiguous DMA under Windows 98/Me/NT/2k/XP/CE/Server 2003, Linux, Solaris and VxWorks.
On Linux, Scatter/Gather DMA is only supported for 2.4 kernels and above (since the 2.2 Linux kernels require a patch to support this type of DMA).
- To access the allocated DMA buffer from your application, do NOT use the physical memory address (`dma.Page[i].pPhysicalAddr`) directly.
To access the memory from a user-mode application, use the user-mode virtual mapping of the address - `dma.pUserAddr`.
For Contiguous Buffer DMA, to access the memory from a kernel application (e.g. when using the Kernel PlugIn) or using WinDriver's API (e.g. `WD_Transfer()`), use the kernel mapping of the physical address, which is returned by `WD_DMALock()` in `dma.pKernelAddr`.
- On Solaris, the user buffer address and size must be aligned on system memory page boundary. Use the aligned memory allocation function, `valloc()` (similar to `malloc()`, except the allocated memory blocks are aligned). On SPARC platforms, the virtual page size is usually 8 KB, and on x86 platforms, it is 4 KB.
- When using the `DMA_LARGE_BUFFER` flag, `dwPages` is an input/output parameter. As an input to a `WD_DMALock()` call, `dwPages` equals the maximum number of elements in an array of pages. On return from `WD_DMALock()`, `dwPages` equals the number of actual physical blocks. The returned `dwPages` may be smaller, because adjacent pages are returned as one block.
- On Solaris x86 platforms, for contiguous buffer allocation greater than the physical page size (4K), use `WD_DMALock()` with `DMA_KERNEL_ONLY_MAP`. Then use the kernel mapping of the physical address, which is returned by `WD_DMALock()` in `dma.pKernelAddr` in `WD_Transfer()` to access the buffer.

EXAMPLE

The following code demonstrates Scatter/Gather DMA allocation:

```
WD_DMA dma;
DWORD dwStatus;
PVOID pBuffer = malloc(20000);

BZERO(dma);
dma.dwBytes = 20000;
dma.pUserAddr = pBuffer;
dma.dwOptions = fIsRead ? DMA_READ_FROM_DEVICE : DMA_WRITE_TO_DEVICE;
/* Initialization of dma.hCard, value obtained from WD_CardRegister call: */
dma.hCard = cardReg.hCard;
dwStatus = WD_DMALock(hWD, &dma);
if (dwStatus)
{
    printf("Could not lock down buffer\n");
}
else
{
    /* On successful return dma.Page has the list of
       physical addresses.
       To access the memory from your user mode
       application, use dma.pUserAddr. */
}
```

EXAMPLE

The following code demonstrates contiguous kernel buffer DMA allocation:

```
WD_DMA dma;
DWORD dwStatus;

BZERO(dma);
dma.dwBytes = 20 * 4096; /* 20 pages */
dma.dwOptions = DMA_KERNEL_BUFFER_ALLOC |
    ( fIsRead ? DMA_READ_FROM_DEVICE : DMA_WRITE_TO_DEVICE );
/* Initialization of dma.hCard, value obtained from WD_CardRegister call: */
dma.hCard = cardReg.hCard;
dwStatus = WD_DMALock(hWD, &dma);
if (dwStatus)
```

```
{
    printf("Failed allocating kernel buffer for DMA\n");
}
else
{
    /* On return dma.pUserAddr holds the user mode virtual
       mapping of the allocated memory and dma.pKernelAddr
       holds the kernel mapping of the physical memory.
       dma.Page[0].pPhysicalAddr points to the allocated
       physical address. */
}
```

A.4.17 WD_DMAUnlock()**PURPOSE**

- Unlocks a DMA buffer.

PROTOTYPE

```
DWORD WD_DMAUnlock(HANDLE hWD, WD_DMA *pDMA) ;
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDMA	WD_DMA *	
□ hDma	DWORD	Input
□ pUserAddr	PVOID	N/A
□ pKernelAddr	KPTR	N/A
□ dwBytes	DWORD	N/A
□ dwOptions	DWORD	N/A
□ dwPages	DWORD	N/A
□ Page	Array of WD_DMA_PAGE	N/A

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pDMA	WD_DMA elements:
hDma	Handle of DMA buffer received by WD_DMALock () [A.4.16].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_DMAUnlock(hWD, &dma);
```


A.4.18 WD_DMASyncCpu()**PURPOSE**

- Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.

NOTE: This function should be called before performing a DMA transfer (see Remarks below).

PROTOTYPE

```
DWORD WD_DMASyncCpu(HANDLE hWD, WD_DMA *pDMA);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDMA	WD_DMA *	

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pDMA	DMA information structure – see WD_DMALock() [A.4.16]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- An asynchronous DMA read or write operation accesses data in memory, not in the processor (CPU) cache, which resides between the CPU and the host's physical memory. Unless the CPU cache has been flushed, by calling WD_DMASyncCpu(), just before a read transfer, the data transferred into system memory by the DMA operation could be overwritten with stale data if the CPU cache is flushed later. Unless the CPU cache has been flushed by calling

WD_DMASyncCpu() just before a write transfer, the data in the CPU cache might be more up-to-date than the copy in memory.

EXAMPLE

```
WD_DMASyncCpu(hWD, &dma);
```

A.4.19 WD_DMASyncIo()

PURPOSE

- Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.

NOTE: This function should be called after performing a DMA transfer (see Remarks below).

PROTOTYPE

```
DWORD WD_DMASyncIo (HANDLE hWD, WD_DMA *pDMA) ;
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDMA	WD_DMA *	

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open () [A.7.2].
pDMA	DMA information structure – see WD_DMALock () [A.4.16]

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- After a DMA transfer has been completed, the data can still be in the I/O cache, which resides between the host's physical memory and the bus-master DMA device, but not yet in the host's main memory. If the CPU accesses the memory, it might read the wrong data from the CPU cache. To ensure a consistent view of the memory for the CPU, you should call WD_DMASyncIo () after a DMA transfer in order to flush the data from the I/O cache and update

the CPU cache with the new data. The function also flushes additional caches and buffers between the device and memory, such as caches associated with bus extenders or bridges.

EXAMPLE

```
WD_DMASyncIo(hWD, &dma);
```

A.4.20 WD_PcmciaControl()**PURPOSE**

- Modifies the settings of a PCMCIA bus controller

PROTOTYPE

```
DWORD WD_PcmciaControl (HANDLE hWD, WD_PCMCIA_CONTROL *pPcmciaControl);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pPcmciaControl	WD_PCMCIA_CONTROL*	Input
□ pcmciaSlot	WD_PCMCIA_SLOT	Input
◆ uBus	BYTE	Input
◆ uSocket	BYTE	Input
◆ uFunction	BYTE	Input
□ uAccessSpeed	BYTE	Input
□ uBusWidth	BYTE	Input
□ uVppLevel	BYTE	Input
□ dwCardBase	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pPcmciaControl	PCMCIA bus controller information structure:
pcmciaSlot	A PCMCIA device location information structure, which can be acquired by calling WDC_PcmciaScanDevices() [A.2.5]
uBus	Bus number
uSocket	Socket number
uFunction	Function number

Name	Description
uAccessSpeed	The access speed to the PCMCIA bus. Can be any of the following WD_PCMCIA_ACC_SPEED enumeration values: <ul style="list-style-type: none"> • WD_PCMCIA_ACC_SPEED_DEFAULT: Use the default access speed • WD_PCMCIA_ACC_SPEED_250NS: 250 ns • WD_PCMCIA_ACC_SPEED_200NS: 200 ns • WD_PCMCIA_ACC_SPEED_150NS: 150 ns • WD_PCMCIA_ACC_SPEED_1000NS: 100 ns
uBusWidth	The PCMCIA bus width. Can be any of the following WD_PCMCIA_ACC_WIDTH enumeration values: <ul style="list-style-type: none"> • WD_PCMCIA_ACC_WIDTH_DEFAULT: Use the default bus width • WD_PCMCIA_ACC_WIDTH_8BIT: 8 bit • WD_PCMCIA_ACC_WIDTH_16BIT: 16 bit
uVppLevel	The power level of the PCMCIA controller's Voltage Power Pin (Vpp). Can be any of the following WD_PCMCIA_VPP enumeration values: <ul style="list-style-type: none"> • WD_PCMCIA_VPP_DEFAULT: Use the default power level of the PCMCIA Vpp pin • WD_PCMCIA_VPP_OFF: Set the voltage on the Vpp pin to zero (disable) • WD_PCMCIA_VPP_ON: Set the voltage on the Vpp pin to 12V (enable) • WD_PCMCIA_VPP_AS_VCC: Set the voltage on the Vpp pin to equal that of the Vcc pin
dwCardBase	The offset in the PCMCIA device's memory from which the memory mapping begins

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_PCMCIA_CONTROL pcmciaControl;
BZERO(pcmciaControl);

pcmciaControl.pcmciaSlot = pcmciaSlot; /* pcmciaSlot recieved from
    WD_PcmciaScanDevices() */
pcmciaControl.uAccessSpeed = WD_PCMCIA_ACC_SPEED_DEFAULT;
```

```
pcmciaControl.uBusWidth = WD_PCMCIA_ACC_WIDTH_DEFAULT;  
pcmciaControl.uVppLevel = WD_PCMCIA_VPP_AS_VCC;  
pcmciaControl.dwCardBase = 0x0;
```

```
WD_PcmciaControl(hWD, &pcmciaControl);
```

A.4.21 InterruptEnable()

PURPOSE

- Call a callback function upon interrupt reception. A convenient function for setting up interrupt handling.

PROTOTYPE

```
DWORD InterruptEnable(HANDLE *phThread , HANDLE hWD,
                     WD_INTERRUPT *pInt , INT_HANDLER func , PVOID pData);
```

PARAMETERS

Name	Type	Input/Output
> phThread	HANDLE *	Output
> hWD	HANDLE	Input
> pInt	WD_INTERRUPT *	
□ hInterrupt	HANDLE	Input
□ dwOptions	DWORD	Input
□ Cmd	WD_TRANSFER *	Input
□ dwCmds	DWORD	Input
□ kpCall	WD_KERNEL_PLUGIN_CALL	
◆ hKernelPlugIn	DWORD	Input
◆ dwResult	DWORD	N/A
□ fEnableOk	DWORD	N/A
□ dwCounter	DWORD	N/A
□ dwLost	DWORD	N/A
□ fStopped	DWORD	N/A
> func	INT_HANDLER	Input
> pData	PVOID	Input

DESCRIPTION

Name	Description
phThread	Returns the handle of the spawned interrupt thread to be used by InterruptDisable() [A.4.22].
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].

Name	Description
pInt	WD_INTERRUPT elements:
hInterrupt	Handle of interrupt internal data structure received by I.Int.hInterrupt in WD_CardRegister() [A.4.11].
dwOptions	<p>A bit mask flag. May be "0" for no option, or:</p> <ul style="list-style-type: none"> • INTERRUPT_CMD_COPY: if set, the WinDriver kernel will copy the data received from the read commands that were used to acknowledge the interrupt, back to the user mode. The data will be available when function is called.
Cmd	<p>An array of transfer commands information structures (WD_TRANSFER *) that define the operations to be performed at the kernel level upon the detection of an interrupt, or NULL if no transfer commands are required.</p> <p>NOTE: When handling level sensitive interrupts (such as PCI interrupts) without a Kernel PlugIn driver, you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received – see section 9.2 for details.</p> <p>The commands in the array can be either of the following:</p> <ul style="list-style-type: none"> • A read/write transfer command that conforms to the following format: <dir><p>_[S]<size> – see the description of the cmdTrans field of the WD_TRANSFER structure in section A.4.14 for details. • CMD_MASK: Interrupt mask command for determining the source of the interrupt: When this command is set, upon the arrival of an interrupt in the kernel WinDriver compares the value of the previous read command in the Cmd array with the mask that is set in the relevant Data field union member of the mask transfer command. For example, if Cmd[i-1].cmdTrans is RM_BYTE, WinDriver performs the following check: Cmd[i-1].Data.Byte & Cmd[i].Data.Byte. If the values match, the driver claims ownership of the interrupt and invokes your interrupt handler routine (funcIntHandler) when the control is returned to the user mode; otherwise, the driver rejects ownership of the interrupt, the interrupt handler routine is not invoked and the subsequent transfer commands in the Cmd array are not executed. <p>NOTE: A CMD_MASK command must be preceded by a read transfer command (RM_XXX / RP_XXX).</p>

Name	Description
dwCmds	Number of transfer commands in Cmd array.
kpCall	WD_KERNEL_PLUGIN_CALL elements:
hKernelPlugIn	Handle to Kernel PlugIn returned from WD_KernelPlugInOpen() [A.8.1].
func	The interrupt handling function that will be called once at every interrupt occurrence. INT_HANDLER is defined in windrvr_int_thread.h.
pData	The pointer that is passed to the interrupt handling function as an argument.
Return Value	TRUE if enabling the interrupt succeeded.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- Implemented in `\WinDriver\src\windrvr_int_thread.c`.
- WD_IntEnable(), WD_IntWait(), WD_IntCount() and WD_IntDisable() compose the above InterruptEnable() and InterruptDisable() functions and can be called separately instead. For more details, please refer to Section A.5.
- To improve the PCI interrupt handling rate on Windows 98/Me/NT/2000/XP/Server 2003, Linux and Solaris, consider using WinDriver's Kernel PlugIn feature (see Section "Understanding the Kernel PlugIn"). On VxWorks, you can use the windrvr_isr() callback function (see Section "Improving the Interrupt Handling Rate on VxWorks").

EXAMPLE

```

VOID DLLCALLCONV interrupt_handler(PVOID pData)
{
    WD_INTERRUPT *pIntrp = (WD_INTERRUPT *)pData;
    /* implement your interrupt handler routine here */

    printf("Got interrupt %d\n", pIntrp->dwCounter);
}

....
main()

```

```

{
    WD_CARD_REGISTER cardReg;
    WD_INTERRUPT Intrp;
    HANDLE hWD, thread_handle;

    ....
    hWD = WD_Open();
    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_INTERRUPT;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.Int.dwInterrupt = MY_IRQ;
    cardReg.Card.Item[0].I.Int.dwOptions = 0;
    ....
    WD_CardRegister(hWD, &cardReg);
    ....
    PVOID pdata = NULL;
    BZERO (Intrp);
    Intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
    Intrp.Cmd = NULL;
    Intrp.dwCmds = 0;
    Intrp.dwOptions = 0;
    printf("starting interrupt thread\n");
    pData = &Intrp;
    if (!InterruptEnable(&thread_handle, hWD, &Intrp,
        interrupt_handler, pdata))
    {
        printf ("failed enabling interrupt\n")
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);
        /* this calls WD_IntDisable() */
        InterruptDisable(thread_handle);
    }
    WD_CardUnregister(hWD, &cardReg);
    ....
}

```

A.4.22 InterruptDisable()

PURPOSE

- A convenient function for shutting down interrupt handling.

PROTOTYPE

```
DWORD InterruptDisable (HANDLE hThread);
```

PARAMETERS

Name	Type	Input/Output
> hThread	HANDLE	Input

DESCRIPTION

Name	Description
phThread	The handle of the spawned interrupt thread which was created by InterruptEnable() [A.4.21].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

- Implemented in \WinDriver\src\windrvr_int_thread.c.
- WD_IntEnable(), WD_IntWait(), WD_IntCount() and WD_IntDisable() compose the above InterruptEnable and InterruptDisable functions and can be called separately instead. For more details, please refer to Section A.5.

EXAMPLE

```
main()
{
    ....
    if (!InterruptEnable(&thread_handle, hWD, &Intrp,
```

```
        interrupt_handler, pData))
    {
        printf("failed enabling interrupt\n");
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);
        /* this calls WD_IntDisable() */
        InterruptDisable(thread_handle);
    }
    ....
}
```

A.5 PCI/PCMCIA/ISA - Low Level WD_xxx Functions

This section describes low-level WD_xxx PCI/PCMCIA/ISA WinDriver functions.

NOTE

It is recommended to use the API from WinDriver's **WDC** library, which provides convenient wrappers to the basic WD_xxx PCI/PCMCIA/ISA API [A.1]. If you decide not to use the WDC API, consider using the high-level WD_xxx API, described in section A.4, instead of using the low-level functions described below directly.

A.5.1 WinDriver Low-Level Interrupt Calling Sequence

The following is a typical calling sequence of the WinDriver API, used for servicing interrupts. The InterruptEnable() and InterruptDisable() functions enable interrupt handling in a more convenient manner.

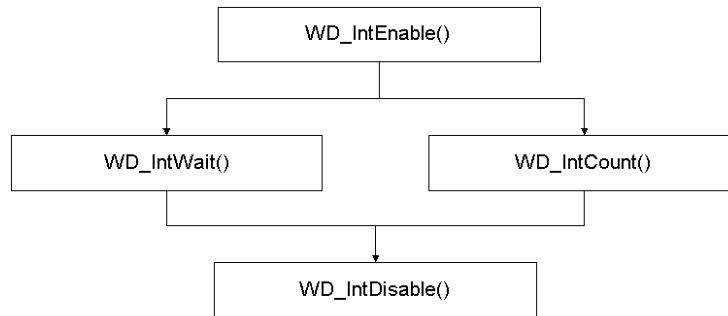


Figure A.2: Low-Level WinDriver Interrupt API Calling Sequence

A.5.2 WD_IntEnable()

PURPOSE

- Register an internal interrupt service routine (ISR) to be called upon interrupt.

PROTOTYPE

```
DWORD WD_IntEnable(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pInterrupt	WD_INTERRUPT *	
□ hInterrupt	HANDLE	Input
□ dwOptions	DWORD	Input
□ Cmd	WD_TRANSFER *	Input
□ dwCmds	DWORD	Input
□ kpCall	WD_KERNEL_PLUGIN_CALL	
◆ hKernelPlugIn	HANDLE	Input
◆ dwMessage	DWORD	N/A
◆ pData	PVOID	N/A
◆ dwResult	DWORD	N/A
□ fEnableOk	DWORD	Output
□ dwCounter	DWORD	N/A
□ dwLost	DWORD	N/A
□ fStopped	DWORD	N/A

DESCRIPTION

Name	Description
HWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pInterrupt	WD_INTERRUPT elements:
hInterrupt	Handle of interrupt to enable. The handle is returned by WD_CardRegister() [A.4.11] in I.Int.hInterrupt.

Name	Description
dwOptions	<p>A bit mask flag. May be 0 for no option, or:</p> <ul style="list-style-type: none"> • INTERRUPT_CMD_COPY: if set, the WinDriver kernel will copy the data received from the read commands that were used to acknowledge the interrupt, back to the user mode. The data will be available when <code>WD_IntWait()</code> [A.5.3] returns.
Cmd	<p>An array of transfer commands information structures (<code>WD_TRANSFER *</code>) that define the operations to be performed at the kernel level upon the detection of an interrupt, or NULL if no transfer commands are required.</p> <p>NOTE: When handling level sensitive interrupts (such as PCI interrupts) without a Kernel PlugIn driver, you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received – see section 9.2 for details.</p> <p>The commands in the array can be either of the following:</p> <ul style="list-style-type: none"> • A read/write transfer command that conforms to the following format: <code><dir><p>[_S]<size></code> – see the description of the <code>cmdTrans</code> field of the <code>WD_TRANSFER</code> structure in section A.4.14 for details. • CMD_MASK: Interrupt mask command for determining the source of the interrupt: When this command is set, upon the arrival of an interrupt in the kernel WinDriver compares the value of the previous read command in the <code>Cmd</code> array with the mask that is set in the relevant <code>Data</code> field union member of the mask transfer command. For example, if <code>Cmd[i-1].cmdTrans</code> is <code>RM_BYTE</code>, WinDriver performs the following check: <code>Cmd[i-1].Data.Byte & Cmd[i].Data.Byte</code>. If the values match, the driver claims ownership of the interrupt and invokes your interrupt handler routine (<code>funcIntHandler</code>) when the control is returned to the user mode; otherwise, the driver rejects ownership of the interrupt, the interrupt handler routine is not invoked and the subsequent transfer commands in the <code>Cmd</code> array are not executed. <p>NOTE: A CMD_MASK command must be preceded by a read transfer command (<code>RM_XXX / RP_XXX</code>).</p>
dwCmds	Number of transfer commands in <code>Cmd</code> array.
kpCall	WD_KERNEL_PLUGIN_CALL elements:

Name	Description
hKernelPlugIn	Handle to Kernel PlugIn returned from WD_KernelPlugInOpen() [A.8.1].
fEnableOk	Returns TRUE if WD_IntEnable() [A.5.2] succeeded.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

(1) For more information regarding interrupt handling please refer to ISA_PCI interrupts section in the WinDriver manual.

(2) kpCall is relevant for Kernel PlugIn implementation.

EXAMPLES

```

• WD_INTERRUPT Intrp;
  WD_CARD_REGISTER cardReg;

  BZERO(cardReg);
  cardReg.Card.dwItems = 1;
  cardReg.Card.Item[0].item = ITEM_INTERRUPT;
  cardReg.Card.Item[0].fNotSharable = TRUE;
  cardReg.Card.Item[0].I.Int.dwInterrupt = 10; /* IRQ 10 */
  /* INTERRUPT_LEVEL_SENSITIVE - Set to level sensitive
     interrupts, otherwise should be 0.
     ISA cards are usually edge triggered while PCI cards
     are usually level sensitive. */
  cardReg.Card.Item[0].I.Int.dwOptions =
      INTERRUPT_LEVEL_SENSITIVE;
  cardReg.fCheckLockOnly = FALSE;
  WD_CardRegister(hWD, &cardReg);
  if (cardReg.hCard == 0)
      printf("Could not lock device\n");
  else
  {
      BZERO(Intrp);
      Intrp.hInterrupt =
          cardReg.Card.Item[0].I.Int.hInterrupt;
      Intrp.Cmd = NULL;
      Intrp.dwCmds = 0;
  }

```

```
Intrp.dwOptions = 0;
WD_IntEnable(hWD, &Intrp);
}
if (!Intrp.fEnableOk)
{
    printf("Failed enabling interrupt\n");
}
```

- For another example please refer to
<WinDriver>\Samples\pci_diag\pci_lib.c.

A.5.3 WD_IntWait()

PURPOSE

- Wait until an interrupt is received or disabled and exit.

PROTOTYPE

```
DWORD WD_IntWait(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pInterrupt	WD_INTERRUPT *	
❑ hInterrupt	HANDLE	Input
❑ dwOptions	DWORD	N/A
❑ Cmd	WD_TRANSFER *	N/A
❑ dwCmds	DWORD	N/A
❑ kpCall	WD_KERNEL_PLUGIN_CALL	N/A
❑ fEnableOk	DWORD	N/A
❑ dwCounter	DWORD	Output
❑ dwLost	DWORD	Output
❑ fStopped	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pInterrupt	WD_INTERRUPT elements:
hInterrupt	Handle of interrupt, returned by WD_CardRegister() [A.4.11] in I.Int.hInterrupt.
dwCounter	Number of interrupts received.
dwLost	Number of interrupts that were acknowledge in kernel mode but not yet handled in user mode.

Name	Description
fStopped	Returns zero if an interrupt occurred. Returns INTERRUPT_STOPPED if an interrupt was disabled while waiting. Returns INTERRUPT_INTERRUPTED if while waiting for an interrupt, WD_IntWait() [A.5.3] was interrupted without an actual hardware interrupt.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

INTERRUPT_INTERRUPTED status can occur on Linux and Solaris if the application that waits on the interrupt is stopped (e.g. by pressing CTRL+Z).

EXAMPLE

```
for (;;)
{
    WD_IntWait(hWD, &Intrp);
    if (Intrp.fStopped)
        break;

    ProcessInterrupt(Intrp.dwCounter);
}
```

A.5.4 WD_IntCount()

PURPOSE

- Retrieve the count number of interrupts since WD_IntEnable() [A.5.2] was called.

PROTOTYPE

```
void WD_IntCount(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pInterrupt	WD_INTERRUPT *	
□ hInterrupt	HANDLE	Input
□ dwOptions	DWORD	N/A
□ Cmd	WD_TRANSFER *	N/A
□ dwCmds	DWORD	N/A
□ kpCall	WD_KERNEL_PLUGIN_CALL	N/A
□ fEnableOk	DWORD	N/A
□ dwCounter	DWORD	Output
□ dwLost	DWORD	Output
□ fStopped	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pInterrupt	WD_INTERRUPT elements:
hInterrupt	Handle of interrupt, returned by WD_CardRegister() [A.4.11] in I.Int.hInterrupt.
dwCounter	Number of interrupts received.
dwLost	Number of interrupts not yet handled.
fStopped	Returns TRUE if interrupt was disabled while waiting.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

EXAMPLE

```
DWORD dwNumInterrupts;  
  
WD_IntCount(hWD, &Intrp);  
dwNumInterrupts = Intrp.dwCounter;
```

A.5.5 WD_IntDisable()

PURPOSE

- Disable interrupt processing.

PROTOTYPE

```
DWORD WD_IntDisable(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pInterrupt	WD_INTERRUPT	
□ hInterrupt	HANDLE	Input
□ dwOptions	DWORD	N/A
□ Cmd	WD_TRANSFER *	N/A
□ dwCmds	DWORD	N/A
□ kpCall	WD_KERNEL_PLUGIN_CALL	N/A
□ fEnableOk	DWORD	N/A
□ dwCounter	DWORD	N/A
□ dwLost	DWORD	N/A
□ fStopped	DWORD	N/A

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pInterrupt	WD_INTERRUPT elements:
hInterrupt	Handle of interrupt, returned by WD_CardRegister() [A.4.11] in I.Int.hInterrupt.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_IntDisable(hWD, &Intrp);
```


A.6 Plug and Play and Power Management

A.6.1 Calling Sequence

The following is a typical calling sequence of the WinDriver API, used for handling Plug and Play and power management events.

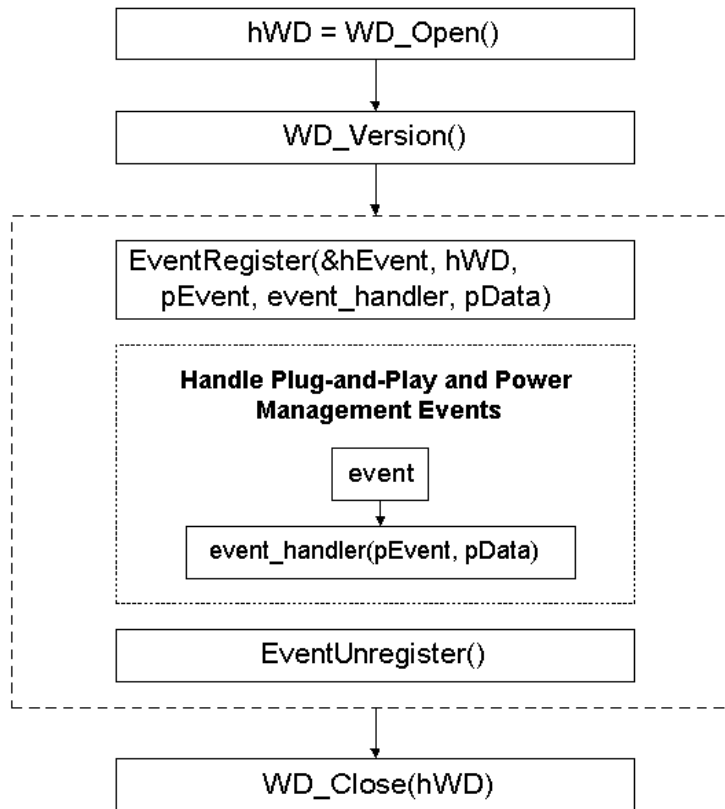


Figure A.3: Plug and Play Calling Sequence

A.6.2 EventRegister()

PURPOSE

• Register your application to receive Plug and Play and power management event notifications, according to a predefined set of criteria, and call a callback function upon event receipt.

PROTOTYPE

```
DWORD EventRegister(HANDLE *phEvent, HANDLE hWD,
    WD_EVENT *pEvent, EVENT_HANDLER pFunc, void *pData);
```

PARAMETERS

Name	Type	Input/Output
➤ phEvent	HANDLE *	Output
➤ hWD	HANDLE	Input
➤ event	WD_EVENT *	Input
☐ handle	DWORD	Output
☐ dwAction	DWORD	Input
☐ dwStatus	DWORD	N/A
☐ dwEventId	DWORD	N/A
☐ dwCardType	WD_BUS_TYPE	Input
☐ hKernelPlugIn	DWORD	Input
☐ dwOptions	DWORD	Input
☐ u	union	
◆ Pci	struct	
◇ cardId	WD_PCI_ID	
◆ dwVendorId	DWORD	Input
◆ dwDeviceId	DWORD	Input
◇ pciSlot	WD_PCI_SLOT	
◆ dwBus	DWORD	Input
◆ dwSlot	DWORD	Input
◆ dwFunction	DWORD	Input
◆ Pcmcia	struct	
◇ deviceId	WD_PCMCIA_ID	
◆ wManufacturerId	WORD	Input
◆ wCardId	WORD	Input
◇ pcmciaSlot	WD_PCMCIA_SLOT	

Name	Type	Input/Output
◆ uBus	BYTE	Input
◆ uSocket	BYTE	Input
◆ uFunction	BYTE	Input
◆ uPadding	BYTE	N/A
➤ func	EVENT_HANDLER	Input
➤ data	void	Input
➤ dwEventVer	DWORD	Internal use
➤ dwNumMatchTables	DWORD	Input
➤ matchTables[1]	WDU_MATCH_TABLE	Input

DESCRIPTION

Name	Description
phEvent	If successful, phEvent will hold the handle to be used in EventUnregister() [A.6.3].
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
event	The criteria set for registering to receive event notifications.
handle	Optional handle to be used by the low-level WD_EventUnregister() function; 0 when event registration fails.

Name	Description
dwAction	<p>A bit mask field indicating which events to register to.</p> <p>Plug and Play events:</p> <ul style="list-style-type: none"> • WD_INSERT - Device inserted • WD_REMOVE - Device removed <p>Device power state:</p> <ul style="list-style-type: none"> • WD_POWER_CHANGED_D0 - Full power • WD_POWER_CHANGED_D1 - Low sleep • WD_POWER_CHANGED_D2 - Medium sleep • WD_POWER_CHANGED_D3 - Full sleep • WD_POWER_SYSTEM_WORKING - Fully on <p>Systems power state:</p> <ul style="list-style-type: none"> • WD_POWER_SYSTEM_SLEEPING1 - Fully on but sleeping • WD_POWER_SYSTEM_SLEEPING2 - CPU off, memory on, PCI/PCMCIA on • WD_POWER_SYSTEM_SLEEPING3 - CPU off, Memory is in refresh, PCI/PCMCIA on aux power • WD_POWER_SYSTEM_HIBERNATE - OS saves context before shutdown • WD_POWER_SYSTEM_SHUTDOWN - No context saved
dwCardType	Should be WD_BUS_PCI or WD_BUS_PCMCIA (from the WD_BUS_TYPE options).
hKernelPlugIn	Handle to Kernel PlugIn returned from WD_KernelPlugInOpen() [A.8.1] (when using the Kernel PlugIn to handle the events).
dwOptions	Can be either WD_ACKNOWLEDGE or zero. If WD_ACKNOWLEDGE, the user can perform actions on the requested event before acknowledging it. The OS waits on the event until the user calls WD_EventSend(). If the EventRegister() [A.6.2] wrapper is called, WD_EventSend() will be called automatically after the callback function exits.
cardId.dwVendorId	PCI Vendor ID to register to. If zero, register to all PCI vendor ID's.
cardId.dwDeviceId	PCI Device ID to register to. If zero, register to all PCI Device ID's.
pciSlot.dwBus	PCI bus number to register to. If zero, register to all PCI buses.
pciSlot.dwSlot	PCI slot to register to. If zero, register to all slots.
pciSlot.dwFunction	PCI function (on the device) to register to. If zero, registers to all functions.

Name	Description
deviceId.wManufacturerId	PCMCIA Manufacturer ID to register to. If zero, register to all PCMCIA manufacturer ID's.
deviceId.wCardId	PCMCIA card ID to register to. If zero, register to all PCMCIA card ID's.
pcmciaSlot.uBus	PCMCIA bus number to register to. If zero, register to all PCMCIA buses.
pcmciaSlot.uSocket	PCMCIA socket to register to. If zero, register to all sockets.
pcmciaSlot.uFunction	PCMCIA function (on the card) to register to. If zero, registers to all functions.
pcmciaSlot.uPadding	1 byte padding (reserved).
func	The callback function to call upon receipt of event notification.
data	The data to pass to the callback function.
dwEventVer	For internal use only.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

This function wraps the low-level WD_EventRegister(), WD_EventPull(), WD_EventSend() and InterruptEnable() [A.4.21] functions.

EXAMPLE

```

HANDLE *event_handle;
WD_EVENT event;
DWORD dwStatus;
BZERO(event);
event.dwAction = WD_INSERT | WD_REMOVE;
event.dwCardType = WD_BUS_PCI;
dwStatus = EventRegister(&event_handle, hWD, &event,
    event_handler_func, NULL);
if (dwStatus!=WD_STATUS_SUCCESS)
{
    printf("Failed register\n");
    return;
}

```

A.6.3 EventUnregister()

PURPOSE

- Un-registers from receiving Plug and Play and power management event notifications.

PROTOTYPE

```
DWORD EventUnregister(HANDLE hEvent);
```

PARAMETERS

Name	Type	Input/Output
➤ hEvent	HANDLE *	Input

DESCRIPTION

Name	Description
hEvent	Handle received from EventRegister() [A.6.2].

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

This function wraps WD_EventUnregister() and InterruptDisable() [A.4.22].

EXAMPLE

```
EventUnregister(event_handle);
```

A.7 General WD_xxx Functions

A.7.1 Calling Sequence WinDriver – General Use

The following is a typical calling sequence for the WinDriver API.

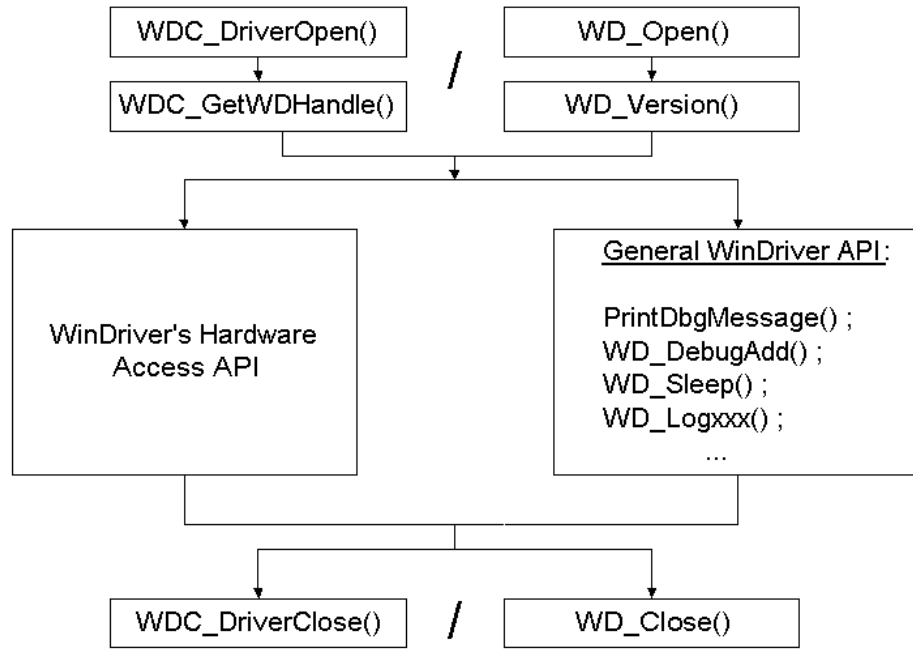


Figure A.4: WinDriver API Calling Sequence

NOTES

- (1) We recommend calling the WinDriver function `WD_Version()` [A.7.3] after calling `WD_Open()` [A.7.2] and before calling any other WinDriver function. Its purpose is to return the WinDriver kernel module (windrvr) version number, thus providing the means to verify that your application is version compatible with the WinDriver kernel module.
- (2) `WD_DebugAdd()` [A.7.6] and `WD_Sleep()` [A.7.8] can be called anywhere after `WD_Open()`.
- (3) Visual Basic and Delphi programmers should note that this Function Reference is C-oriented.
WinDriver Visual Basic and Delphi codes have been written as closely as possible to the C code, to enable maximal compatibility for all users. Most of the APIs have a single implementation that can be used from a C, VB or Delphi application. However, some of the WinDriver functions require a specific implementation for VB and Delphi. Please refer to the relevant Delphi/Visual Basic samples and include files:
 1. `\WinDriver\delphi`
 2. `\WinDriver\vb`

A.7.2 WD_Open()

PURPOSE

- Opens a handle to access the WinDriver kernel module. The handle is used by all WinDriver APIs, and therefore must be called before any other WinDriver API is called.

PROTOTYPE

```
HANDLE WD_Open() ;
```

RETURN VALUE

The handle to the WinDriver kernel module.

If device could not be opened, returns INVALID_HANDLE_VALUE.

REMARKS

If you are a registered user, please refer to `WD_License()` [\[A.7.9\]](#) function reference to see an example of how to register your license.

EXAMPLE

```
HANDLE hWD;  
  
hWD = WD_Open();  
if (hWD==INVALID_HANDLE_VALUE)  
{  
    printf("Cannot open WinDriver device\n");  
}
```

A.7.3 WD_Version()

PURPOSE

- Returns the version number of the WinDriver kernel module currently running.

PROTOTYPE

```
DWORD WD_Version(HANDLE hWD, WD_VERSION *pVer);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pVer	WD_VERSION *	
dwVer	DWORD	Output
cVer[100]	CHAR	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pVer	WD_VERSION elements:
dwVer	The version number.
cVer[100]	Version info string.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_VERSION ver;

BZERO(ver);
WD_Version(hWD, &ver);
printf("%s\n", ver.cVer)
```

```
if (ver.dwVer < WD_VER)
{
    printf("Error - incorrect WinDriver version\n");
}
```

A.7.4 WD_Close()

PURPOSE

- Closes the access to the WinDriver kernel module.

PROTOTYPE

```
void WD_Close(HANDLE hWD);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].

REMARKS

This function must be called when you finish using WinDriver kernel module.

EXAMPLE

```
WD_Close(hWD);
```

A.7.5 WD_Debug()

PURPOSE

- Sets debugging level for collecting debug messages.

PROTOTYPE

```
DWORD WD_Debug(HANDLE hWD, WD_DEBUG *pDebug);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDebug	WD_DEBUG *	Input
□ dwCmd	DWORD	Input
□ dwLevel	DWORD	Input
□ dwSection	DWORD	Input
□ dwLevelMessageBox	DWORD	Input
□ dwBufferSize	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pDebug	WD_DEBUG elements:
dwCmd	Debug command: Set filter, Clear buffer, etc. For more details please refer to DEBUG_COMMAND in windrvr.h .
dwLevel	Used for dwCmd=DEBUG_SET_FILTER. Sets the debugging level to collect: Error, Warning, Info, Trace. For more details please refer to DEBUG_LEVEL in windrvr.h .
dwSection	Used for dwCmd=DEBUG_SET_FILTER. Sets the sections to collect: IO, Mem, Int, etc. Use S_ALL for all. For more details please refer to DEBUG_SECTION in windrvr.h .

Name	Description
dwLevelMessageBox	Used for dwCmd=DEBUG_SET_FILTER. Sets the debugging level to print in a message box. For more details please refer to DEBUG_LEVEL in windrvr.h .
dwBufferSize	Used for dwCmd=DEBUG_SET_BUFFER. The size of buffer in the kernel.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

EXAMPLE

```
WD_DEBUG dbg;  
  
BZERO(dbg);  
dbg.dwCmd = DEBUG_SET_FILTER;  
dbg.dwLevel = D_ERROR;  
dbg.dwSection = S_ALL;  
dbg.dwLevelMessageBox = D_ERROR;  
  
WD_Debug(hWD, &dbg);
```

A.7.6 WD_DebugAdd()

PURPOSE

- Sends debug messages to the debug log. Used by the driver code.

PROTOTYPE

```
DWORD WD_DebugAdd (HANDLE hWD, WD_DEBUG_ADD *pData) ;
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pData	WD_DEBUG_ADD *	
□ dwLevel	DWORD	Input
□ dwSection	DWORD	Input
□ pcBuffer	CHAR [256]	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pData	WD_DEBUG_ADD elements:
dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If dwLevel is 0, D_ERROR will be declared. For more details please refer to DEBUG_LEVEL in windrvr.h .
dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If dwSection is 0, S_MISC section will be declared. For more details please refer to DEBUG_SECTION in windrvr.h .
pcBuffer	The string to copy into the message log.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_DEBUG_ADD add;

BZERO(add);
add.dwLevel = D_WARN;
add.dwSection = S_MISC;
sprintf(add.pcBuffer, "This message will be displayed in "
        "the debug monitor\n");
WD_DebugAdd(hWD, &add);
```


A.7.7 WD_DebugDump()

PURPOSE

- Retrieves debug messages buffer.

PROTOTYPE

```
DWORD WD_DebugDump (HANDLE hWD, WD_DEBUG_DUMP *pDebugDump );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pDebug	WD_DEBUG_DUMP *	Input
□ pcBuffer	PCHAR	Input/Output
□ dwSize	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pDebugDump	WD_DEBUG_DUMP elements:
pcBuffer	Buffer to receive debug messages
dwSize	Size of buffer in bytes

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
char buffer[1024];
WD_DEBUG_DUMP dump;
dump.pcBuffer=buffer;
dump.dwSize = sizeof(buffer);
WD_DebugDump(hWD, &dump);
```

A.7.8 WD_Sleep()

PURPOSE

- Delays execution for a specific duration of time.

PROTOTYPE

```
DWORD WD_Sleep (HANDLE hWD, WD_SLEEP *pSleep) ;
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pSleep	WD_SLEEP *	
dwMicroSeconds	DWORD	Input
dwOptions	DWORD	Input

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pSleep	WD_SLEEP elements:
dwMicroSeconds	Sleep time in microseconds - 1/1,000,000 of a second.
dwOptions	A bit mask flag: <ul style="list-style-type: none"> • SLEEP_NON_BUSY - If set, delays execution without consuming CPU resources. (Not relevant for under 17,000 micro seconds. Less accurate than busy sleep). Default - Busy sleep.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

Example usage: to access slow response hardware.

EXAMPLE

```
WD_Sleep slp;  
  
BZERO(slp);  
slp.dwMicroSeconds = 200;  
WD_Sleep(hWD, &slp);
```

A.7.9 WD_License()

PURPOSE

- Transfers the license string to the WinDriver kernel module and returns information regarding the license type of the specified license string.

PROTOTYPE

```
DWORD WD_License(HANDLE hWD, WD_LICENSE *pLicense);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pLicense	WD_LICENSE *	
□ cLicense[]	CHAR	Input
□ dwLicense	DWORD	Output
□ dwLicense2	DWORD	Output

DESCRIPTION

Name	Description
hWD	The handle to WinDriver's kernel-mode driver received from WD_Open() [A.7.2].
pLicense	WD_LICENSE elements:
cLicense[]	A buffer to contain the license string that is to be transferred to the WinDriver kernel module. If an empty string is transferred, then WinDriver kernel module returns the current license type to the parameter dwLicense.
dwLicense	Returns the license type of the specified license string (cLicense). The return value is a mask of license type flags, defined as an enum in windrvr.h . 0 = Invalid license string. Additional flags for determining the license type will be returned in dwLicense2, if needed.
dwLicense2	Returns additional flags for determining the license type, if dwLicense could not hold all the relevant information (otherwise - 0).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

When using a registered version, this function must be called before any other WinDriver API call, apart from WD_Open(), in order to register the license from the code.

Example usage: Add registration routine to your application.

EXAMPLE

```
DWORD RegisterWinDriver()
{
    HANDLE hWD;
    WD_LICENSE lic;
    DWORD dwStatus = WD_INVALID_HANDLE;

    hWD = WD_Open();
    if (hWD!=INVALID_HANDLE_VALUE)
    {
        BZERO(lic);
        // replace the following string with your license string
        strcpy(lic.cLicense, "12345abcde12345.CompanyName");
        dwStatus = WD_License(hWD, &lic);
        WD_Close(hWD);
    }

    return dwStatus;
}
```

A.7.10 WD_LogStart()

PURPOSE

- Opens a log file.

PROTOTYPE

```
DWORD WD_LogStart(const char *sFileName, const char *sMode)
```

PARAMETERS

Name	Type	Input/Output
➤ sFileName	const char *	Input
➤ sMode	const char *	Input

DESCRIPTION

Name	Description
sFileName	Name of log file to be opened.
sMode	Type of access permitted. For example, when NULL or w , opens an empty file for writing. If the given file exists, its contents are destroyed. When a , opens for writing at the end of the file (appending).

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

Once a log file is opened, all API calls are logged in this file. You may add your own printouts to the log file by calling WD_LogAdd() [A.7.12].

A.7.11 WD_LogStop()**PURPOSE**

- Closes a log file.

PROTOTYPE

```
VOID WD_LogStop()
```

RETURN VALUE

None

A.7.12 WD_LogAdd()

PURPOSE

- Adds user printouts into log file.

PROTOTYPE

```
VOID DLLCALLCONV WD_LogAdd(const char *sFormat[, argument ]...)
```

PARAMETERS

Name	Type	Input/Output
➤ sFormat	const char *	Input
➤ argument		Input

DESCRIPTION

Name	Description
sFormat	Format-control string
argument	Optional arguments

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.8 Kernel PlugIn - User-Mode Functions

The following functions are the user-mode functions that initiate the Kernel PlugIn operations, and activate its callbacks.

A.8.1 WD_KernelPlugInOpen()

PURPOSE

- Obtain a valid handle to the Kernel PlugIn.

PROTOTYPE

```
DWORD WD_KernelPlugInOpen (HANDLE hWD, WD_KERNEL_PLUGIN
    *pKernelPlugIn );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Output
> pKernelPlugIn	WD_KERNEL_PLUGIN *	
□ hKernelPlugIn	DWORD	Output
□ pcDriverName	PCHAR	Input
□ pcDriverPath	PCHAR	Input
□ pOpenData	PVOID	Input

DESCRIPTION

Name	Description
hWD	Handle to WinDriver
pKernelPlugIn	Pointer to WD_KERNEL_PLUGIN information
hKernelPlugIn	Returns the handle to the Kernel PlugIn
pcDriverName	Name of Kernel PlugIn to load, up to 8 chars
pcDriverPath	This field should be set to NULL. WinDriver will search for the driver in the operating system's drivers/modules directory.
pOpenData	Pointer to data that will be passed to the KP_Open () [A.9.2] callback in the Kernel PlugIn

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_KERNEL_PLUGIN kernelPlugIn;
BZERO(kernelPlugIn);

// Tells WinDriver which driver to open
kernelPlugIn.pcDriverName = "KPDriver";

HANDLE hWD = WD_Open(); // validate handle here

dwStatus = WD_KernelPlugInOpen(hWD, &kernelPlugIn);
if (dwStatus)
    printf ("Failed opening a handle to the Kernel PlugIn. Error: 0x%x (%s)\n",
           dwStatus, Stat2Str(dwStatus));
else
    printf("Opened a handle to the Kernel PlugIn (0x%x)\n",
           kernelPlugIn.hKernelPlugIn);
```

A.8.2 WD_KernelPlugInClose()

PURPOSE

- Closes the WinDriver Kernel PlugIn handle obtained from WD_KernelPlugInOpen() [A.8.1].

PROTOTYPE

```
DWORD WD_KernelPlugInClose(HANDLE hWD, WD_KERNEL_PLUGIN
    *pKernelPlugIn);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pKernelPlugIn	WD_KERNEL_PLUGIN *	Input

DESCRIPTION

Name	Description
hWD	Handle to WinDriver
pKernelPlugIn	Pointer to WD_KERNEL_PLUGIN information

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

EXAMPLE

```
WD_KernelPlugInClose(hWD, &kernelPlugIn);
```

A.8.3 WD_KernelPlugInCall()

PURPOSE

- Calls a routine in the Kernel PlugIn to be executed.

PROTOTYPE

```
DWORD WD_KernelPlugInCall( HANDLE hWD, WD_KERNEL_PLUGIN_CALL
    *pKernelPlugInCall );
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pKernelPlugInCall	WD_KERNEL_PLUGIN_CALL *	Input
□ hKernelPlugIn	DWORD	Input
□ dwMessage	DWORD	Input
□ pData	PVOID	Input
□ dwResult	DWORD	Output

DESCRIPTION

Name	Description
hWD	Handle to WinDriver
pKernelPlugInCall	Pointer to WD_KERNEL_PLUGIN_CALL information
hKernelPlugIn	Handle to the Kernel PlugIn
dwMessage	Message ID to pass to the KP_Call() [A.9.4] callback
pData	Pointer to data to pass to the KP_Call() callback
dwResult	Value set by KP_Call() callback

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

Calling the `WD_KernelPlugInCall()` [A.8.3] function in the user mode will call your `KP_Call()` [A.9.4] callback function in the kernel. The `KP_Call()` function in the Kernel PlugIn will determine what routine to execute according to the message passed to it in the `WD_KERNEL_PLUGIN_CALL` structure.

EXAMPLE

```
WD_KERNEL_PLUGIN_CALL kpCall;

BZERO (kpCall);
// Prepare the kpCall structure from WD_KernelPlugInOpen():
kpCall.hKernelPlugIn = hKernelPlugIn;
// Set the message to pass to KP_Call(). This will determine
// the action performed in the kernel:
kpCall.dwMessage = MY_DRV_MSG;

kpCall.pData = &mydrv; // The data to pass to the Kernel PlugIn.
dwStatus = WD_KernelPlugInCall(hWD, &kpCall);
if (dwStatus == WD_STATUS_SUCCESS)
    printf("Result = 0x%x\n", kpCall.dwResult);
else
    printf("WD_KernelPlugInCall() failed. Error: 0x%x (%s)\n",
          dwStatus, Stat2Str(dwStatus));
```

A.8.4 WD_IntEnable()

PURPOSE

- Enables interrupts for Kernel PlugIn

PROTOTYPE

```
DWORD WD_IntEnable(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> hWD	HANDLE	Input
> pInterrupt	WD_INTERRUPT *	
▣ kpCall	WD_KERNEL_PLUGIN_CALL	
◆ hKernelPlugIn	HANDLE	Input
◆ dwMessage	DWORD	N/A
◆ pData	PVOID	Input
◆ dwResult	DWORD	N/A

DESCRIPTION

Name	Description
hWD	Handle to WinDriver
pInterrupt	Pointer to WD_INTERRUPT information
hKernelPlugIn	Handle to the Kernel PlugIn. If zero, no Kernel PlugIn interrupt handler is installed
dwMessage	N/A
pData	Pointer to data to pass to the KP_IntEnable() callback
dwResult	N/A

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

REMARKS

If a valid handle to a Kernel PlugIn is passed to this function, the interrupts will be handled in the Kernel PlugIn. In such a case, the `KP_IntEnable()` callback will execute as a result of the call to `WD_IntEnable()` and upon receiving the interrupt, your kernel-mode `KP_IntAtIrql()` function [A.9.8] will execute. If this function returns a value greater than 0, your deferred procedure call, `KP_IntAtDpc()` [A.9.9], will be called.

For information about all other parameters of `WD_IntEnable()`, refer to the documentation of `WD_IntEnable()` in Section A.5.2.

EXAMPLE

```
WD_INTERRUPT Intrp;
BZERO(Intrp);
Intrp.hInterrupt = hInterrupt; // from WD_CardRegister()
// from WD_KernelPlugInOpen():
Intrp.kpCall.hKernelPlugIn = hKernelPlugIn;

WD_IntEnable(hWD, &Intrp);

if (!Intrp.fEnableOk)
    printf ("failed enabling interrupt\n");
```

A.9 Kernel PlugIn - Kernel-Mode Functions

The following functions are callback functions which are implemented in your Kernel PlugIn driver, and which will be called when their calling event occurs. For example: `KP_Init()` [A.9.1] is the callback function that is called when the driver is loaded. Any code that you want to execute upon loading should be in this function.

`KP_Init()` sets the name of the driver and the `KP_Open()` function.

`KP_Open()` sets the rest of the driver's callback functions.

For example:

```
kpOpenCall->funcClose = KP_Close;
kpOpenCall->funcCall = KP_Call;
kpOpenCall->funcIntEnable = KP_IntEnable;
kpOpenCall->funcIntDisable = KP_IntDisable;
kpOpenCall->funcIntAtIrql = KP_IntAtIrql;
kpOpenCall->funcIntAtDpc = KP_IntAtDpc;
kpOpenCall->funcEvent = KP_Event;
```

NOTE

It is the convention of this reference guide to mark the Kernel PlugIn callback functions as `KP_XXX()` - i.e. `KP_Open()`, `KP_Call()`, etc. However, you are free to select any name that you wish for your Kernel PlugIn callback functions, apart from `KP_Init()`, provided you implement relevant callback functions in your Kernel PlugIn. The generated DriverWizard Kernel PlugIn code, for example, uses the selected driver name in the callback function names (e.g. for a <MyKP> driver: `KP_MyKP_Open()`, `KP_MyKP_Call()`, etc.).

A.9.1 KP_Init()

PURPOSE

• Called when the Kernel PlugIn driver is loaded.
Sets the name of the Kernel PlugIn driver and the KP_Open () [A.9.2] callback function.

PROTOTYPE

```
BOOL __cdecl KP_Init(KP_INIT *kpInit);
```

PARAMETERS

Name	Type	Input/Output
> kpInit	KP_INIT *	
□ dwVerWD	DWORD	Output
□ cDriverName[12]	CHAR	Output
□ funcOpen	KP_FUNC_OPEN	Output

DESCRIPTION

Name	Description
kpInit	KP_INIT elements:
dwVerWD	The version of the WinDriver Kernel PlugIn library
cDriverName	The device driver name (up to 12 characters)
funcOpen	The KP_Open () callback function, which will be executed when WD_KernelPlugInOpen () is called

RETURN VALUE

TRUE if successful. Otherwise FALSE.

REMARKS

You must define the KP_Init () function in your code in order to link the Kernel PlugIn driver to WinDriver. KP_Init () is called when the driver is loaded. Any code that you want to execute upon loading should be in this function.

EXAMPLE

```
BOOL __cdecl KP_Init(KP_INIT *kpInit)
{
    // Check if the version of the WinDriver Kernel
    // PlugIn library is the same version
    // as windrvr.h and wd_kp.h
    if (kpInit->dwVerWD != WD_VER)
    {
        // You need to re-compile your Kernel PlugIn
        // with the compatible version of the WinDriver
        // Kernel PlugIn library, windrvr.h and wd_kp.h
        return FALSE;
    }

    kpInit->funcOpen = KP_Open;
    strcpy (kpInit->cDriverName, "KPDriver"); // up to 12 chars

    return TRUE;
}
```

A.9.2 KP_Open()

PURPOSE

• Called when `WD_KernelPlugInOpen()` [A.8.1] is called from user mode. Sets the rest of the Kernel PlugIn callback functions (`KP_Call()`, `KP_IntEnable()`, etc.) and performs any other desired initialization (such as allocating memory for the driver context and filling it with data passed from the user mode, etc.). The returned driver context (`pDrvContext`) will be passed to rest of the Kernel PlugIn callback functions.

PROTOTYPE

```
BOOL __cdecl KP_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
    PVOID pOpenData, PVOID *ppDrvContext);
```

PARAMETERS

Name	Type	Input/Output
➤ <code>kpOpenCall</code>	<code>KP_OPEN_CALL</code>	Input
➤ <code>hWD</code>	<code>HANDLE</code>	Input
➤ <code>pOpenData</code>	<code>PVOID</code>	Input
➤ <code>ppDrvContext</code>	<code>PVOID *</code>	Output

DESCRIPTION

Name	Description
<code>kpOpenCall</code>	Structure to fill in the addresses of the <code>KP_xxx()</code> callback functions
<code>hWD</code>	The WinDriver handle that <code>WD_KernelPlugInOpen()</code> [A.8.1] was called with
<code>pOpenData</code>	Pointer to data passed from user mode
<code>ppDrvContext</code>	Pointer to driver context data with which the <code>KP_Close()</code> [A.9.3], <code>KP_Call()</code> [A.9.4], <code>KP_IntEnable()</code> [A.9.6] and <code>KP_Event()</code> [A.9.5] functions will be called. Use this to keep driver specific information, which will be shared among these callbacks

RETURN VALUE

TRUE if successful. If FALSE, the call to WD_KernelPlugInOpen() from the user mode will fail.

EXAMPLE

```
BOOL __cdecl KP_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
                    PVOID pOpenData, PVOID *ppDrvContext)
{
    kpOpenCall->funcClose = KP_Close;
    kpOpenCall->funcCall = KP_Call;
    kpOpenCall->funcIntEnable = KP_IntEnable;
    kpOpenCall->funcIntDisable = KP_IntDisable;
    kpOpenCall->funcIntAtIrql = KP_IntAtIrql;
    kpOpenCall->funcIntAtDpc = KP_IntAtDpc;
    kpOpenCall->funcEvent = KP_Event;

    // You can allocate driver context memory here:
    *ppDrvContext = malloc(sizeof(MYDRV_STRUCT));
    return *ppDrvContext!=NULL;
}
```

A.9.3 KP_Close()

PURPOSE

- Called when WD_KernelPlugInClose() [A.8.2] is called from the user mode.
Can be used to perform any required clean-up for the Kernel PlugIn (such as freeing memory previously allocated for the driver context, etc.).

PROTOTYPE

```
void __cdecl KP_Close(PVOID pDrvContext);
```

PARAMETERS

Name	Type	Input/Output
> pDrvContext	PVOID	Input

DESCRIPTION

Name	Description
pDrvContext	Driver context data that was set by KP_Open() [A.9.2]

RETURN VALUE

None

EXAMPLE

```
void __cdecl KP_Close(PVOID pDrvContext)
{
    if (pDrvContext)
        free(pDrvContext); // Free allocated driver context memory
}
```

A.9.4 KP_Call()

PURPOSE

• Called when the user-mode application calls the `WD_KernelPlugInCall()` [A.8.3] function.

This function is a message handler for your utility functions.

PROTOTYPE

```
void __cdecl KP_Call(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL
    *kpCall, BOOL fIsKernelMode);
```

PARAMETERS

Name	Type	Input/Output
> pDrvContext	PVOID	Input/Output
> kpCall	WD_KERNEL_PLUGIN_CALL	
□ dwMessage	DWORD	Input
□ pData	PVOID	Input/Output
□ dwResult	DWORD	Output
> fIsKernelMode	BOOL	Input

DESCRIPTION

Name	Description
pDrvContext	Driver context data that was set by <code>KP_Open()</code> [A.9.2] and will also be passed to <code>KP_Close()</code> [A.9.3], <code>KP_IntEnable()</code> [A.9.6] and <code>KP_Event()</code> [A.9.5]
kpCall	Structure with user-mode information from <code>WD_KernelPlugInCall()</code> [A.8.3] and/or with information to return back to the user mode
dwMessage	Message ID passed from <code>WD_KernelPlugInCall()</code>
pData	Pointer to data passed from <code>WD_KernelPlugInCall()</code> and/or to data to return to the user mode
dwResult	Value to return to the user mode
fIsKernelMode	This parameter is passed by the WinDriver kernel (see remarks)

RETURN VALUE

None

REMARKS

- Calling the `WD_KernelPlugInCall()` [A.8.3] function in the user mode will call your `KP_Call()` [A.9.4] callback function in the kernel mode. The `KP_Call()` function in the Kernel PlugIn will determine which routine to execute according to the message passed to it in the `WD_KERNEL_PLUGIN_CALL` structure.
- The `flsKernelMode` parameter is passed by the WinDriver kernel to the `KP_Call()` routine. The user is not required to do anything about this parameter. However, notice how this parameter is passed in the sample code to the macro `COPY_TO_USER_OR_KERNEL` – This is required for the macro to function correctly. Please refer to section A.9.10 for more details regarding the `COPY_TO_USER_OR_KERNEL` and `COPY_FROM_USER_OR_KERNEL` macros.

EXAMPLE

```

void __cdecl KP_Call(PVOID pDrvContext,
    WD_KERNEL_PLUGIN_CALL *kpCall, BOOL fIsKernelMode)
{
    kpCall->dwResult = MY_DRV_OK;
    switch (kpCall->dwMessage)
    {
        // in this sample we implement a GetVersion message
        case MY_DRV_MSG_VERSION:
            {
                DWORD dwVer = 100;
                MY_DRV_VERSION *ver = (MY_DRV_VERSION *)kpCall->pData;
                COPY_TO_USER_OR_KERNEL(&ver->dwVer, &dwVer,
                    sizeof(DWORD), fIsKernelMode);
                COPY_TO_USER_OR_KERNEL(ver->cVer, "My Driver V1.00",
                    sizeof("My Driver V1.00")+1, fIsKernelMode);

                kpCall->dwResult = MY_DRV_OK;
            }
            break;
        // you can implement other messages here
        default:
            kpCall->dwResult = MY_DRV_NO_IMPL_MESSAGE;
    }
}

```


A.9.5 KP_Event()

PURPOSE

• Called when a Plug-and-Play or power management event for the device is received, provided the user-mode application first called `EventRegister()` [A.6.2] with a handle to the Kernel PlugIn (see Remarks).

PROTOTYPE

```
BOOL __cdecl KP_Event(PVOID pDrvContext, WD_EVENT *wd_event);
```

PARAMETERS

Name	Type	Input/Output
> pDrvContext	PVOID	Input/Output
> wd_event	WD_EVENT *	Input

DESCRIPTION

Name	Description
pDrvContext	Driver context data that was set by <code>KP_Open()</code> [A.9.2] and will also be passed to <code>KP_Close()</code> [A.9.3], <code>KP_IntEnable()</code> [A.9.6] and <code>KP_Call()</code> [A.9.4]
wd_event	Pointer to the PnP/power management event information received from the user mode

RETURN VALUE

TRUE in order to notify the user about the event.

REMARKS

`KP_Event()` will be called if the application called `EventRegister()` [A.6.2] with a handle to the Kernel PlugIn.

EXAMPLE

```
BOOL __cdecl KP_Event(PVOID pDrvContext, WD_EVENT *wd_event)
{
    // handle the event here
    return TRUE; // Return TRUE to notify the user about the event.
}
```

A.9.6 KP_IntEnable()

PURPOSE

• Called when `WD_IntEnable()` [A.5.2] is called from the user mode with a Kernel PlugIn handle.

The interrupt context (`pIntContext`) will be passed to the rest of the Kernel PlugIn interrupt functions.

PROTOTYPE

```
BOOL __cdecl KP_IntEnable (PVOID pDrvContext ,
                          WD_KERNEL_PLUGIN_CALL *kpCall , PVOID *ppIntContext );
```

PARAMETERS

Name	Type	Input/Output
> pDrvContext	PVOID	Input/Output
> kpCall	WD_KERNEL_PLUGIN_CALL	Input
□ dwMessage	DWORD	Input
□ pData	PVOID	Input/Output
□ dwResult	DWORD	Output
> ppIntContext	PVOID *	Input/Output

DESCRIPTION

Name	Description
pDrvContext	Driver context data that was set by <code>KP_Open()</code> [A.9.2] and will also be passed to <code>KP_Close()</code> [A.9.3], <code>KP_Call()</code> [A.9.4] and <code>KP_Event()</code> [A.9.5]
kpCall	Structure with information from <code>WD_IntEnable()</code> [A.5.2]
dwMessage	Message ID passed from <code>WD_IntEnable()</code>
pData	Pointer to data passed from <code>WD_IntEnable()</code> or to data to return to the user mode
dwResult	Value to return to the user mode
ppIntContext	Pointer to interrupt context data that will be passed to the <code>KP_IntDisable()</code> [A.9.7], <code>KP_IntAtIrql()</code> [A.9.8] and <code>KP_IntAtDpc()</code> [A.9.9] functions. Use this to keep interrupt specific information

RETURN VALUE

Returns TRUE if enable is successful.

REMARKS

This function should contain any initialization needed for your Kernel PlugIn interrupt handling.

EXAMPLE

```
BOOL __cdecl KP_IntEnable(PVOID pDrvContext,
    WD_KERNEL_PLUGIN_CALL *kpCall, PVOID *ppIntContext)
{
    DWORD *pIntCount;

    // You can allocate specific memory for each interrupt
    // in *ppIntContext
    *ppIntContext = malloc(sizeof (DWORD));
    if (!*ppIntContext)
        return FALSE;
    // In this sample the information is a DWORD used to
    // count the incoming interrupts
    pIntCount = (DWORD *) *ppIntContext;
    *pIntCount = 0; // reset the count to zero

    return TRUE;
}
```

A.9.7 KP_IntDisable()

PURPOSE

• Called when the user-mode application calls the WD_IntDisable() [A.5.5] function.
This function should free any memory that was allocated in KP_IntEnable() [A.9.6].

PROTOTYPE

```
void __cdecl KP_IntDisable(PVOID pIntContext);
```

PARAMETERS

Name	Type	Input/Output
> pIntContext	PVOID	Input

DESCRIPTION

Name	Description
pIntContext	Interrupt context data that was set by KP_IntEnable() [A.9.6]

RETURN VALUE

None

EXAMPLE

```
void __cdecl KP_IntDisable(PVOID pIntContext)
{
    // You can free the interrupt specific memory
    // allocated to pIntContext here
    free(pIntContext);
}
```

A.9.8 KP_IntAtIrql()

PURPOSE

- This function will run at high interrupt request level if the Kernel PlugIn handle is passed when enabling interrupts.

PROTOTYPE

```
BOOL __cdecl KP_IntAtIrql(PVOID pIntContext ,
    BOOL *pfIsMyInterrupt);
```

PARAMETERS

Name	Type	Input/Output
> pIntContext	PVOID	Input/Output
> pfIsMyInterrupt	BOOL *	Output

DESCRIPTION

Name	Description
pIntContext	Interrupt context data that was set by KP_IntEnable() [A.9.6] and will also be passed to KP_IntAtDpc() [A.9.9] (if executed) and KP_IntDisable() [A.9.7]
pfIsMyInterrupt	Set *pfIsMyInterrupt to TRUE if the interrupt belongs to this driver; otherwise set it to FALSE in order to enable the interrupt service routines of other drivers for the same interrupt to be called

RETURN VALUE

TRUE if deferred interrupt processing (DPC) is required; otherwise FALSE.

REMARKS

Code running at IRQL will only be interrupted by higher priority interrupts.

Code running at IRQL is limited by the following restrictions:

- You may only access non-pageable memory.

- You may only call the following functions: `WD_Transfer()` [A.4.14], `WD_MultiTransfer()` [A.4.15], `WD_DebugAdd()` [A.7.6] and specific OS kernel functions (such as Windows DDK functions) that are allowed to be called from an IRQL. Note that using OS-specific kernel functions may damage the code's portability to other operating systems. You may not call `malloc()`, `free()`, or any `WD_xxx()` API other than the aforementioned functions.

The code performed at high interrupt request level should be minimal (e.g., only the code that acknowledges level-sensitive interrupts), since it is operating at a high priority. The rest of your code should be written at `KP_IntAtDpc()` [A.9.9], which runs at the deferred DISPATCH level and is not subject to the above restrictions.

EXAMPLE

```

BOOL __cdecl KP_IntAtIrql(PVOID pIntContext,
    BOOL *pfIsMyInterrupt)
{
    DWORD *pdwIntCount = (DWORD *) pIntContext;

    /* Check your hardware here to see if the interrupt belongs to you.
       If it does, you must set *pfIsMyInterrupt to TRUE.
       Otherwise, set *pfIsMyInterrupt to FALSE. */
    *pfIsMyInterrupt = FALSE;

    /* In this example we will schedule a DPC
       once in every 5 interrupts */
    (*pdwIntCount)++;
    if (*pdwIntCount==5)
    {
        *pdwIntCount = 0;
        return TRUE;
    }

    return FALSE;
}

```

A.9.9 KP_IntAtDpc()

PURPOSE

- This is the Deferred Procedure Call which is executed only if the KP_IntAtIrql() [A.9.8] function returned TRUE.

PROTOTYPE

```
DWORD __cdecl KP_IntAtDpc(PVOID pIntContext , DWORD dwCount);
```

PARAMETERS

Name	Type	Input/Output
> pIntContext	PVOID	Input/Output
> dwCount	DWORD	Input

DESCRIPTION

Name	Description
pIntContext	Interrupt context data that was set by KP_IntEnable() [A.9.6], passed to KP_IntAtIrql() [A.9.8] and will be passed to KP_IntDisable() [A.9.7]
dwCount	The number of times KP_IntAtIrql() [A.9.8] returned TRUE since the last DPC call. If dwCount is 1, KP_IntAtIrql() requested a DPC only once since the last DPC call. If the value is greater than 1, KP_IntAtIrql() has already requested a DPC a few times, but the interval was too short, therefore KP_IntAtDpc() was not called for each DPC request.

RETURN VALUE

Returns the number of times to notify user mode (i.e., return from WD_IntWait() [A.5.3]).

REMARKS

Most of the interrupt handling should be written at DPC.

If `KP_IntAtDpc()` returns with a value of 1 or more, `WD_IntWait()` returns and the user-mode interrupt handler will execute in the number of times set in the return value. If you do not want the user-mode interrupt handler to execute, `KP_IntAtDpc()` should return 0.

EXAMPLE

```
DWORD __cdecl KP_IntAtDpc(PVOID pIntContext, DWORD dwCount)
{
    // Return WD_IntWait as many times as KP_IntAtIrql
    // scheduled KP_IntAtDpc()
    return dwCount;
}
```

A.9.10 COPY_TO_USER_OR_KERNEL, COPY_FROM_USER_OR_KERNEL

PURPOSE

- Macros for copying data from the user mode to the Kernel PlugIn and vice versa.

REMARKS

The `COPY_TO_USER_OR_KERNEL` and `COPY_FROM_USER_OR_KERNEL` are macros used for copying data (when necessary) to/from user-mode memory addresses (respectively), when accessing such addresses from within the Kernel PlugIn. Copying the data ensures that the user-mode address can be used correctly, even if the context of the user-mode process changes in the midst of the I/O operation. This is particularly relevant for long operations, during which the context of the user-mode process may change. The use of macros to perform the copy provides a generic solution for all supported operating systems.

Please note that if you wish to access the user-mode data from within the `KP_IntAtIrql()` [A.9.8] or `KP_IntAtDpc()` [A.9.9] functions, you should first copy the data into some variable in the Kernel PlugIn before the execution of these routines.

The `COPY_TO_USER_OR_KERNEL` and `COPY_FROM_USER_OR_KERNEL` macros are defined in the **WinDriver\include\kpstdlib.h** header file.

For an example of using the `COPY_TO_USER_OR_KERNEL` macro, see the `KP_Call()` [A.9.4] implementation (`KP_PCI_Call()`) in the sample **WinDriver/samples/pci_diag/kp_pci/kp_pci.c** Kernel PlugIn file.

To share a data buffer between the user-mode and Kernel PlugIn routines (e.g., `KP_IntAtIrql()` [A.9.8] and `KP_IntAtDpc()` [A.9.9]) safely, consider using the technique outlined in the technical document titled "How do I share a memory buffer between Kernel PlugIn and user-mode projects for DMA or other purposes?" found under the "Kernel PlugIn" technical documents section of the "Support" section.

A.9.11 Kernel PlugIn Synchronization APIs

This section describes the Kernel Plug-In synchronization APIs.

These APIs support the following synchronization mechanisms: .

- Spinlocks [A.9.11.2 – A.9.11.5], which are used to synchronize between threads on a single or multiple CPU system.

NOTE

The Kernel PlugIn spinlock functions can be called from any context apart from high interrupt request level. Hence they can be called from any Kernel PlugIn function **except** for `KP_IntAtIrql()` [A.9.8]. Note that the spinlock functions **can** be called from `KP_IntAtDpc()` [A.9.9].

- Interlocked operations [A.9.11.6 – A.9.11.7], which are used for synchronizing access to a variable that is shared by multiple threads by performing complex operations on the variable in an atomic manner.

NOTE

The Kernel PlugIn interlocked functions can be called from any context in the Kernel PlugIn, including from high interrupt request level. Hence they can be called from any Kernel PlugIn function, **including** `KP_IntAtIrql()` [A.9.8] and `KP_IntAtDpc()` [A.9.9].

A.9.11.1 Kernel PlugIn Synchronization Types

The Kernel PlugIn synchronization APIs use the following types:

- **KP_SPINLOCK** – A Kernel PlugIn spinlock object structure:

```
typedef struct _KP_SPINLOCK KP_SPINLOCK;
```

`_KP_SPINLOCK` is an internal WinDriver spinlock object structure, opaque to the user.

- **KP_INTERLOCKED** – a Kernel PlugIn interlocked operations counter:

```
typedef volatile int KP_INTERLOCKED;
```

A.9.11.2 kp_spinlock_init()**PURPOSE**

- Initializes a new Kernel PlugIn spinlock object.

PROTOTYPE

```
KP_SPINLOCK * kp_spinlock_init(void);
```

RETURN VALUE

If successful, returns a pointer to the new Kernel PlugIn spinlock object [A.9.11.1], otherwise returns NULL.

A.9.11.3 kp_spinlock_wait()**PURPOSE**

- Waits on a Kernel PlugIn spinlock object.

PROTOTYPE

```
void kp_spinlock_wait(KP_SPINLOCK *spinlock);
```

PARAMETERS

Name	Type	Input/Output
➤ spinlock	KP_SPINLOCK*	Input

DESCRIPTION

Name	Description
➤ spinlock	Pointer to the Kernel PlugIn spinlock object [A.9.11.1] on which to wait

RETURN VALUE

None

A.9.11.4 kp_spinlock_release()**PURPOSE**

- Releases a Kernel PlugIn spinlock object.

PROTOTYPE

```
void kp_spinlock_release (KP_SPINLOCK *spinlock);
```

PARAMETERS

Name	Type	Input/Output
➤ spinlock	KP_SPINLOCK*	Input

DESCRIPTION

Name	Description
➤ spinlock	Pointer to the Kernel PlugIn spinlock object [A.9.11.1] to release

RETURN VALUE

None

A.9.11.5 kp_spinlock_uninit()**PURPOSE**

- Un-initializes a Kernel PlugIn spinlock object.

PROTOTYPE

```
void kp_spinlock_uninit (KP_SPINLOCK *spinlock);
```

PARAMETERS

Name	Type	Input/Output
➤ spinlock	KP_SPINLOCK*	Input

DESCRIPTION

Name	Description
➤ spinlock	Pointer to the Kernel PlugIn spinlock object [A.9.11.1] to un-initialize

RETURN VALUE

None

A.9.11.6 kp_interlocked_init()**PURPOSE**

- Initializes a Kernel PlugIn interlocked counter.

PROTOTYPE

```
void kp_interlocked_init(KP_INTERLOCKED *target);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to initialize

RETURN VALUE

None

A.9.11.7 kp_interlocked_uninit()**PURPOSE**

- Un-initializes a Kernel PlugIn interlocked counter.

PROTOTYPE

```
void kp_interlocked_uninit(KP_INTERLOCKED *target);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to un-initialize

RETURN VALUE

None

A.9.11.8 kp_interlocked_increment()**PURPOSE**

- Increments the value of a Kernel PlugIn interlocked counter by one.

PROTOTYPE

```
int kp_interlocked_increment(KP_INTERLOCKED *target);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to increment

RETURN VALUE

Returns the new value of the interlocked counter (target).

A.9.11.9 kp_interlocked_decrement()**PURPOSE**

- Decrements the value of a Kernel PlugIn interlocked counter by one.

PROTOTYPE

```
int kp_interlocked_decrement (KP_INTERLOCKED *target);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to decrement

RETURN VALUE

Returns the new value of the interlocked counter (target).

A.9.11.10 kp_interlocked_add()**PURPOSE**

- Adds a specified value to the current value of a Kernel PlugIn interlocked counter.

PROTOTYPE

```
int kp_interlocked_add (KP_INTERLOCKED *target , int val);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output
➤ val	val	Input

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to which to add
➤ val	The value to add to the interlocked counter (target)

RETURN VALUE

Returns the new value of the interlocked counter (target).

A.9.11.11 kp_interlocked_read()**PURPOSE**

- Reads to the value of a Kernel PlugIn interlocked counter.

PROTOTYPE

```
int kp_interlocked_read(KP_INTERLOCKED *target);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to read

RETURN VALUE

Returns the value of the interlocked counter (target).

A.9.11.12 kp_interlocked_set()**PURPOSE**

- Sets the value of a Kernel PlugIn interlocked counter to the specified value.

PROTOTYPE

```
void kp_interlocked_set(KP_INTERLOCKED *target , int val);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output
➤ val	val	Input

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to set
➤ val	The value to set for the interlocked counter (target)

RETURN VALUE

None

A.9.11.13 kp_interlocked_exchange()**PURPOSE**

- Sets the value of a Kernel PlugIn interlocked counter to the specified value and returns the previous value of the counter.

PROTOTYPE

```
int kp_interlocked_exchange (KP_INTERLOCKED *target , int val);
```

PARAMETERS

Name	Type	Input/Output
➤ target	KP_INTERLOCKED*	Input/Output
➤ val	val	Input

DESCRIPTION

Name	Description
➤ target	Pointer to the Kernel PlugIn interlocked counter [A.9.11.1] to exchange
➤ val	The new value to set for the interlocked counter (target)

RETURN VALUE

Returns the previous value of the interlocked counter (target).

A.10 Kernel PlugIn - Structure Reference

This section contains detailed information about the different Kernel PlugIn related structures. **WD_XXX** structures are used in user-mode functions and **KP_XXX** structures are used in kernel-mode functions.

The Kernel PlugIn synchronization types are documented in section [A.9.11.1](#).

A.10.1 WD_KERNEL_PLUGIN

Defines a Kernel PlugIn open command.

This structure is used by `WD_KernelPlugInOpen()` [[A.8.1](#)] and `WD_KernelPlugInClose()` [[A.8.2](#)].

Members:

Name	Type	Description
hKernelPlugIn	DWORD	Handle to a Kernel PlugIn
pcDriverName	PCHAR	Name of Kernel PlugIn driver. Should be no longer than 12 characters. Should not include the VXD or SYS extension.
pcDriverPath	PCHAR	This field should be set to NULL. WinDriver will search for the driver in the operating system's drivers/modules directory.
pOpenData	PVOID	Data to pass to the <code>KP_Open()</code> [A.9.2] callback in the Kernel PlugIn.

A.10.2 WD_INTERRUPT

Used to describe an interrupt.

This structure is used by the following functions: `InterruptEnable()` [A.4.21], `InterruptDisable()` [A.4.22], `WD_IntEnable()` [A.5.2], `WD_IntDisable()` [A.5.5], `WD_IntWait()` [A.5.3], `WD_IntCount()` [A.5.4].

Members:

Name	Type	Description
<code>kpCall</code>	<code>WD_KERNEL_PLUGIN_CALL</code> [A.10.3]	The <code>kpCall</code> structure contains the handle to the Kernel PlugIn and to other information that should be passed to the kernel-mode interrupt handler when installing it. If the handle is zero, the interrupt is installed without a Kernel PlugIn interrupt handler. If a valid Kernel PlugIn handle is set, this structure will be passed as a parameter to the <code>KP_IntEnable()</code> [A.9.6] Kernel PlugIn callback function.

For information about the other members of `WD_INTERRUPT`, see Section A.4.21.

A.10.3 WD_KERNEL_PLUGIN_CALL

Contains information that will be passed to the Kernel PlugIn. This structure is used when passing messages to the Kernel PlugIn or when installing a Kernel PlugIn interrupt.

This structure is used by `WD_KernelPlugInCall()` [A.8.3], `InterruptEnable()` [A.4.21] and `WD_IntEnable()` [A.5.2], and is passed as a parameter to the Kernel PlugIn `KP_Call()` [A.9.4] and `KP_IntEnable()` [A.9.6] callback functions.

Members:

Name	Type	Description
<code>hKernelPlugIn</code>	DWORD	Handle to a Kernel PlugIn.
<code>dwMessage</code>	DWORD	Message ID to pass to a Kernel PlugIn callback.
<code>pData</code>	PVOID	Pointer to data to pass to Kernel PlugIn callback.
<code>dwResult</code>	DWORD	Value set by a Kernel PlugIn callback, to return back to user mode.

A.10.4 KP_INIT

This structure is used by the `KP_Init()` [A.9.1] function in the Kernel PlugIn. Its primary use is for notifying WinDriver of the given driver's name and of which kernel-mode function to call when `WD_KernelPlugInOpen()` [A.8.1] is called from the user mode.

MEMBERS:

Name	Type	Description
dwVerWD	DWORD	The version of the WinDriver Kernel PlugIn library.
cDriverName	CHAR[12]	The device driver name, up to 12 characters.
funcOpen	KP_FUNC_OPEN	The <code>KP_Open()</code> [A.9.2] kernel-mode function that WinDriver should call when <code>WD_KernelPlugInOpen()</code> [A.8.1] is called from the user mode.

A.10.5 KP_OPEN_CALL

This is the structure through which the Kernel PlugIn defines the names of the callbacks which it implements. It is used in the `KP_Open()` [A.9.2] Kernel PlugIn function.

A Kernel PlugIn may implement 7 different callback functions:

funcClose – Called when the application is done with this instance of the driver.

funcCall – Called when the application calls `WD_KernelPlugInCall()` [A.8.3]. This function is a general purpose function. In it, implement any functions that should run in kernel mode (except the interrupt handler which is a special case). The `funcCall` will determine which function to execute according to the message passed to it.

funcIntEnable – Called when the application calls `WD_IntEnable()` [A.5.2] / `InterruptEnable()` [A.4.21] with a Kernel PlugIn handle. This callback function should perform any initialization required when enabling an interrupt.

funcIntDisable – The cleanup function, which is called when the application calls `WD_IntDisable()` [A.5.5] / `InterruptDisable()` [A.4.22].

funcIntAtIrql – This is the kernel mode interrupt handler. This callback function is called at high interrupt request level when WinDriver processes the interrupt that is assigned to this Kernel PlugIn. If this function returns a value greater than 0, `funcIntAtDpc()` is called as a Deferred procedure call.

funcIntAtDpc – Most of your interrupt handler code should be written in this callback. It is called as a deferred procedure call, if `funcIntAtIrql()` returns a value greater than 0.

funcEvent – Called when a Plug-and-Play or power management event occurs, if the application first called `EventRegister()` [A.6.2] with a Kernel PlugIn handle. This callback function should implement the desired kernel handling for Plug-and-Play and power management events.

MEMBERS:

Name	Type	Description
funcClose	KP_FUNC_CLOSE	Name of your KP_Close() [A.9.3] function in the kernel.
funcCall	KP_FUNC_CALL	Name of your KP_Call() [A.9.4] function in the kernel.
funcIntEnable	KP_FUNC_INT_ENABLE	Name of your KP_IntEnable() [A.9.6] function in the kernel.
funcIntDisable	KP_FUNC_INT_DISABLE	Name of your KP_IntDisable() [A.9.7] function in the kernel.
funcIntAtIrql	KP_FUNC_INT_AT_IRQL	Name of your KP_IntAtIrql() [A.9.8] function in the kernel.
funcIntAtDpc	KP_FUNC_INT_AT_DPC	Name of your KP_IntAtDpc() [A.9.9] function in the kernel.
funcEvent	KP_FUNC_EVENT	Name of your KP_Event() [A.9.5] function in the kernel.

A.11 WinDriver Status/Error Codes

A.11.1 Introduction

Most of the WinDriver API functions return a status code, where 0 (WD_STATUS_SUCCESS) means success and a non-zero value means failure. The Stat2Str() and WDL_Stat2Str() can be used to retrieve the status description string for a given status code. The status codes and their descriptive strings are listed below.

A.11.2 Status Codes Returned by WinDriver

Status Code	Description
WD_STATUS_SUCCESS	Success
WD_STATUS_INVALID_WD_HANDLE	Invalid WinDriver handle
WD_WINDRIVER_STATUS_ERROR	Error
WD_INVALID_HANDLE	Invalid handle
WD_INVALID_PIPE_NUMBER	Invalid pipe number
WD_READ_WRITE_CONFLICT	Conflict between read and write operations
WD_ZERO_PACKET_SIZE	Packet size is zero
WD_INSUFFICIENT_RESOURCES	Insufficient resources
WD_UNKNOWN_PIPE_TYPE	Unknown pipe type
WD_SYSTEM_INTERNAL_ERROR	Internal system error
WD_DATA_MISMATCH	Data mismatch
WD_NO_LICENSE	No valid license
WD_NOT_IMPLEMENTED	Function not implemented
WD_KERPLUG_FAILURE	Kernel PlugIn failure
WD_FAILED_ENABLING_INTERRUPT	Failed enabling interrupt
WD_INTERRUPT_NOT_ENABLED	Interrupt not enabled
WD_RESOURCE_OVERLAP	Resource overlap
WD_DEVICE_NOT_FOUND	Device not found
WD_WRONG_UNIQUE_ID	Wrong unique ID
WD_OPERATION_ALREADY_DONE	Operation already done
WD_SET_CONFIGURATION_FAILED	Set configuration operation failed
WD_CANT_OBTAIN_PDO	Cannot obtain PDO
WD_TIME_OUT_EXPIRED	Timeout expired
WD_IRP_CANCELED	IRP operation cancelled
WD_FAILED_USER_MAPPING	Failed to map in user space
WD_FAILED_KERNEL_MAPPING	Failed to map in kernel space

Status Code	Description
WD_NO_RESOURCES_ON_DEVICE	No resources on the device
WD_NO_EVENTS	No events
WD_INVALID_PARAMETER	Invalid parameter
WD_INCORRECT_VERSION	Incorrect WinDriver version installed
WD_TRY_AGAIN	Try again
WD_INVALID_IOCTL	Received an invalid IOCTL

A.12 User-Mode Utility Functions

This section describes a number of user-mode utility functions you will find useful for implementing various tasks. These utility functions are multi-platform, implemented on all operating systems supported by WinDriver.

A.12.1 Stat2Str()

PURPOSE

- Retrieves the status string that corresponds to a status code.

PROTOTYPE

```
const char * Stat2Str(DWORD dwStatus);
```

PARAMETERS

Name	Type	Input/Output
> dwStatus	DWORD	Input

DESCRIPTION

Name	Description
dwStatus	A numeric status code

RETURN VALUE

Returns the verbal status description (string) that corresponds to the specified numeric status code.

REMARKS

See Section [A.11](#) for a complete list of status codes and strings.

A.12.2 get_os_type()**PURPOSE**

- Retrieves the type of the operating system.

PROTOTYPE

```
OS_TYPE get_os_type();
```

RETURN VALUE

NoneReturns the type of the operating system.

If the operating system type is not detected, returns OS_CAN_NOT_DETECT.

A.12.3 ThreadStart()

PURPOSE

- Creates a thread.

PROTOTYPE

```
DWORD ThreadStart(HANDLE *phThread, HANDLER_FUNC pFunc, void *pData);
```

PARAMETERS

Name	Type	Input/Output
➤ phThread	HANDLE *	Output
➤ pFunc	HANDLER_FUNC	Input
➤ pData	VOID *	Input

DESCRIPTION

Name	Description
phThread	Returns the handle to the created thread
pFunc	Starting address of the code that the new thread is to execute
pData	Pointer to the data to be passed to the new thread

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.12.4 ThreadWait()

PURPOSE

- Waits for a thread to exit.

PROTOTYPE

```
void ThreadWait(HANDLE hThread);
```

PARAMETERS

Name	Type	Input/Output
> hThread	HANDLE	Input

DESCRIPTION

Name	Description
hThread	The handle to the thread whose completion is awaited

RETURN VALUE

None

A.12.5 OsEventCreate()

PURPOSE

- Creates an event object.

PROTOTYPE

```
DWORD OsEventCreate(HANDLE *phOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➤ phOsEvent	HANDLE *	Output

DESCRIPTION

Name	Description
phOsEvent	The pointer to a variable that receives a handle to the newly created event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.12.6 OsEventClose()

PURPOSE

- Closes a handle to an event object.

PROTOTYPE

```
void OsEventClose(HANDLE hOsEvent)
```

PARAMETERS

Name	Type	Input/Output
➤ hOsEvent	HANDLE	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object to be closed

RETURN VALUE

None

A.12.7 OsEventWait()

PURPOSE

- Waits until a specified event object is in the signaled state or the time-out interval elapses.

PROTOTYPE

```
DWORD OsEventWait(HANDLE hOsEvent, DWORD dwSecTimeout)
```

PARAMETERS

Name	Type	Input/Output
➤ hOsEvent	HANDLE	Input
➤ dwSecTimeout	DWORD	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object
dwSecTimeout	Time-out interval of the event, in seconds. If dwSecTimeout is INFINITE, the function's time-out interval never elapses.

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.12.8 OsEventSignal()

PURPOSE

- Sets the specified event object to the signaled state.

PROTOTYPE

```
DWORD OsEventSignal(HANDLE hOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➤ hOsEvent	HANDLE	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.12.9 OsEventReset()

PURPOSE

- Resets the specified event object to the non-signaled state.

PROTOTYPE

```
DWORD OsEventReset(HANDLE hOsEvent);
```

PARAMETERS

Name	Type	Input/Output
➤ hOsEvent	HANDLE	Input

DESCRIPTION

Name	Description
hOsEvent	The handle to the event object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A.11].

A.12.10 OsMutexCreate()**PURPOSE**

- Creates a mutex object.

PROTOTYPE

```
DWORD OsMutexCreate(HANDLE *phOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➤ phOsMutex	HANDLE *	Output

DESCRIPTION

Name	Description
phOsMutex	The pointer to a variable that receives a handle to the newly created mutex object

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.12.11 OsMutexClose()**PURPOSE**

- Closes a handle to a mutex object.

PROTOTYPE

```
void OsMutexClose(HANDLE hOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➤ hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be closed

RETURN VALUE

None

A.12.12 OsMutexLock()**PURPOSE**

- Locks the specified mutex object.

PROTOTYPE

```
DWORD OsMutexLock(HANDLE hOsMutex)
```

PARAMETERS

Name	Type	Input/Output
➤ hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be locked

RETURN VALUE

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A.11].

A.12.13 OsMutexUnlock()**PURPOSE**

- Releases (unlocks) a locked mutex object.

PROTOTYPE

```
DWORD OsMutexUnlock(HANDLE hOsMutex);
```

PARAMETERS

Name	Type	Input/Output
➤ hOsMutex	HANDLE	Input

DESCRIPTION

Name	Description
hOsMutex	The handle to the mutex object to be unlocked

RETURN VALUE

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [\[A.11\]](#).

A.12.14 PrintDbgMessage()

PURPOSE

- Sends debug messages to the debug monitor.

PROTOTYPE

```
void PrintDbgMessage(DWORD dwLevel, DWORD dwSection,  
    const char *format[, argument]...);
```

PARAMETERS

Name	Type	Input/Output
➤ dwLevel	DWORD	Input
➤ dwSection	DWORD	Input
➤ format	const char *	Input
➤ argument		Input

DESCRIPTION

Name	Description
dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If dwLevel is 0, then D_ERROR will be declared. For more details please refer to DEBUG_LEVEL in windrvr.h .
dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If dwSection is 0, then S_MISC section will be declared. For more details please refer to DEBUG_SECTION in windrvr.h .
format	Format-control string
argument	Optional arguments, limited to 256 bytes

RETURN VALUE

None

Appendix B

Troubleshooting and Support

Please refer to <http://www.jungo.com/support> for addition resources for developers, including:

- Technical documents
- FAQs
- Samples
- Quick start guides

Appendix C

Evaluation Version Limitations

C.1 Windows 98/Me/2000/XP/Server 2003

- Each time WinDriver is activated, an **Unregistered** message appears.
- When using the DriverWizard, a dialog box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- WinDriver will function for only 30 days after the original installation.

C.2 Windows CE

- Each time WinDriver is activated, an **Unregistered** message appears.
- The WinDriver CE Kernel (**windrvr6.dll**) will operate for no more than 60 minutes at a time.
- WinDriver CE emulation on Windows 2000/XP/Server 2003 will stop working after 30 days.

C.3 Linux

- Each time WinDriver is activated, an **Unregistered** message appears.
- When using the DriverWizard, a dialog box with a message stating that an evaluation version is being run appears on every interaction with the hardware.

- WinDriver's kernel module will work for no more than 60 minutes at a time. In order to continue working, the WinDriver kernel module must be reloaded (remove and insert the module) using the following commands:

To remove:

```
/sbin/rmmmod windrvr6
```

To insert:

```
/sbin/modprobe windrvr6
```

C.4 Solaris

- Each time WinDriver is activated, an **Unregistered** message appears.
- When using the DriverWizard, a dialog box with a message stating that an evaluation version is being run appears on every interaction with the hardware.
- The Solaris kernel will work for no more than 60 minutes at a time. In order to continue working, WinDriver kernel module must be reloaded (remove and insert the module) using the following commands:

To remove:

```
/usr/sbin/rem_drv windrvr6
```

To insert:

```
/usr/sbin/add_drv windrvr6
```

C.5 VxWorks

- The VxWorks Kernel will work for no more than 60 minutes at a time. In order to continue working, the system must be rebooted.

C.6 DriverWizard GUI

- Each time WinDriver is activated, an **Unregistered** message appears.
- When using the DriverWizard, a dialog box with a message stating that an evaluation version is being run appears on every interaction with the hardware.

Appendix D

Purchasing WinDriver

Fill in the order form found in **Start | WinDriver | Order Form** on your Windows start menu, and send it to Jungo via email, fax or mail (see details below).

Your WinDriver package will be sent to you via Fedex or standard postal mail. The WinDriver license string will be emailed to you immediately.

E M A I L

Support: support@jungo.com

Sales: sales@jungo.com

P H O N E / F A X

Phone:

USA (Toll-Free): 1-877-514-0537

Worldwide: +972-9-8859365

Fax:

USA (Toll-Free): 1-877-514-0538

Worldwide: +972-9-8859366

W E B:

<http://www.jungo.com>

P O S T A L A D D R E S S

Jungo Ltd.
P.O.Box 8493
Netanya 42504
ISRAEL

Appendix E

Distributing Your Driver – Legal Issues

*WinDriver is licensed per-seat. The WinDriver license allows one developer on a single computer to develop an unlimited number of device drivers, and to freely distribute the created drivers without royalties, as outlined in the license agreement in the **WinDriver/docs/license.txt** file.*

Appendix F

Additional Documentation

Updated Manual

The most updated WinDriver User's manual can be found on Jungo's site at:
<http://www.jungo.com/support/manuals.html#manuals>

Version History

If you wish to view WinDriver version history, please refer to
<http://www.jungo.com/wdver.html>. Here you will be able to view a list of all new features, enhancements and fixes which have been added in each WinDriver version.

Technical Documents

For additional information, you may refer to the Technical Documents database on our site at:

http://www.jungo.com/support/tech_docs_indexes/main_index.html.

The Technical Documents database includes detailed descriptions of WinDriver's features, utilities and APIs and their correct usage, troubleshooting of common problems, useful tips and answers to frequently asked questions.