



WinDriver 15.1.1 User Manual

© 2022 Jungo Connectivity Ltd.

1 Overview	1
1.1. Introduction	1
1.2. Main Features	2
1.3. Architecture	2
1.4. Supported Platforms	4
1.5. Linux ARM	5
1.6. Operating Systems Support Overview	5
1.7. Limitations of Different Evaluation Versions	6
1.7.1. Limitations on MacOS	7
1.8. What WinDriver includes	7
2 Understanding Device Drivers	9
2.1. Device Driver Overview	9
2.2. Classification of Drivers According to Functionality	9
2.2.1. Monolithic Drivers	9
2.2.2. Layered Drivers	10
2.2.3. Miniport Drivers	11
2.3. Classification of Drivers According to Operating Systems	12
2.3.1. WDM Drivers	12
2.3.2. WDF Drivers	12
2.3.3. Unix Device Drivers	13
2.3.4. Linux Device Drivers	13
2.4. The Entry Point of the Driver	13
2.5. Associating the Hardware with the Driver	13
2.6. Communicating with Drivers	14
3 Installing WinDriver	15
3.1. System Requirements	15
3.1.1. Windows System Requirements	15
3.1.2. Linux System Requirements	15
3.1.3. MacOS System Requirements	16
3.2. WinDriver Installation Process	16
3.2.1. Windows WinDriver Installation Instructions	16
3.2.2. Linux WinDriver Installation Instructions	17
3.2.2.1. Preparing the System for Installation	17
3.2.2.2. Installing WinDriver on x86/x86_64 systems	17
3.2.2.3. Installing WinDriver on ARM/ARM64 systems	19
3.2.2.4. Restricting Hardware Access on Linux	20
3.2.3. MacOS WinDriver Installation Instructions	20
3.2.3.1 Preparing the System for Installation	20
3.2.3.2 Installation on MacOS x86_64	21
3.2.3.3 Installation on MacOS ARM64 (M1)	21
3.3. Upgrading Your Installation	21

3.4. Checking Your Installation	22
3.4.1. Installation Check	22
3.5. Uninstalling WinDriver	22
3.5.1. Windows WinDriver Uninstall Instructions	22
3.5.2. Linux WinDriver Uninstall Instructions	23
3.5.3. MacOS WinDriver Uninstall Instructions	24
4 PCI Express Overview	25
4.1. Overview	25
4.2. WinDriver for PCI Express	25
4.3. The pci_dump and pci_scan Utilities	26
4.3.1. Dump PCI Configuration Space into a file	26
4.4. FAQ	26
4.4.1. Enabling legacy PCI configuration space read/write for identifying PCI devices on Windows	26
4.4.2. How do I access the memory on my PCI card using WinDriver?	27
4.4.3. How do I use WinDriver for PCI Express over Thunderbolt?	27
5 USB Overview	29
5.1. Introduction to USB	29
5.2. WinDriver USB Benefits	29
5.3. USB Components	30
5.4. Data Flow in USB Devices	31
5.5. USB Data Exchange	32
5.6. USB Data Transfer Types	32
5.6.1. Control Transfer	32
5.6.2. Isochronous Transfer	33
5.6.3. Interrupt Transfer	33
5.6.4. Bulk Transfer	33
5.7. USB Configuration	33
5.8. WinDriver USB	35
5.9. WinDriver USB Architecture	35
5.10. WinDriver USB (WDU) Library Overview	36
5.10.1. Calling Sequence for WinDriver USB	36
5.10.2. Upgrading from the WD_xxx USB API to the WDU_xxx API	38
5.11. FAQ	38
5.11.1 How do I reset my USB device using WinDriver?	38
6 Using DriverWizard	39
6.1. An Overview	39
6.2. DriverWizard Walkthrough	39
6.2.1. Attach your hardware to the computer	39
6.2.2. Run DriverWizard and select your device	40
6.2.3. Generate and install an INF file for your device (Windows)	42

6.2.4. Uninstall the INF file of your device (Windows)	44
6.2.5. Select the desired alternate setting (USB)	44
6.2.6. Diagnose your device (PCI)	44
6.2.7. Diagnose your device (USB)	46
6.2.8. Import Register Information from CSV file	48
6.2.9. Samples And Automatic Code Generation	49
6.2.9.1. Samples Or Code Generation	49
6.2.9.2. The Generated PCI/ISA and USB C Code	51
6.2.9.3. The Generated PCI/ISA Python Code	52
6.2.9.4. The Generated USB Python Code	52
6.2.9.5. The Generated PCI/ISA Java Code	52
6.2.9.6. The Generated USB Java Code	53
6.2.9.7. The Generated PCI/ISA and USB .NET Code	53
6.3. Compiling the Generated Code	54
6.3.1. C/C#/VB Windows Compilation	54
6.3.1.1. C Compilation with a Linux Makefile	54
6.3.1.2. C Compilation with CMake	54
6.3.1.2.1 Generating a WinDriver C project for Xcode (MacOS)	55
6.3.1.2.2 Generating a WinDriver C project for MinGW (Windows)	55
6.3.2. Java Compilation	55
6.3.2.1. Opening the DriverWizard generated Java code with Eclipse IDE	55
6.3.2.2. Compiling and running the DriverWizard generated Java code from the command line	55
6.4. FAQ	56
6.4.1. If a variable requires a pointer to be assigned to it, as in pBuffer = &dwVal, how do I do it in C# dotNET/Java/Python?	56
7 Developing a Driver	57
7.1. Using DriverWizard to Build a Device Driver	57
7.2. Using a code sample to Build a Device Driver	57
7.3. Writing the Device Driver Without DriverWizard	57
7.3.1. Include the Required WinDriver Files	58
7.3.2. Write Your Code (PCI/ISA)	58
7.3.3. Write Your Code (USB)	59
7.3.4. Configure and Build Your Code	59
7.4. Upgrading a Driver	60
7.4.1. Regenerating code and gradually merging	60
7.4.2. Checklist for Driver Code upgrade	60
7.4.2.1 Register Your New License	60
7.4.2.2 Conform to API Updates	60
7.4.2.3 64-bit OS upgrade (Windows and Linux)	62
7.4.2.4 Rename your driver (Windows and Linux)	62
7.4.2.5 Ensure that your code uses the correct driver module	62
7.4.2.6 Rebuild your updated driver	62

7.4.2.7 Upgrade Your Device INF File (Windows)	62
7.4.2.8 Digitally Sign Your Driver Files (Windows)	62
7.4.2.9 Upgrade Your Driver Distribution/Installation Package	63
7.4.2.10 Check for changes in variable and struct sizes	63
7.5. 32-Bit Applications on 64-Bit Windows and Linux Platforms	63
7.5.1. Developing a 32-Bit Application for Both 32-Bit and 64-Bit Platforms	64
7.5.2 64-Bit and 32-Bit Data Types	65
7.6. WinDriver .NET APIs in PowerShell	65
7.7. WinDriver Server API	66
7.8. FAQ	67
7.8.1. Using WinDriver to build a GUI Application	67
7.8.2. Can WinDriver handle multiple devices, of different or similar types, at the same time?	67
7.8.3. Can I run two different device drivers, both developed with WinDriver, on the same machine?	67
7.8.4. Can WinDriver group I/O and memory transfers?	68
7.8.5. I need to define more than 20 "hardware items" (I/O, memory, and interrupts) for my ISA card. Therefore, I increased the value of WD_CARD_ITEMS in the windrvr.h header file (due to the definition of the Item member of the WD_CARD structure as an array of WD_CARD← ITEMS WD_ITEMS structures). But now WD_CardRegister() will not work. Why?	68
7.8.6. I have a WinDriver based application running on a certain operating system, how do I port my code to a different operating system?	68
8 Debugging Drivers	69
8.1. User-Mode Debugging	69
8.2. Debug Monitor	69
8.2.1. The wddebug_gui Utility	70
8.2.1.1. Search in wddebug_gui	72
8.2.1.2. Opening Windows kernel crash dump with wddebug_gui	72
8.2.1.3. Running wddebug_gui for a Renamed Driver	73
8.2.2. The wddebug Utility	73
8.2.2.1. Console-Mode wddebug Execution	73
8.2.2.2. Debugging on a test machine	75
8.3. FAQ	75
8.3.1. Should I use wddebug_gui or wddebug?	75
8.3.2. Can I debug WinDriver-based code easily using MS Visual Studio (Visual C++)?	76
8.3.3. How would you recommend debugging a WinDriver-based application?	76
9 Enhanced Support for Specific Chipsets	77
9.1. Enhanced Support for Specific Chipsets Overview	77
9.2. Developing a Driver Using the Enhanced Chipset Support	77
9.3. The XDMA sample code	77
9.3.1. Performing Direct Memory Access (DMA) tests	78
9.3.2. The XDMA GUI utility	79
9.3.3. XDMA code generation in DriverWizard	82
9.4. The QDMA sample code	82

9.4.1. Performing Direct Memory Access (DMA) transaction	82
9.4.2. Changing between physical functions	83
9.4.3. Requests info	83
9.4.4. QDMA code generation in DriverWizard	83
9.5. The Avalon-MM sample code	84
9.5.1. Avalon-MM code generation in DriverWizard	84
10 PCI Advanced Features	87
10.1. Handling Interrupts	87
10.1.1. Interrupt Handling - Overview	87
10.1.2. WinDriver Interrupt Handling Sequence	88
10.1.3. Registering IRQs for Non-Plug-and-Play Hardware	89
10.1.4. Determining the Interrupt Types Supported by the Hardware	90
10.1.5. Determining the Interrupt Type Enabled for a PCI Card	90
10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands	90
10.1.6.1. Interrupt Mask Commands	91
10.1.6.2. Sample WinDriver Transfer Commands Code	91
10.1.7. WinDriver MSI/MSI-X Interrupt Handling	92
10.1.7.1. Windows MSI/MSI-X Device INF Files	93
10.1.8. Sample User-Mode WinDriver Interrupt Handling Code	93
10.2. Reserving and locking physical memory on Windows and Linux	94
10.3. Buffer sharing between multiple processes	97
10.4. Single Root I/O Virtualization (SR-IOV)	98
10.4.1. Introduction	98
10.4.2. Using SR-IOV with WinDriver	98
10.5. Using WinDriver's IPC APIs	99
10.5.1 IPC Overview	99
10.5.2. Shared Interrupts via IPC	99
10.5.2.1. Enabling Shared Interrupts:	99
10.5.2.2. Disabling Shared Interrupts:	99
10.6. FAQ	100
10.6.1. What is the significance of marking a resource as 'shared' with WinDriver, and how can I verify the 'shared' status of a specific resource on my card?	100
10.6.2. Can I access the same device, simultaneously, from several WinDriver applications?	100
10.6.3. How can I read the value of the PCI interrupt status register from my WinDriver ISR, in order to determine, for example, which card generated the interrupt when the IRQ is shared between several devices?	101
10.6.4. I need to be able to count the number of interrupts occurring and possibly call a routine every time an interrupt occurs. Is this possible with WinDriver?	101
10.6.5. Does WinDriver poll the interrupt (Busy Wait)?	101
10.6.6. Can I write to disk files during an interrupt routine?	101
11 Improving PCI Performance	103
11.1. Improving PCI Performance Overview	103

11.1.1. Performance Improvement Checklist	103
11.1.2 PCI Transfers Overview	104
11.1.3. Improving the Performance of a User-Mode Driver	105
11.1.3.1. Using Direct Access to Memory-Mapped Regions	105
11.1.3.2. Block Transfers and Grouping Multiple Transfers (Burst Transfer)	106
11.1.3.3. Performing 64-Bit Data Transfers	106
11.2. Performing Direct Memory Access (DMA)	107
11.2.1. Implementing Scatter/Gather DMA	108
11.2.1.1. C Example	108
11.2.1.2. C# Example	109
11.2.1.3. What Should You Implement?	110
11.2.2. Implementing Contiguous-Buffer DMA	110
11.2.2.1. C Example	110
11.2.2.2. C# Example	111
11.2.2.3. What Should You Implement?	112
11.2.2.4. Preallocating Contiguous DMA Buffers on Windows	112
11.3. Performing Direct Memory Access (DMA) transactions	114
11.3.1. Implementing Scatter/Gather DMA transactions	115
11.3.1.1. C Example	116
11.3.1.2. C# Example	117
11.3.1.3. What Should You Implement?	118
11.3.2. Implementing Contiguous-Buffer DMA transactions	118
11.3.2.1. C Example	118
11.3.2.2. C# Example	120
11.3.2.3. What Should You Implement?	120
11.4. DMA between PCI devices and NVIDIA GPUs with GPUDirect (Linux only)	120
11.4.1. What is GPUDirect for RDMA?	120
11.4.2. System Requirements	121
11.4.3. Software Prerequisites	121
11.4.4. WinDriver installation	121
11.4.5. Moving DMA from CPU to GPU	121
11.4.6. Modify Compilation	121
11.4.7. Modify your code	121
11.4.8. CMake Example	122
11.5. FAQ	122
11.5.1. How do I perform system or slave DMA using WinDriver?	122
11.5.2. I have locked a memory buffer for DMA on Windows. Now, when I access this memory directly, using the user-mode pointer, it seems to be 5 times slower than accessing a “regular” memory buffer, allocated with malloc(). Why?	122
11.5.3. My attempt to allocate and lock a 1GB DMA buffer with WinDriver on Windows fails. Is this a limitation of the operating system?	123
11.5.4. How do I perform PCI DMA Writes from system memory to my card, using WinDriver?	123
11.5.5. How do I perform Direct Block transfers from one PCI card to another?	123

12 Understanding the Kernel PlugIn	125
12.1. Background	125
12.2. Expected Performance	125
12.3. Overview of the Development Process	125
12.4. The Kernel PlugIn Architecture	125
12.4.1. Architecture Overview	125
12.4.2. WinDriver's Kernel and Kernel PlugIn Interaction	126
12.4.3. Kernel PlugIn Components	126
12.4.4. Kernel PlugIn Event Sequence	127
12.4.4.1. Opening a Handle from the User Mode to a Kernel PlugIn Driver	127
12.4.4.2. Handling User-Mode Requests from the Kernel PlugIn	127
12.4.4.3. Interrupt Handling - Enable/Disable and High Interrupt Request Level Processing	127
12.4.4.4. Interrupt Handling - Deferred Procedure Calls Event/Callback Notes	128
12.4.4.5. Plug-and-Play and Power Management Events	128
12.5. How Does Kernel PlugIn Work?	129
12.5.1. Minimal Requirements for Creating a Kernel PlugIn Driver	129
12.5.2. Kernel PlugIn Implementation	129
12.5.2.1. Before You Begin	129
12.5.2.2. Write Your KP_Init Function	129
12.5.2.3. Write Your KP_Open Function(s)	130
12.5.2.4. Write the Remaining PlugIn Callbacks	132
12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview	132
12.5.4. Kernel PlugIn Sample/Generated Code Directory Structure	133
12.5.4.1. pci_diag and kp_pci Sample Directories	133
12.5.4.2. Xilinx BMD Kernel PlugIn Directory Structure	134
12.5.4.3. The Generated DriverWizard Kernel PlugIn Directory	134
12.5.5. Handling Interrupts in the Kernel PlugIn	135
12.5.5.1. Interrupt Handling in the User Mode (Without the Kernel PlugIn)	136
12.5.5.2. Interrupt Handling in the Kernel (Using the Kernel PlugIn)	136
12.5.6. Message Passing	138
12.6. FAQ	138
12.6.1. Why does my WD_KernelPlugInOpen() call fail?	138
12.6.2. When handling my interrupts entirely in the Kernel PlugIn, can I erase the interrupt handler in the user mode?	138
12.6.3. How can I print debug statements from the Kernel PlugIn that I can view using a kernel debugger, such as WinDbg?	139
12.6.4. My PC hangs while closing my application. The code fails in WD_IntDisable(). Why is this happening? I am using the Kernel PlugIn to handle interrupts.	139
13 Creating a Kernel PlugIn Driver	141
13.1. Determine Whether a Kernel PlugIn is Needed	141
13.2. What programming languages can be used with a Kernel PlugIn?	141
13.3. Prepare the User-Mode Source Code	141

13.4. Create a New Kernel PlugIn Project	142
13.5. Open a Handle to the Kernel PlugIn	143
13.6. Set Interrupt Handling in the Kernel PlugIn	143
13.7. Set I/O Handling in the Kernel PlugIn	144
13.8. Compile Your Kernel PlugIn Driver	144
13.8.1. Windows Kernel PlugIn Driver Compilation	144
13.8.2. Linux Kernel PlugIn Driver Compilation	146
13.8.3. Porting a Kernel PlugIn project developed prior to version 10.3.0, to support working with a 32-bit user-mode application and a 64-bit Kernel PlugIn driver	147
13.9. Install Your Kernel PlugIn Driver	150
13.9.1. Windows Kernel PlugIn Driver Installation	150
13.9.2. Linux Kernel PlugIn Driver Installation	150
13.10. FAQ	151
13.10.1. I would like to execute in the kernel some pieces of code written in languages other than C/C++ (Python/Java/C#/Visual Basic.NET), using the Kernel PlugIn. Is it possible?	151
13.10.2. How do I allocate locked memory in the kernel that can be used inside the interrupt handler?	151
13.10.3. How do I handle shared PCI interrupts in the Kernel PlugIn?	151
13.10.4. What is plntContext in the Kernel PlugIn interrupt functions?	153
13.10.5. I need to call WD_Transfer() in the Kernel PlugIn. From where do I get hWD to pass to these functions?	153
13.10.6. A restriction in KP_IntAtIrql is to use only non-pageable memory. What does this mean?	153
13.10.7. How do I call WD_Transfer() in the Kernel PlugIn interrupt handler?	154
13.10.8. How do I share a memory buffer between Kernel PlugIn and user-mode projects for DMA or other purposes?	154
13.10.9. If I write a new function in my SYS Kernel PlugIn driver, must it also be declared with __cdecl?	154
14 USB Advanced Features	155
14.1. USB Control Transfers	155
14.2. USB Control Transfers Overview	155
14.2.1. Control Data Exchange	155
14.2.2. More About the Control Transfer	155
14.2.3. The Setup Packet	156
14.2.4. USB Setup Packet Format	156
14.2.5. Standard Device Request Codes	157
14.2.6. Setup Packet Example	157
14.3. Performing Control Transfers with WinDriver	158
14.3.1. Control Transfers with DriverWizard	158
14.3.2. Control Transfers with WinDriver API	160
14.4. Functional USB Data Transfers	161
14.4.1. Functional USB Data Transfers Overview	161
14.4.2. Single-Blocking Transfers	161
14.4.2.1. Performing Single-Blocking Transfers with WinDriver	161
14.4.3. Streaming Data Transfers	161

14.4.3.1. Performing Streaming with WinDriver	161
14.5. FAQ	162
14.5.1. Buffer Overrun Error: WDU_Transfer() sometimes returns the 0xC000000C error code. What does this error code mean? How do I solve this problem?	162
14.5.2. How do I extract the string descriptors contained in the Device and Configuration descriptor tables?	163
14.5.3. How do I detect that a USB device has been plugged in or disconnected?	163
14.5.4. How do I setup the transfer buffer to send a null data packet through the control pipe?	163
14.5.5. Can I write a driver for a USB hub or a USB Host Controller card using WinDriver?	163
14.5.6. Does WinDriver USB support isochronous streaming mode?	163
15 Distributing Your Driver	165
15.1. Getting a Valid WinDriver License	165
15.2. Windows Driver Distribution	165
15.2.1. Preparing the Distribution Package	166
15.2.2. Installing Your Driver on the Target Computer	166
15.2.3. Installing Your Kernel PlugIn on the Target Computer	168
15.2.4. Redistribute Your WinDriver-based package as a self-extracting EXE	169
15.2.4.1. The Installer	169
15.2.4.2. Requirements	169
15.2.4.3. Instructions	170
15.3. Linux Driver Distribution	174
15.3.1. Preparing the Distribution Package	174
15.3.1.1. Kernel Module Components	174
15.3.1.2. User-Mode Hardware-Control Application or Shared Object	176
15.3.2. Building and Installing the WinDriver Driver Module on the Target	176
15.3.3. Building and Installing Your Kernel PlugIn Driver on the Target	177
15.3.4. Installing the User-Mode Hardware-Control Application or Shared Object	178
15.3.5. Redistribute Your WinDriver-based package as a self-extracting SH (STGZ)	178
15.3.5.1. The Installer	178
15.3.5.2. Requirements	178
15.3.5.3. Instructions	178
16 Dynamically Loading Your Driver	181
16.1. Why Do You Need a Dynamically Loadable Driver?	181
16.2. Windows Dynamic Driver Loading	181
16.2.1. The wdreg Utility	181
16.2.1.1. WDM Drivers	181
16.2.1.2. Non-WDM Drivers	182
16.2.2. Dynamically Loading/Unloading windrvr1511.sys INF Files	183
16.2.3. Dynamically Loading/Unloading Your Kernel PlugIn Driver	184
16.3. The wdreg_frontend utility	184
16.4. Linux Dynamic Driver Loading	187

16.4.1. Dynamically Loading/Unloading Your Kernel Plugin Driver	187
17 Driver Installation - Advanced Issues	189
17.1. Windows INF Files	189
17.1.1. Why Should I Create an INF File?	189
17.1.2. How Do I Install an INF File When No Driver Exists?	189
17.1.3. How Do I Replace an Existing Driver Using the INF File?	190
17.2. Renaming the WinDriver Kernel Driver	190
17.2.1. Windows Driver Renaming	191
17.2.1.1 Rebuilding with Custom Version Information (Windows)	192
17.2.2. Linux Driver Renaming	192
17.3. Windows Digital Driver Signing and Certification	193
17.3.1. Windows Digital Driver Signing and Certification Overview	193
17.3.1.1. Authenticode Driver Signature	194
17.3.1.2. Windows Certification Program	194
17.3.2. Driver Signing and Certification of WinDriver-Based Drivers	194
17.3.2.1. HCK/HLK Test Notes	195
17.3.3. Secure Boot and Driver Development	195
17.3.3.1. Through System Information	196
17.3.3.2. Through PowerShell	197
17.3.3.3. Through WinDriver User-Mode API	198
17.3.3.4. Disabling Secure Boot	198
17.3.4. Temporary disabling digital signature enforcement in Windows 10	198
18 Data Structure Index	199
Data Structures	199
19 File Index	201
File List	201
20 Data Structure Documentation	203
CCString Class Reference	203
Detailed Description	204
Constructor & Destructor Documentation	204
CCString() [1/3]	204
CCString() [2/3]	204
CCString() [3/3]	204
~CCString()	204
Member Function Documentation	204
cat_printf()	205
Compare()	205
CompareNoCase()	205
contains()	205
find_first()	205

find_last()	205
Format()	205
GetBuffer()	205
Init()	206
is_empty()	206
IsAllocOK()	206
Length()	206
MakeLower()	206
MakeUpper()	206
Mid() [1/2]	206
Mid() [2/2]	206
operator char *()	206
operator PCSTR()	207
operator"!=" [1/3]	207
operator"!=" [2/3]	207
operator"!=" [3/3]	207
operator+=()	207
operator=() [1/2]	207
operator=() [2/2]	207
operator==() [1/2]	207
operator==() [2/2]	207
operator[]()	208
sprintf()	208
strcmp()	208
stricmp()	208
StrRemove()	208
StrReplace()	208
substr()	208
tolower()	208
tolower_copy()	209
toupper()	209
toupper_copy()	209
trim()	209
vsprintf()	209
Field Documentation	209
m_buf_size	209
m_str	209
KP_INIT Struct Reference	209
Detailed Description	210
Field Documentation	210
cDriverName	210
dwVerWD	210

funcOpen	210
funcOpen_32_64	210
KP_OPEN_CALL Struct Reference	211
Detailed Description	211
Field Documentation	211
funcCall	211
funcClose	211
funcEvent	211
funcIntAtDpc	211
funcIntAtDpcMSI	212
funcIntAtIrql	212
funcIntAtIrqlMSI	212
funcIntDisable	212
funcIntEnable	212
WD_BUS Struct Reference	212
Detailed Description	213
Field Documentation	213
dwBusNum	213
dwBusType	213
dwDomainNum	213
dwSlotFunc	213
WD_CARD Struct Reference	213
Detailed Description	214
Field Documentation	214
dwItems	214
Item	214
WD_CARD_CLEANUP Struct Reference	214
Detailed Description	214
Field Documentation	214
Cmd	215
dwCmds	215
dwOptions	215
hCard	215
WD_CARD_REGISTER Struct Reference	215
Detailed Description	216
Field Documentation	216
Card	216
cDescription	216
cName	216
dwOptions	216
fCheckLockOnly	216
hCard	216

WD_DEBUG Struct Reference	217
Detailed Description	217
Field Documentation	217
dwBufferSize	217
dwCmd	217
dwLevel	218
dwLevelMessageBox	218
dwSection	218
WD_DEBUG_ADD Struct Reference	218
Detailed Description	218
Field Documentation	218
dwLevel	218
dwSection	219
pcBuffer	219
WD_DEBUG_DUMP Struct Reference	219
Detailed Description	219
Field Documentation	219
cBuffer	219
WD_DMA Struct Reference	219
Detailed Description	220
Field Documentation	220
DMATransactionCallback	220
DMATransactionCallbackCtx	220
dwAlignment	220
dwBytes	221
dwBytesTransferred	221
dwMaxTransferSize	221
dwOptions	221
dwPages	221
dwTransferElementSize	221
hCard	221
hDma	222
Page	222
pKernelAddr	222
pUserAddr	222
WD_DMA_PAGE Struct Reference	222
Detailed Description	222
Field Documentation	222
dwBytes	223
pPhysicalAddr	223
WD_EVENT Struct Reference	223
Detailed Description	224

Field Documentation	224
cardId	224
dwAction	224
dwEventId	224
dwEventType	224
dwGroupID	224
dwMsgID	224
dwNumMatchTables	225
dwOptions	225
dwSenderUID	225
dwSubGroupID	225
dwUniqueID	225
hEvent	225
hLpc	225
hKernelPlugin	225
.	226
matchTables	226
.	226
pciSlot	226
qwMsgData	226
.	226
.	226
WD_GET_DEVICE_PROPERTY Struct Reference	226
Detailed Description	227
Field Documentation	227
dwBytes	227
dwOptions	227
dwProperty	227
dwUniqueID	227
.	227
hDevice	228
pBuf	228
WD_INTERRUPT Struct Reference	228
Detailed Description	228
Field Documentation	228
Cmd	229
dwCmds	229
dwCounter	229
dwEnabledIntType	229
dwLastMessage	229
dwLost	229
dwOptions	229

fEnableOk	230
fStopped	230
hInterrupt	230
kpCall	230
WD_IPC_PROCESS Struct Reference	230
Detailed Description	230
Field Documentation	231
cProcessName	231
dwGroupID	231
dwSubGroupID	231
hIpc	231
WD_IPC_REGISTER Struct Reference	231
Detailed Description	231
Field Documentation	232
dwOptions	232
procInfo	232
WD_IPC_SCAN_PROCS Struct Reference	232
Detailed Description	232
Field Documentation	232
dwNumProcs	232
hIpc	233
procInfo	233
WD_IPC_SEND Struct Reference	233
Detailed Description	233
Field Documentation	233
dwMsgID	233
dwOptions	233
dwRecipientID	234
hIpc	234
qwMsgData	234
WD_ITEMS Struct Reference	234
Detailed Description	235
Field Documentation	235
Bus	235
dwBar	235
dwBytes	235
dwInterrupt	236
dwOptions	236
dwReserved1	236
fNotSharable	236
hInterrupt	236
.	236

item	237
.	237
pAddr	237
pPhysicalAddr	237
pReserved	237
pReserved2	237
pTransAddr	238
pUserDirectAddr	238
qwBytes	238
WD_KERNEL_BUFFER Struct Reference	238
Detailed Description	238
Field Documentation	238
dwOptions	239
hKerBuf	239
pKernelAddr	239
pUserAddr	239
qwBytes	239
WD_KERNEL_PLUGIN Struct Reference	239
Detailed Description	240
Member Function Documentation	240
PAD_TO_64()	240
Field Documentation	240
cDriverName	240
cDriverPath	240
hKernelPlugIn	240
WD_KERNEL_PLUGIN_CALL Struct Reference	240
Detailed Description	241
Field Documentation	241
dwMessage	241
dwResult	241
hKernelPlugIn	241
pData	241
WD_LICENSE Struct Reference	241
Detailed Description	242
Field Documentation	242
cLicense	242
WD_OS_INFO Struct Reference	242
Detailed Description	242
Field Documentation	242
cBuild	242

cCsdVersion	242
cCurrentVersion	243
cInstallationType	243
cProdName	243
dwMajorVersion	243
dwMinorVersion	243
WD_PCI_CAP Struct Reference	243
Detailed Description	243
Field Documentation	244
dwCapId	244
dwCapOffset	244
WD_PCI_CARD_INFO Struct Reference	244
Detailed Description	244
Field Documentation	244
Card	244
pciSlot	245
WD_PCI_CONFIG_DUMP Struct Reference	245
Detailed Description	245
Field Documentation	245
dwBytes	245
dwOffset	245
dwResult	246
flsRead	246
pBuffer	246
pciSlot	246
WD_PCI_ID Struct Reference	246
Detailed Description	246
Field Documentation	246
dwDeviceId	247
dwVendorId	247
WD_PCI_SCAN_CAPS Struct Reference	247
Detailed Description	247
Field Documentation	247
dwCapId	247
dwNumCaps	248
dwOptions	248
pciCaps	248
pciSlot	248
WD_PCI_SCAN_CARDS Struct Reference	248
Detailed Description	249
Field Documentation	249
cardId	249

cardSlot	249
dwCards	249
dwOptions	249
searchId	249
WD_PCI_SLOT Struct Reference	250
Detailed Description	250
Field Documentation	250
dwBus	250
dwDomain	250
dwFunction	250
dwSlot	251
WD_PCI_SRIOV Struct Reference	251
Detailed Description	251
Field Documentation	251
dwNumVFs	251
pciSlot	251
WD_SLEEP Struct Reference	251
Detailed Description	252
Field Documentation	252
dwMicroSeconds	252
dwOptions	252
WD_TRANSFER Struct Reference	252
Detailed Description	253
Field Documentation	253
Byte	253
cmdTrans	253
.	253
dwBytes	253
dwOptions	253
Dword	254
fAutoinc	254
pBuffer	254
pPort	254
Qword	254
Word	254
WD_USAGE Struct Reference	255
Detailed Description	255
Field Documentation	255
applications_num	255
devices_num	255
WD_VERSION Struct Reference	255
Detailed Description	255

Field Documentation	255
cVer	256
dwVer	256
WDC_ADDR_DESC Struct Reference	256
Detailed Description	256
Field Documentation	256
dwAddrSpace	256
dwItemIndex	257
flsMemory	257
pAddr	257
pUserDirectMemAddr	257
qwBytes	257
reserved	257
WDC_DEVICE Struct Reference	257
Detailed Description	258
Field Documentation	258
cardReg	258
dwNumAddrSpaces	258
Event	258
hEvent	259
hIntThread	259
id	259
Int	259
kerPlug	259
pAddrDesc	259
pCtx	259
slot	260
WDC_INTERRUPT_PARAMS Struct Reference	260
Detailed Description	260
Field Documentation	260
dwNumCmds	260
dwOptions	260
funcIntHandler	260
fUseKP	261
pData	261
pTransCmds	261
WDC_PCI_SCAN_CAPS_RESULT Struct Reference	261
Detailed Description	261
Field Documentation	261
dwNumCaps	261
pciCaps	262
WDC_PCI_SCAN_RESULT Struct Reference	262

Detailed Description	262
Field Documentation	262
deviceID	262
deviceSlot	262
dwNumDevices	263
WDS_IPC_MSG_RX Struct Reference	263
Detailed Description	263
Field Documentation	263
dwMsgID	263
dwSenderUID	263
qwMsgData	264
WDS_IPC_SCAN_RESULT Struct Reference	264
Detailed Description	264
Field Documentation	264
dwNumProcs	264
procInfo	264
WDU_ALTERNATE_SETTING Struct Reference	264
Detailed Description	265
Field Documentation	265
Descriptor	265
pEndpointDescriptors	265
pPipes	265
WDU_CONFIGURATION Struct Reference	265
Detailed Description	266
Field Documentation	266
Descriptor	266
dwNumInterfaces	266
pInterfaces	266
WDU_CONFIGURATION_DESCRIPTOR Struct Reference	266
Detailed Description	267
Field Documentation	267
bConfigurationValue	267
bDescriptorType	267
bLength	267
bmAttributes	267
bNumInterfaces	268
iConfiguration	268
MaxPower	268
wTotalLength	268
WDU_DEVICE Struct Reference	268
Detailed Description	269
Field Documentation	269

Descriptor	269
pActiveConfig	269
pActiveInterface	269
pConfigs	269
Pipe0	269
WDU_DEVICE_DESCRIPTOR Struct Reference	270
Detailed Description	270
Field Documentation	270
bcdDevice	270
bcdUSB	271
bDescriptorType	271
bDeviceClass	271
bDeviceProtocol	271
bDeviceSubClass	271
bLength	271
bMaxPacketSize0	271
bNumConfigurations	272
idProduct	272
idVendor	272
iManufacturer	272
iProduct	272
iSerialNumber	272
WDU_ENDPOINT_DESCRIPTOR Struct Reference	272
Detailed Description	273
Field Documentation	273
bDescriptorType	273
bEndpointAddress	273
bInterval	273
bLength	274
bmAttributes	274
wMaxPacketSize	274
WDU_EVENT_TABLE Struct Reference	274
Detailed Description	274
Field Documentation	274
pfDeviceAttach	274
pfDeviceDetach	275
pfPowerChange	275
pUserData	275
WDU_GET_DESCRIPTOR Struct Reference	275
Detailed Description	275
Field Documentation	275
blIndex	275

bType	276
dwUniqueID	276
pBuffer	276
wLanguage	276
wLength	276
WDU_GET_DEVICE_DATA Struct Reference	276
Detailed Description	276
Field Documentation	276
dwBytes	277
dwOptions	277
dwUniqueID	277
pBuf	277
WDU_HALT_TRANSFER Struct Reference	277
Detailed Description	277
Field Documentation	277
dwOptions	277
dwPipeNum	278
dwUniqueID	278
WDU_INTERFACE Struct Reference	278
Detailed Description	278
Field Documentation	278
dwNumAltSettings	278
pActiveAltSetting	278
pAlternateSettings	279
WDU_INTERFACE_DESCRIPTOR Struct Reference	279
Detailed Description	279
Field Documentation	279
bAlternateSetting	279
bDescriptorType	280
bInterfaceClass	280
bInterfaceNumber	280
bInterfaceProtocol	280
bInterfaceSubClass	280
bLength	280
bNumEndpoints	280
iInterface	281
WDU_MATCH_TABLE Struct Reference	281
Detailed Description	281
Field Documentation	281
bDeviceClass	281
bDeviceSubClass	282
bInterfaceClass	282

blInterfaceProtocol	282
blInterfaceSubClass	282
wProductId	282
wVendorId	282
WDU_PIPE_INFO Struct Reference	282
Detailed Description	283
Field Documentation	283
direction	283
dwInterval	283
dwMaximumPacketSize	283
dwNumber	284
type	284
WDU_RESET_DEVICE Struct Reference	284
Detailed Description	284
Field Documentation	284
dwOptions	284
dwUniqueID	284
WDU_RESET_PIPE Struct Reference	285
Detailed Description	285
Field Documentation	285
dwOptions	285
dwPipeNum	285
dwUniqueID	285
WDU_SELECTIVE_SUSPEND Struct Reference	285
Detailed Description	285
Field Documentation	286
dwOptions	286
dwUniqueID	286
WDU_SET_INTERFACE Struct Reference	286
Detailed Description	286
Field Documentation	286
dwAlternateSetting	286
dwInterfaceNum	286
dwOptions	287
dwUniqueID	287
WDU_STREAM Struct Reference	287
Detailed Description	287
Field Documentation	287
dwBufferSize	287
dwOptions	287
dwPipeNum	288
dwReserved	288

dwRxSize	288
dwRxTxTimeout	288
dwUniqueID	288
fBlocking	288
WDU_STREAM_STATUS Struct Reference	288
Detailed Description	289
Field Documentation	289
dwBytesInBuffer	289
dwLastError	289
dwOptions	289
dwReserved	289
dwUniqueID	289
flsRunning	289
WDU_TRANSFER Struct Reference	289
Detailed Description	290
Field Documentation	290
dwBufferSize	290
dwBytesTransferred	290
dwOptions	290
dwPipeNum	291
dwTimeout	291
dwUniqueID	291
fRead	291
pBuffer	291
SetupPacket	291
WDU_WAKEUP Struct Reference	291
Detailed Description	292
Field Documentation	292
dwOptions	292
dwUniqueID	292
21 File Documentation	293
bits.h File Reference	293
Enumeration Type Documentation	293
BIT	293
bits.h	294
cstring.h File Reference	294
Typedef Documentation	295
PCSTR	295
Function Documentation	295
operator+()	295
strcmp()	295

cstring.h	295
kpstdlib.h File Reference	296
Macro Definition Documentation	297
__KERNEL__	297
COPY_FROM_USER_OR_KERNEL	298
COPY_TO_USER_OR_KERNEL	298
FALSE	298
NULL	299
TRUE	299
Typedef Documentation	299
KP_INTERLOCKED	299
KP_SPINLOCK	299
Function Documentation	299
free()	299
KDBG()	299
kp_interlocked_add()	300
kp_interlocked_decrement()	300
kp_interlocked_exchange()	300
kp_interlocked_increment()	301
kp_interlocked_init()	301
kp_interlocked_read()	301
kp_interlocked_set()	302
kp_interlocked_uninit()	302
kp_spinlock_init()	302
kp_spinlock_release()	302
kp_spinlock_uninit()	303
kp_spinlock_wait()	303
malloc()	303
strcpy()	303
kpstdlib.h	304
pci_regs.h File Reference	306
Macro Definition Documentation	318
GET_CAPABILITY_STR	318
GET_EXTENDED_CAPABILITY_STR	319
PCI_CAP_EXP_ENDPOINT_SIZEOF_V1	319
PCI_CAP_EXP_ENDPOINT_SIZEOF_V2	319
PCI_CAP_ID_AF	319
PCI_CAP_ID_AGP	319
PCI_CAP_ID_AGP3	320
PCI_CAP_ID_CCRC	320
PCI_CAP_ID_CHSWP	320
PCI_CAP_ID_DBG	320

PCI_CAP_ID_EXP	320
PCI_CAP_ID_HT	320
PCI_CAP_ID_MSI	320
PCI_CAP_ID_MSIX	321
PCI_CAP_ID_PCIX	321
PCI_CAP_ID_PM	321
PCI_CAP_ID_SATA	321
PCI_CAP_ID_SECDEV	321
PCI_CAP_ID_SHPC	321
PCI_CAP_ID_SLOTID	322
PCI_CAP_ID_SSVID	322
PCI_CAP_ID_VNDR	322
PCI_CAP_ID_VPD	322
PCI_CAP_LIST_ID	322
PCI_CAP_LIST_NEXT	322
PCI_COMMAND	322
PCI_COMMAND_FAST_BACK	323
PCI_COMMAND_INTX_DISABLE	323
PCI_COMMAND_INVALIDATE	323
PCI_COMMAND_IO	323
PCI_COMMAND_MASTER	323
PCI_COMMAND_MEMORY	323
PCI_COMMAND_PARITY	323
PCI_COMMAND_SERR	324
PCI_COMMAND_SPECIAL	324
PCI_COMMAND_VGA_PALETTE	324
PCI_COMMAND_WAIT	324
PCI_EXP_DEVCAP	324
PCI_EXP_DEVCAP2	324
PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP	324
PCI_EXP_DEVCAP2_ARI	325
PCI_EXP_DEVCAP2_ATOMIC_COMP32	325
PCI_EXP_DEVCAP2_ATOMIC_COMP64	325
PCI_EXP_DEVCAP2_ATOMIC_ROUTE	325
PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP	325
PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP	325
PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP	325
PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT	326
PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP	326
PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP	326
PCI_EXP_DEVCAP2_LTR	326
PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES	326

PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR	326
PCI_EXP_DEVCAP2_OBFF_MASK	326
PCI_EXP_DEVCAP2_OBFF_MSG	327
PCI_EXP_DEVCAP2_OBFF_WAKE	327
PCI_EXP_DEVCAP2_RANGE_A	327
PCI_EXP_DEVCAP2_RANGE_B	327
PCI_EXP_DEVCAP2_RANGE_C	327
PCI_EXP_DEVCAP2_RANGE_D	327
PCI_EXP_DEVCAP2_TPH_COMP_SUPP	327
PCI_EXP_DEVCAP_ATN_BUT	328
PCI_EXP_DEVCAP_ATN_IND	328
PCI_EXP_DEVCAP_EXT_TAG	328
PCI_EXP_DEVCAP_FLR	328
PCI_EXP_DEVCAP_L0S	328
PCI_EXP_DEVCAP_L1	328
PCI_EXP_DEVCAP_L1_SHIFT	328
PCI_EXP_DEVCAP_PAYLOAD	329
PCI_EXP_DEVCAP_PHANTOM	329
PCI_EXP_DEVCAP_PHANTOM_SHIFT	329
PCI_EXP_DEVCAP_PWD_SCL_SHIFT	329
PCI_EXP_DEVCAP_PWR_IND	329
PCI_EXP_DEVCAP_PWR_SCL	329
PCI_EXP_DEVCAP_PWR_VAL	329
PCI_EXP_DEVCAP_PWR_VAL_SHIFT	330
PCI_EXP_DEVCAP_RBER	330
PCI_EXP_DEVCTL	330
PCI_EXP_DEVCTL2	330
PCI_EXP_DEVCTL2_ARI	330
PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK	330
PCI_EXP_DEVCTL2_ATOMIC_REQ	330
PCI_EXP_DEVCTL2_COMP_TIMEOUT	331
PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE	331
PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK	331
PCI_EXP_DEVCTL2_IDO_CMP_EN	331
PCI_EXP_DEVCTL2_IDO_REQ_EN	331
PCI_EXP_DEVCTL2_LTR_EN	331
PCI_EXP_DEVCTL2_OBFF_MSGA_EN	331
PCI_EXP_DEVCTL2_OBFF_MSGB_EN	332
PCI_EXP_DEVCTL2_OBFF_WAKE_EN	332
PCI_EXP_DEVCTL_AUX_PME	332
PCI_EXP_DEVCTL_BCR_FLR	332
PCI_EXP_DEVCTL_CERE	332

PCI_EXP_DEVCTL_EXT_TAG	332
PCI_EXP_DEVCTL_FERE	332
PCI_EXP_DEVCTL_NFERE	333
PCI_EXP_DEVCTL_NOSNOOP_EN	333
PCI_EXP_DEVCTL_PAYLOAD	333
PCI_EXP_DEVCTL_PAYLOAD_SHIFT	333
PCI_EXP_DEVCTL_PHANTOM	333
PCI_EXP_DEVCTL_READRQ	333
PCI_EXP_DEVCTL_READRQ_1024B	333
PCI_EXP_DEVCTL_READRQ_128B	334
PCI_EXP_DEVCTL_READRQ_256B	334
PCI_EXP_DEVCTL_READRQ_512B	334
PCI_EXP_DEVCTL_READRQ_SHIFT	334
PCI_EXP_DEVCTL_RELAX_EN	334
PCI_EXP_DEVCTL_URRE	334
PCI_EXP_DEVSTA	334
PCI_EXP_DEVSTA2	335
PCI_EXP_DEVSTA_AUXPD	335
PCI_EXP_DEVSTA_CED	335
PCI_EXP_DEVSTA_FED	335
PCI_EXP_DEVSTA_NFED	335
PCI_EXP_DEVSTA_TRPND	335
PCI_EXP_DEVSTA_URD	335
PCI_EXP_FLAGS	336
PCI_EXP_FLAGS_IRQ	336
PCI_EXP_FLAGS_SLOT	336
PCI_EXP_FLAGS_TYPE	336
PCI_EXP_FLAGS_TYPE_SHIFT	336
PCI_EXP_FLAGS_VERS	336
PCI_EXP_LNKCAP	336
PCI_EXP_LNKCAP2	337
PCI_EXP_LNKCAP2_CROSSLINK	337
PCI_EXP_LNKCAP2_SLS_2_5GB	337
PCI_EXP_LNKCAP2_SLS_5_0GB	337
PCI_EXP_LNKCAP2_SLS_8_0GB	337
PCI_EXP_LNKCAP_ASPM	337
PCI_EXP_LNKCAP_ASPM_SHIFT	337
PCI_EXP_LNKCAP_CLKPM	338
PCI_EXP_LNKCAP_DLLLARC	338
PCI_EXP_LNKCAP_L0SEL	338
PCI_EXP_LNKCAP_L0SEL_SHIFT	338
PCI_EXP_LNKCAP_L1EL	338

PCI_EXP_LNKCAP_L1EL_SHIFT	338
PCI_EXP_LNKCAP_LBNC	338
PCI_EXP_LNKCAP_MLW	339
PCI_EXP_LNKCAP_MLW_SHIFT	339
PCI_EXP_LNKCAP_PN	339
PCI_EXP_LNKCAP_SDERC	339
PCI_EXP_LNKCAP_SLS	339
PCI_EXP_LNKCAP_SLS_2_5GB	339
PCI_EXP_LNKCAP_SLS_5_0GB	339
PCI_EXP_LNKCTL	340
PCI_EXP_LNKCTL2	340
PCI_EXP_LNKCTL2_COMP_SOS	340
PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL	340
PCI_EXP_LNKCTL2_ENTER_COMP	340
PCI_EXP_LNKCTL2_ENTER_MOD_COMP	340
PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS	340
PCI_EXP_LNKCTL2_LNK_SPEED_2_5	341
PCI_EXP_LNKCTL2_LNK_SPEED_5_0	341
PCI_EXP_LNKCTL2_LNK_SPEED_8_0	341
PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH	341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK	341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT	341
PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL	341
PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK	342
PCI_EXP_LNKCTL_ASPM_L0S	342
PCI_EXP_LNKCTL_ASPM_L1	342
PCI_EXP_LNKCTL_ASPMC	342
PCI_EXP_LNKCTL_CCC	342
PCI_EXP_LNKCTL_CLKREQ_EN	342
PCI_EXP_LNKCTL_ES	342
PCI_EXP_LNKCTL_HAWD	343
PCI_EXP_LNKCTL_LABIE	343
PCI_EXP_LNKCTL_LBMIE	343
PCI_EXP_LNKCTL_LD	343
PCI_EXP_LNKCTL_RCB	343
PCI_EXP_LNKCTL_RL	343
PCI_EXP_LNKSTA	343
PCI_EXP_LNKSTA2	344
PCI_EXP_LNKSTA2_CDL	344
PCI_EXP_LNKSTA2_EQUALIZ_COMP	344
PCI_EXP_LNKSTA2_EQUALIZ_PH1	344
PCI_EXP_LNKSTA2_EQUALIZ_PH2	344

PCI_EXP_LNKSTA2_EQUALIZ_PH3	344
PCI_EXP_LNKSTA2_LINE_EQ_REQ	344
PCI_EXP_LNKSTA_CLS	345
PCI_EXP_LNKSTA_CLS_2_5GB	345
PCI_EXP_LNKSTA_CLS_5_0GB	345
PCI_EXP_LNKSTA_CLS_8_0GB	345
PCI_EXP_LNKSTA_DLLLA	345
PCI_EXP_LNKSTA_LABS	345
PCI_EXP_LNKSTA_LBMS	345
PCI_EXP_LNKSTA_LT	346
PCI_EXP_LNKSTA_NLW	346
PCI_EXP_LNKSTA_NLW_SHIFT	346
PCI_EXP_LNKSTA_NLW_X1	346
PCI_EXP_LNKSTA_NLW_X2	346
PCI_EXP_LNKSTA_NLW_X4	346
PCI_EXP_LNKSTA_NLW_X8	346
PCI_EXP_LNKSTA_SLC	347
PCI_EXP_RTCAP	347
PCI_EXP_RTCAP_CRSVIS	347
PCI_EXP_RTCTL	347
PCI_EXP_RTCTL_CRSSVE	347
PCI_EXP_RTCTL_PMEIE	347
PCI_EXP_RTCTL_SECEE	347
PCI_EXP_RTCTL_SEFEE	348
PCI_EXP_RTCTL_SENFEE	348
PCI_EXP_RTSTA	348
PCI_EXP_RTSTA_PENDING	348
PCI_EXP_RTSTA_PME	348
PCI_EXP_SLTCAP	348
PCI_EXP_SLTCAP2	348
PCI_EXP_SLTCAP_ABP	349
PCI_EXP_SLTCAP_AIC_SHIFT	349
PCI_EXP_SLTCAP_AIP	349
PCI_EXP_SLTCAP_EIP	349
PCI_EXP_SLTCAP_HPC	349
PCI_EXP_SLTCAP_HPS	349
PCI_EXP_SLTCAP_MRLSP	349
PCI_EXP_SLTCAP_NCCS	350
PCI_EXP_SLTCAP_PCP	350
PCI_EXP_SLTCAP_PIP	350
PCI_EXP_SLTCAP_PSN	350
PCI_EXP_SLTCAP SPLS	350

PCI_EXP_SLTCAP_SPLS_SHIFT	350
PCI_EXP_SLTCAP_SPLV	350
PCI_EXP_SLTCAP_SPLV_SHIFT	351
PCI_EXP_SLTCTL	351
PCI_EXP_SLTCTL2	351
PCI_EXP_SLTCTL_ABPE	351
PCI_EXP_SLTCTL_AIC	351
PCI_EXP_SLTCTL_ATTN_IND_BLINK	351
PCI_EXP_SLTCTL_ATTN_IND_OFF	351
PCI_EXP_SLTCTL_ATTN_IND_ON	352
PCI_EXP_SLTCTL_CCIE	352
PCI_EXP_SLTCTL_DLLSCE	352
PCI_EXP_SLTCTL_EIC	352
PCI_EXP_SLTCTL_HPIE	352
PCI_EXP_SLTCTL_MRLSCE	352
PCI_EXP_SLTCTL_PCC	352
PCI_EXP_SLTCTL_PDCE	353
PCI_EXP_SLTCTL_PFDE	353
PCI_EXP_SLTCTL_PIC	353
PCI_EXP_SLTCTL_PIC_SHIFT	353
PCI_EXP_SLTCTL_PWR_IND_BLINK	353
PCI_EXP_SLTCTL_PWR_IND_OFF	353
PCI_EXP_SLTCTL_PWR_IND_ON	353
PCI_EXP_SLTCTL_PWR_OFF	354
PCI_EXP_SLTCTL_PWR_ON	354
PCI_EXP_SLTSTA	354
PCI_EXP_SLTSTA2	354
PCI_EXP_SLTSTA_ABP	354
PCI_EXP_SLTSTA_CC	354
PCI_EXP_SLTSTA_DLLSC	354
PCI_EXP_SLTSTA_EIS	355
PCI_EXP_SLTSTA_MRLSC	355
PCI_EXP_SLTSTA_MRLSS	355
PCI_EXP_SLTSTA_PDC	355
PCI_EXP_SLTSTA_PDS	355
PCI_EXP_SLTSTA_PFD	355
PCI_EXP_TYPE_DOWNSTREAM	355
PCI_EXP_TYPE_ENDPOINT	356
PCI_EXP_TYPE_LEG_END	356
PCI_EXP_TYPE_PCI_BRIDGE	356
PCI_EXP_TYPE_PCIE_BRIDGE	356
PCI_EXP_TYPE_RC_EC	356

PCI_EXP_TYPE_RC_END	356
PCI_EXP_TYPE_ROOT_PORT	356
PCI_EXP_TYPE_UPSTREAM	357
PCI_EXT_CAP_ID	357
PCI_EXT_CAP_ID_ACS	357
PCI_EXT_CAP_ID_AMD_XXX	357
PCI_EXT_CAP_ID_ARI	357
PCI_EXT_CAP_ID_ATS	357
PCI_EXT_CAP_ID_CAC	357
PCI_EXT_CAP_ID_DPA	358
PCI_EXT_CAP_ID_DPC	358
PCI_EXT_CAP_ID_DSN	358
PCI_EXT_CAP_ID_ERR	358
PCI_EXT_CAP_ID_FRSQ	358
PCI_EXT_CAP_ID_L1PMS	358
PCI_EXT_CAP_ID_LNR	358
PCI_EXT_CAP_ID_LTR	359
PCI_EXT_CAP_ID_MCAST	359
PCI_EXT_CAP_ID_MFVC	359
PCI_EXT_CAP_ID_MPHY	359
PCI_EXT_CAP_ID_MRIOV	359
PCI_EXT_CAP_ID_PASID	359
PCI_EXT_CAP_ID_PMUX	359
PCI_EXT_CAP_ID_PRI	360
PCI_EXT_CAP_ID_PTMR	360
PCI_EXT_CAP_ID_PWR	360
PCI_EXT_CAP_ID_RCEC	360
PCI_EXT_CAP_ID_RCILC	360
PCI_EXT_CAP_ID_RCLD	360
PCI_EXT_CAP_ID_RCRB	360
PCI_EXT_CAP_ID_REBAR	361
PCI_EXT_CAP_ID_RTR	361
PCI_EXT_CAP_ID_SECPCI	361
PCI_EXT_CAP_ID_SRIOV	361
PCI_EXT_CAP_ID_TPH	361
PCI_EXT_CAP_ID_VC	361
PCI_EXT_CAP_ID_VC9	361
PCI_EXT_CAP_ID_VNDR	362
PCI_EXT_CAP_NEXT	362
PCI_EXT_CAP_VER	362
PCI_HEADER_TYPE	362
PCI_HEADER_TYPE_BRIDGE	362

PCI_HEADER_TYPE_CARDBUS	362
PCI_HEADER_TYPE_NORMAL	362
PCI_SR_CAP_LIST_BIT	362
PCI_STATUS	363
PCI_STATUS_66MHZ	363
PCI_STATUS_CAP_LIST	363
PCI_STATUS_DETECTED_PARITY	363
PCI_STATUS_DEVSEL_FAST	363
PCI_STATUS_DEVSEL_MASK	363
PCI_STATUS_DEVSEL_MEDIUM	363
PCI_STATUS_DEVSEL_SHIFT	364
PCI_STATUS_DEVSEL_SLOW	364
PCI_STATUS_FAST_BACK	364
PCI_STATUS_INTERRUPT	364
PCI_STATUS_PARITY	364
PCI_STATUS_REC_MASTER_ABORT	364
PCI_STATUS_REC_TARGET_ABORT	364
PCI_STATUS_SIG_SYSTEM_ERROR	365
PCI_STATUS_SIG_TARGET_ABORT	365
PCI_STATUS_UDF	365
Enumeration Type Documentation	365
AD_PCI_BAR	365
PCI_CONFIG_REGS_OFFSET	365
PCIE_CONFIG_REGS_OFFSET	366
WDC_PCI_HEADER_TYPE	367
pci_regs.h	367
pci_strings.h File Reference	373
Function Documentation	373
PciConfRegData2Str()	373
PciExpressConfRegData2Str()	374
pci_strings.h	374
status_strings.h File Reference	375
Function Documentation	375
Stat2Str()	375
status_strings.h	375
utils.h File Reference	376
Macro Definition Documentation	377
INFINITE	377
MAX_PATH	377
OsMemoryBarrier	377
snprintf	377
vsnprintf	377

Typedef Documentation	378
HANDLER_FUNC	378
Function Documentation	378
GetNumberOfProcessors()	378
GetPageSize()	378
OsEventClose()	378
OsEventCreate()	379
OsEventReset()	379
OsEventSignal()	379
OsEventWait()	380
OsMutexClose()	380
OsMutexCreate()	380
OsMutexLock()	380
OsMutexUnlock()	381
print2wstr()	381
PrintDbgMessage()	381
SleepWrapper()	382
UtilGetFileName()	382
UtilGetFileSize()	382
UtilGetStringFromUser()	383
vPrintDbgMessage()	383
utils.h	384
wd_kp.h File Reference	385
Macro Definition Documentation	386
KERNEL	386
KERPLUG	386
Typedef Documentation	386
KP_FUNC_CALL	386
KP_FUNC_CLOSE	386
KP_FUNC_EVENT	387
KP_FUNC_INT_AT_DPC	387
KP_FUNC_INT_AT_DPC_MSI	387
KP_FUNC_INT_AT_IRQL	387
KP_FUNC_INT_AT_IRQL_MSI	387
KP_FUNC_INT_DISABLE	387
KP_FUNC_INT_ENABLE	388
KP_FUNC_OPEN	388
Function Documentation	388
KP_Init()	388
wd_kp.h	388
wd_log.h File Reference	389
Macro Definition Documentation	390

WD_Close	390
WD_FUNCTION	390
WD_Open	390
Function Documentation	390
WD_CloseLog()	390
WD_LogAdd()	390
WD_LogStart()	391
WD_LogStop()	391
WD_OpenLog()	391
WdFunctionLog()	391
wd_log.h	392
wd_types.h File Reference	392
Typedef Documentation	392
u16	392
u32	392
u64	393
u8	393
wd_types.h	393
wd_ver.h File Reference	393
Macro Definition Documentation	394
COPYRIGHTS_FULL_STR	394
COPYRIGHTS_YEAR_STR	394
WD_MAJOR_VER	394
WD_MAJOR_VER_STR	394
WD_MINOR_VER	394
WD_MINOR_VER_STR	394
WD_SUB_MINOR_VER	394
WD_SUB_MINOR_VER_STR	395
WD_VER	395
WD_VER_BETA_STR	395
WD_VER_ITOA	395
WD_VERSION_MAC_STR	395
WD_VERSION_STR	395
wd_ver.h	395
wdc_defs.h File Reference	396
Macro Definition Documentation	397
WDC_ADDR_IS_MEM	397
WDC_GET_ADDR_DESC	397
WDC_GET_ADDR_SPACE_SIZE	397
WDC_GET_CARD_HANDLE	398
WDC_GET_ENABLED_INT_LAST_MSG	398
WDC_GET_ENABLED_INT_TYPE	399

WDC_GET_INT_OPTIONS	399
WDC_GET_KP_HANDLE	399
WDC_GET_PCARD	400
WDC_GET_PPCI_ID	400
WDC_GET_PPCI_SLOT	400
WDC_INT_IS_MSI	401
WDC_IS_KP	401
WDC_MEM_DIRECT_ADDR	401
Typedef Documentation	401
PWDC_DEVICE	401
WDC_DEVICE	401
wdc_defs.h	402
wdc_lib.h File Reference	403
Macro Definition Documentation	410
MAX_DESC	410
MAX_NAME	411
MAX_NAME_DISPLAY	411
WDC_AD_CFG_SPACE	411
WDC_ADDR_MODE_TO_SIZE	411
WDC_ADDR_SIZE_TO_MODE	411
WDC_DBG_DBM_ERR	411
WDC_DBG_DBM_FILE_ERR	411
WDC_DBG_DBM_FILE_TRACE	412
WDC_DBG_DBM_TRACE	412
WDC_DBG_DEFAULT	412
WDC_DBG_FILE_ERR	412
WDC_DBG_FILE_TRACE	412
WDC_DBG_FULL	412
WDC_DBG_LEVEL_ERR	412
WDC_DBG_LEVEL_TRACE	412
WDC_DBG_NONE	413
WDC_DBG_OUT_DBM	413
WDC_DBG_OUT_FILE	413
WDC_DMAGetGlobalHandle	413
WDC_DRV_OPEN_ALL	413
WDC_DRV_OPEN_BASIC	413
WDC_DRV_OPEN_CHECK_VER	414
WDC_DRV_OPEN_DEFAULT	414
WDC_DRV_OPEN_KP	414
WDC_DRV_OPEN_REG_LIC	414
WDC_ReadAddrBlock16	414
WDC_ReadAddrBlock32	414

WDC_ReadAddrBlock64	415
WDC_ReadAddrBlock8	415
WDC_ReadMem16	415
WDC_ReadMem32	416
WDC_ReadMem64	416
WDC_ReadMem8	416
Read/Write memory and I/O addresses	416
WDC_SIZE_16	416
WDC_SIZE_32	416
WDC_SIZE_64	416
WDC_SIZE_8	416
WDC_SLEEP_BUSY	417
WDC_SLEEP_NON_BUSY	417
WDC_WriteAddrBlock16	417
WDC_WriteAddrBlock32	417
WDC_WriteAddrBlock64	417
WDC_WriteAddrBlock8	418
WDC_WriteMem16	418
WDC_WriteMem32	418
WDC_WriteMem64	418
WDC_WriteMem8	418
Typedef Documentation	419
WDC_ADDR_SIZE	419
WDC_DBG_OPTIONS	419
WDC_DEVICE_HANDLE	419
WDC_DRV_OPEN_OPTIONS	419
WDC_SLEEP_OPTIONS	419
Enumeration Type Documentation	419
WDC_ADDR_MODE	419
WDC_ADDR_RW_OPTIONS	419
WDC_DIRECTION	420
Function Documentation	420
WDC_AddrSpacelsActive()	420
WDC_CallKerPlug()	420
Send Kernel Plugin messages	420
WDC_CardCleanupSetup()	421
Set card cleanup commands	421
WDC_DMABufGet()	421
WDC_DMABufUnlock()	422
WDC_DMAContigBufLock()	423
DMA (Direct Memory Access)	423
WDC_DMAReservedBufLock()	423

WDC_DMASGBufLock()	424
WDC_DMASyncCpu()	425
WDC_DMASynclo()	425
WDC_DMATransactionContigInit()	425
WDC_DMATransactionExecute()	426
WDC_DMATransactionRelease()	427
WDC_DMATransactionSGInit()	427
WDC_DMATransactionUninit()	428
WDC_DMATransferCompletedAndCheck()	428
WDC_DriverClose()	429
WDC_DriverOpen()	430
WDC_Err()	432
WDC_EventIsRegistered()	432
WDC_EventRegister()	433
Plug-and-play and power management events	433
WDC_EventUnregister()	435
WDC_GetBusType()	436
WDC_GetDevContext()	436
WDC_GetWDHandle()	437
WDC_IntDisable()	437
WDC_IntEnable()	437
WDC_IntIsEnabled()	439
WDC_IntType2Str()	439
WDC_IsaDeviceClose()	440
WDC_IsaDeviceOpen()	440
WDC_KernelPlugInOpen()	440
Open a handle to Kernel PlugIn driver	441
WDC_MultiTransfer()	441
WDC_PciDeviceClose()	441
WDC_PciDeviceOpen()	443
Open/Close device	443
WDC_PciGetDeviceInfo()	444
Get device's resources information (PCI)	444
WDC_PciGetExpressGen()	446
WDC_PciGetExpressGenBySlot()	446
WDC_PciGetExpressOffset()	446
Scan PCI Capabilities	446
WDC_PciGetHeaderType()	447
WDC_PciReadCfg()	447
WDC_PciReadCfg16()	447
WDC_PciReadCfg32()	448
WDC_PciReadCfg64()	448

WDC_PciReadCfg8()	448
WDC_PciReadCfgBySlot()	449
Access PCI configuration space	449
WDC_PciReadCfgBySlot16()	449
WDC_PciReadCfgBySlot32()	450
WDC_PciReadCfgBySlot64()	450
WDC_PciReadCfgBySlot8()	451
WDC_PciScanCaps()	451
WDC_PciScanCapsBySlot()	451
WDC_PciScanDevices()	452
Scan bus (PCI)	452
WDC_PciScanDevicesByTopology()	452
WDC_PciScanExtCaps()	453
WDC_PciScanRegisteredDevices()	453
WDC_PciSriovDisable()	453
WDC_PciSriovEnable()	454
Control device's SR-IOV capability (PCIe)	454
WDC_PciSriovGetNumVFs()	455
WDC_PciWriteCfg()	455
WDC_PciWriteCfg16()	456
WDC_PciWriteCfg32()	456
WDC_PciWriteCfg64()	456
WDC_PciWriteCfg8()	457
WDC_PciWriteCfgBySlot()	457
WDC_PciWriteCfgBySlot16()	458
WDC_PciWriteCfgBySlot32()	458
WDC_PciWriteCfgBySlot64()	458
WDC_PciWriteCfgBySlot8()	459
WDC_ReadAddr16()	459
WDC_ReadAddr32()	460
WDC_ReadAddr64()	460
WDC_ReadAddr8()	460
WDC_ReadAddrBlock()	461
WDC_SetDebugOptions()	461
Debugging and error handling	462
WDC_Sleep()	463
WDC_Trace()	464
WDC_Version()	464
WDC_WriteAddr16()	464
WDC_WriteAddr32()	465
WDC_WriteAddr64()	465
WDC_WriteAddr8()	466

WDC_WriteAddrBlock()	466
wdc_lib.h	466
wds_lib.h File Reference	472
Macro Definition Documentation	473
WDS_SharedBufferGetGlobalHandle	473
Typedef Documentation	474
IPC_MSG_RX_HANDLER	474
Function Documentation	474
WDS_IpcMulticast()	474
WDS_IpcRegister()	475
WDS_IpcScanProcs()	476
WDS_IpcSubGroupMulticast()	478
WDS_IpcUidUnicast()	478
WDS_IpcUnRegister()	480
WDS_IsIpcRegistered()	481
WDS_IsSharedIntsEnabledLocally()	482
WDS_SharedBufferAlloc()	482
WDS_SharedBufferFree()	483
WDS_SharedBufferGet()	483
WDS_SharedIntDisableGlobal()	483
WDS_SharedIntDisableLocal()	484
WDS_SharedIntEnable()	484
wds_lib.h	485
wdu_lib.h File Reference	486
Typedef Documentation	488
WDU_ATTACH_CALLBACK	488
WDU_DETACH_CALLBACK	489
WDU_DEVICE_HANDLE	489
WDU_DRIVER_HANDLE	489
WDU_LANGID	489
WDU_POWER_CHANGE_CALLBACK	489
WDU_STREAM_HANDLE	490
Function Documentation	490
WDU_GetDeviceAddr()	490
WDU_GetDeviceInfo()	490
WDU_GetDeviceRegistryProperty()	491
WDU_GetLangIDs()	492
WDUGetStringDesc()	492
WDU_HaltTransfer()	493
WDU_Init()	493
WDU_PutDeviceInfo()	494
WDU_ResetDevice()	494

WDU_ResetPipe()	494
WDU_SelectiveSuspend()	495
WDU_SetConfig()	495
WDU_SetInterface()	495
WDU_StreamClose()	496
WDU_StreamFlush()	496
WDU_StreamGetStatus()	497
WDU_StreamOpen()	498
WDU_StreamRead()	499
WDU_StreamStart()	500
WDU_StreamStop()	501
WDU_StreamWrite()	501
WDU_Transfer()	502
WDU_TransferBulk()	503
WDU_TransferDefaultPipe()	503
WDU_TransferInterrupt()	503
WDU_TransferIsoch()	503
WDU_Uninit()	504
WDU_Wakeup()	504
wdu_lib.h	504
windrvr.h File Reference	506
Macro Definition Documentation	516
_ALIGN_DOWN	516
_ALIGN_UP	516
FUNCTION	516
In	516
Inout	516
Out	516
Outptr	517
BZERO	517
CTL_CODE	517
DEBUG_USER_BUF_LEN	517
DLLCALLCONV	517
DMA_ADDRESS_WIDTH_MASK	517
DMA_BIT_MASK	517
DMA_DIRECTION_MASK	517
DMA_OPTIONS_ADDRESS_WIDTH_SHIFT	518
DMA_OPTIONS_ALL	518
DMA_READ_FROM_DEVICE	518
DMA_WRITE_TO_DEVICE	518
FILE_ANY_ACCESS	518
FILE_READ_ACCESS	518

FILE_WRITE_ACCESS	518
FUNC_MASK	518
INSTALLATION_TYPE_NOT_DETECT_TEXT	519
INVALID_HANDLE_VALUE	519
IOCTL_WD_CARD_CLEANUP_SETUP	519
IOCTL_WD_CARD_REGISTER	519
IOCTL_WD_CARD_UNREGISTER	519
IOCTL_WD_DEBUG	519
IOCTL_WD_DEBUG_ADD	519
IOCTL_WD_DEBUG_DUMP	519
IOCTL_WD_DMA_LOCK	519
IOCTL_WD_DMA_SYNC_CPU	519
IOCTL_WD_DMA_SYNC_IO	520
IOCTL_WD_DMA_TRANSACTION_EXECUTE	520
IOCTL_WD_DMA_TRANSACTION_INIT	520
IOCTL_WD_DMA_TRANSACTION_RELEASE	520
IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK	520
IOCTL_WD_DMA_UNLOCK	520
IOCTL_WD_EVENT_PULL	520
IOCTL_WD_EVENT_REGISTER	520
IOCTL_WD_EVENT_SEND	520
IOCTL_WD_EVENT_UNREGISTER	520
IOCTL_WD_GET_DEVICE_PROPERTY	521
IOCTL_WD_INT_COUNT	521
IOCTL_WD_INT_DISABLE	521
IOCTL_WD_INT_ENABLE	521
IOCTL_WD_INT_WAIT	521
IOCTL_WD_IPC_REGISTER	521
IOCTL_WD_IPC_SCAN_PROCS	521
IOCTL_WD_IPC_SEND	521
IOCTL_WD_IPC_SHARED_INT_DISABLE	521
IOCTL_WD_IPC_SHARED_INT_ENABLE	521
IOCTL_WD_IPC_UNREGISTER	522
IOCTL_WD_KERNEL_BUF_LOCK	522
IOCTL_WD_KERNEL_BUF_UNLOCK	522
IOCTL_WD_KERNEL_PLUGIN_CALL	522
IOCTL_WD_KERNEL_PLUGIN_CLOSE	522
IOCTL_WD_KERNEL_PLUGIN_OPEN	522
IOCTL_WD_LICENSE	522
IOCTL_WD_MULTI_TRANSFER	522
IOCTL_WD_PCI_CONFIG_DUMP	522
IOCTL_WD_PCI_GET_CARD_INFO	522

IOCTL_WD_PCI_SCAN_CAPS	523
IOCTL_WD_PCI_SCAN_CARDS	523
IOCTL_WD_PCI_SRIOV_DISABLE	523
IOCTL_WD_PCI_SRIOV_ENABLE	523
IOCTL_WD_PCI_SRIOV_GET_NUMVFS	523
IOCTL_WD_SLEEP	523
IOCTL_WD_TRANSFER	523
IOCTL_WD_USAGE	523
IOCTL_WD_VERSION	523
IOCTL_WDU_GET_DEVICE_DATA	523
IOCTL_WDU_HALT_TRANSFER	524
IOCTL_WDU_RESET_DEVICE	524
IOCTL_WDU_RESET_PIPE	524
IOCTL_WDU_SELECTIVE_SUSPEND	524
IOCTL_WDU_SET_INTERFACE	524
IOCTL_WDU_STREAM_CLOSE	524
IOCTL_WDU_STREAM_FLUSH	524
IOCTL_WDU_STREAM_GET_STATUS	524
IOCTL_WDU_STREAM_OPEN	524
IOCTL_WDU_STREAM_START	524
IOCTL_WDU_STREAM_STOP	525
IOCTL_WDU_TRANSFER	525
IOCTL_WDU_WAKEUP	525
KPRI	525
MAX	525
METHOD_BUFFERED	525
METHOD_IN_DIRECT	525
METHOD_NEITHER	525
METHOD_OUT_DIRECT	525
MIN	526
OS_CAN_NOT_DETECT_TEXT	526
PAD_TO_64	526
PRI64	526
REGKEY_BUFSIZE	526
SAFE_STRING	526
SIZE_OF_WD_DMA	526
SIZE_OF_WD_EVENT	526
strcmp	527
UNUSED_VAR	527
UPRI	527
va_copy	527
WD_ACTIONS_ALL	527

WD_ACTIONS_POWER	527
WD_CardCleanupSetup	527
WD_CardRegister	528
WD_CardUnregister	529
WD_Close	529
WD_CloseLocal	529
WD_CPU_SPEC	529
WD_CTL_CODE	529
WD_CTL_DECODE_FUNC	530
WD_CTL_DECODE_TYPE	530
WD_CTL_IS_64BIT_AWARE	530
WD_DATA_MODEL	530
WD_Debug	530
WD_DebugAdd	530
WD_DebugDump	531
WD_DEFAULT_DRIVER_NAME	531
WD_DEFAULT_DRIVER_NAME_BASE	531
WD_DMALock	532
WD_DMASyncCpu	533
WD_DMASynclo	533
WD_DMATransactionExecute	534
WD_DMATransactionInit	534
WD_DMATransactionRelease	535
WD_DMATransactionUninit	535
WD_DMATransferCompletedAndCheck	535
WD_DMAUnlock	536
WD_DRIVER_NAME	536
WD_DRIVER_NAME_PREFIX	536
WD_EventPull	536
WD_EventRegister	536
WD_EventSend	537
WD_EventUnregister	537
WD_FUNCTION	537
WD_GetDeviceProperty	537
WD_IntCount	537
WD_IntDisable	537
WD_IntEnable	538
WD_IntWait	539
WD_IPC_ALL_MSG	539
WD_IpcRegister	539
WD_IpcScanProcs	540
WD_IpcSend	540

WD_IpcUnRegister	540
WD_KernelBufLock	541
WD_KernelBufUnlock	541
WD_KernelPlugInCall	543
WD_KernelPlugInClose	543
WD_KernelPlugInOpen	545
WD_License	545
WD_LICENSE_LENGTH	546
WD_MAX_DRIVER_NAME_LENGTH	546
WD_MAX_KP_NAME_LENGTH	546
WD_MultiTransfer	546
WD_Open	547
WD_OpenLocal	547
WD_OpenStreamLocal	547
WD_PciConfigDump	548
WD_PciGetCardInfo	548
WD_PciScanCaps	549
WD_PciScanCards	549
WD_PciSriovDisable	550
WD_PciSriovEnable	550
WD_PciSriovGetNumVFs	551
WD_PROCESS_NAME_LENGTH	551
WD_PROD_NAME	551
WD_SharedIntDisable	551
WD_SharedIntEnable	552
WD_Sleep	552
WD_StreamClose	552
WD_StreamOpen	553
WD_Transfer	553
WD_TYPE	553
WD_UGetDeviceData	553
WD_UHaltTransfer	553
WD_UResetDevice	554
WD_UResetPipe	554
WD_Usage	554
WD_USelectiveSuspend	554
WD_USetInterface	554
WD_UStreamClose	554
WD_UStreamFlush	555
WD_UStreamGetStatus	555
WD_UStreamOpen	555
WD_UStreamRead	555

WD_UStreamStart	555
WD_UStreamStop	555
WD_UStreamWrite	556
WD_UTransfer	556
WD_UWakeups	556
WD_VER_STR	556
WD_Version	556
WD_VERSION_STR_LENGTH	557
WIN32	557
WINAPI	557
Typedef Documentation	557
BYTE	557
DMA_ADDR	557
DMA_TRANSACTION_CALLBACK	557
KPTR	557
PHYS_ADDR	557
UINT32	557
UINT64	558
UPTR	558
WD_BUS_TYPE	558
WD_BUS_V30	558
WD_CARD_REGISTER_V118	558
WD_CARD_V118	558
WD_DEBUG_ADD_V503	558
WD_DEBUG_DUMP_V40	558
WD_DEBUG_V40	558
WD_DMA_PAGE_V80	558
WD_DMA_V80	558
WD_EVENT_TYPE	559
WD_EVENT_V121	559
WD_INTERRUPT_V91	559
WD_IPC_PROCESS_V121	559
WD_IPC_REGISTER_V121	559
WD_IPC_SCAN_PROCS_V121	559
WD_IPC_SEND_V121	559
WD_ITEMS_V118	559
WD_KERNEL_BUFFER_V121	559
WD_KERNEL_PLUGIN_CALL_V40	559
WD_KERNEL_PLUGIN_V40	559
WD_LICENSE_V122	559
WD_PCI_CARD_INFO_V118	560
WD_PCI_CONFIG_DUMP_V30	560

WD_PCI_SCAN_CAPS_V118	560
WD_PCI_SCAN_CARDS_V124	560
WD_PCI_SRIOV_V122	560
WD_SLEEP_V40	560
WD_TRANSFER_V61	560
WD_VERSION_V30	560
WORD	560
Enumeration Type Documentation	560
anonymous enum	560
anonymous enum	561
anonymous enum	561
anonymous enum	561
anonymous enum	561
anonymous enum	561
anonymous enum	561
anonymous enum	562
anonymous enum	562
anonymous enum	562
anonymous enum	562
anonymous enum	563
anonymous enum	563
anonymous enum	563
anonymous enum	563
DEBUG_COMMAND	563
DEBUG_LEVEL	564
DEBUG_SECTION	564
ITEM_TYPE	565
PCI_ACCESS_RESULT	565
WD_DMA_OPTIONS	565
WD_ERROR_CODES	566
WD_EVENT_ACTION	570
WD_EVENT_OPTION	571
WD_GET_DEVICE_PROPERTY_OPTION	571
WD_INTERRUPT_WAIT_RESULT	571
WD_ITEM_MEM_OPTIONS	571
WD_KER_BUF_OPTION	572
WD_PCI_SCAN_CAPS_OPTIONS	572
WD_PCI_SCAN_OPTIONS	572
WD_TRANSFER_CMD	572
Function Documentation	573
check_secureBoot_enabled()	573
get_os_type()	574
WD_DriverName()	574

windrivr.h	576
windrivr_32bit.h File Reference	594
Typedef Documentation	594
ptr32	594
windrivr_32bit.h	594
windrivr_events.h File Reference	597
Typedef Documentation	597
event_handle_t	597
EVENT_HANDLER	597
Function Documentation	597
EventAlloc()	598
EventDup()	598
EventFree()	598
EventRegister()	598
EventUnregister()	598
PciEventCreate()	598
UsbEventCreate()	598
windrivr_events.h	598
windrivr_int_thread.h File Reference	599
Typedef Documentation	599
INT_HANDLER	599
INT_HANDLER_FUNC	599
Function Documentation	599
InterruptDisable()	600
InterruptEnable()	600
windrivr_int_thread.h	600
windrivr_usb.h File Reference	600
Macro Definition Documentation	602
PAD_TO_64	602
PAD_TO_64_PTR_ARR	602
WD_USB_MAX_ALT_SETTINGS	602
WD_USB_MAX_ENDPOINTS	602
WD_USB_MAX_INTERFACES	602
WD_USB_MAX_PIPE_NUMBER	602
WDU_CONFIG_DESC_TYPE	602
WDU_DEVICE_DESC_TYPE	602
WDU_ENDPOINT_ADDRESS_MASK	602
WDU_ENDPOINT_DESC_TYPE	603
WDU_ENDPOINT_DIRECTION_IN	603
WDU_ENDPOINT_DIRECTION_MASK	603
WDU_ENDPOINT_DIRECTION_OUT	603
WDU_ENDPOINT_TYPE_MASK	603

WDU_GET_MAX_PACKET_SIZE	603
WDU_INTERFACE_DESC_TYPE	603
WDU_STRING_DESC_STRING	603
Typedef Documentation	603
WDU_REGISTER_DEVICES_HANDLE	603
Enumeration Type Documentation	604
anonymous enum	604
USB_DIR	604
USB_PIPE_TYPE	604
WD_DEVICE_REGISTRY_PROPERTY	604
WDU_DIR	605
WDU_SELECTIVE_SUSPEND_OPTIONS	605
WDU_WAKEUP_OPTIONS	606
windrvr_usb.h	606
kp_pci.c File Reference	610
Macro Definition Documentation	611
DRIVER_VER_STR	611
PTR32	611
USE_MULTI_TRANSFER	611
Function Documentation	611
KP_Init()	612
KP_PCI_Call()	612
KP_PCI_Close()	612
KP_PCI_Event()	613
KP_PCI_IntAtDpc()	613
KP_PCI_IntAtDpcMSI()	614
KP_PCI_IntAtIrql()	614
KP_PCI_IntAtIrqlMSI()	615
KP_PCI_IntDisable()	616
KP_PCI_IntEnable()	616
KP_PCI_Open()	617
KP_PCI_Open_32_64()	617
kp_pci.c	618
Index	625

Chapter 1

Overview

This chapter outlines general WinDriver information, architecture, supported busses and operating systems, as well as what the toolkit package includes.

** Note**

This manual outlines WinDriver's support for PCI/ISA/EISA/CompactPCI/PCI Express devices as well as the Universal Serial Bus (USB).

1.1. Introduction

WinDriver is a complete development solution that dramatically simplifies the difficult task of creating high-performance custom device drivers and hardware-access applications that is on the market since 1998.

WinDriver includes a DriverWizard - a graphical diagnostics tool - that automatically detects your hardware, and using which you can generate skeletal diagnostics code for your device, providing you with convenient wrapper functions that utilize WinDriver's API to communicate with your specific device. Furthermore, you can review various samples provided with WinDriver to see if there is a sample that may serve as a skeletal basis for your driver application. The driver and application you develop using WinDriver are source code compatible across all supported operating systems. WinDriver allows you to create your driver in the familiar user-mode environment, using MS Visual Studio, CMake, GCC, Windows GCC, or any other appropriate compiler or development environment, generating the driver code for the project in C, C#, Java and Python.

WinDriver supports development for all PCI/ISA/EISA/CompactPCI/PCI Express chipsets, along with all USB chipsets, and offers enhanced support for specific chipsets, for instance Xilinx, on Windows, Linux and Mac OS.

WinDriver enables all development to be done in the user mode, utilizing our low-level kernel driver(s), and thus frees you of the need for any kernel and/or OS-specific driver development knowledge, such as familiarity with the Windows Driver Kit (WDK).

Don't let the size of this manual fool you. WinDriver makes developing device drivers an easy task that takes hours instead of months, providing stability and technical support both during the development and post. A part of this manual deals with the features that WinDriver offers to advanced users. However, most developers will find that reading these chapters and glancing through the DriverWizard and function-reference chapters is all they need to successfully write their driver.

[Understanding Device Drivers](#) gives a general overview of device drivers.

[Installing WinDriver](#) explains WinDriver installation and uninstallation steps.

[PCI Express Overview](#) and [USB Overview](#) - PCIe and USB overview, architecture and other details respectively.

[Using DriverWizard](#) provides detailed driver development guidance using DriverWizard.

[Developing a Driver](#) and [Debugging Drivers](#) deal with driver development and debugging.

[Enhanced Support for Specific Chipsets](#) focuses on special support for Altera Qsys, Avalon-MM designs; Xilinx BMD, XDMA designs.

[PCI Advanced Features](#) describes special PCI features like DMA transfer, interrupt handling and much more.

[Improving PCI Performance](#) talk about performance enhancement for already written and debugged drivers.

[Understanding the Kernel Plugin](#) and [Creating a Kernel Plugin Driver](#) - Kernel Plugin chapter, a special WinDriver feature that allows critical sections of the driver code to be moved to the kernel in order to enhance its performance even more.

[USB Advanced Features](#) dives into USB data transfer matters.

[Distributing Your Driver](#) and [Dynamically Loading Your Driver](#) guide through WinDriver-written drivers distribution process as well as an option of creating a dynamically loadable driver, which enables your customers to start your

application immediately after installing it, without the need for reboot a computer.

[Driver Installation - Advanced Issues](#) clarifies such important points as INF creation, drivers digital signature and some other.

Data Structures and Files are WinDriver APIs information.

Visit [Jungo's] (<https://www.jungo.com>) website for the latest news about WinDriver and other products that Jungo offers.

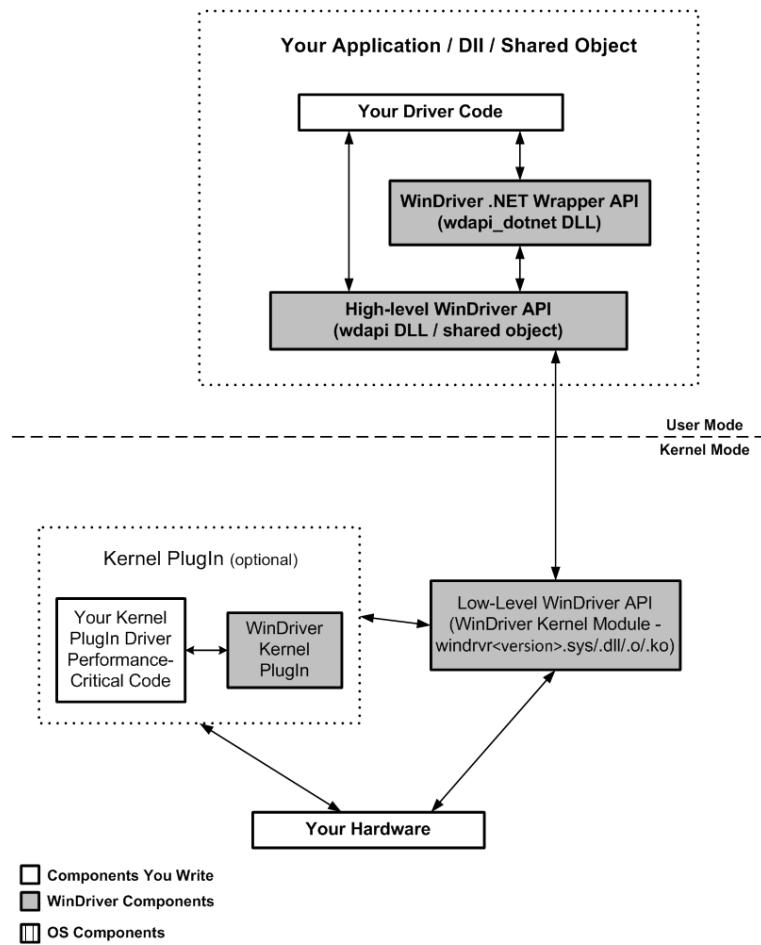
1.2. Main Features

WinDriver features include:

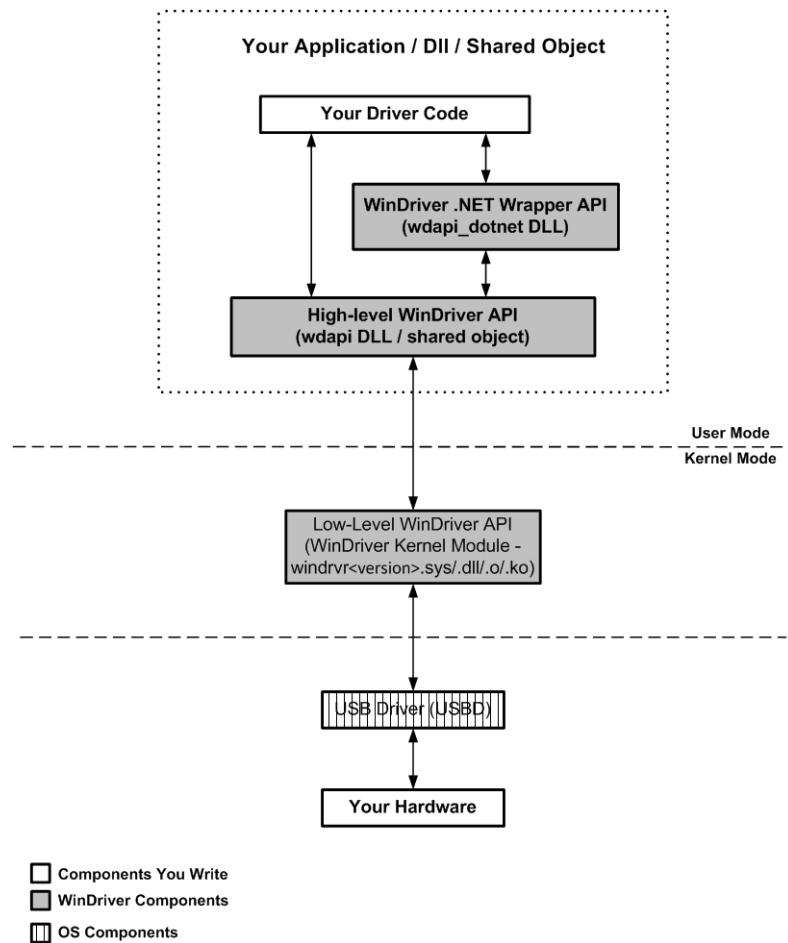
- Hardware diagnostics by DriverWizard.
- Automatic generation and detailed samples of the driver code in C, C#, Visual Basic.NET, PowerShell, Java and Python.
- Enhanced support for specific chipsets.
- Binary compatible applications.
- Source code applications compatible across all supported operating systems.
- Common IDE support.
- No WDK, ETK, DDI or any system-level programming knowledge required.
- Kernel-Mode Performance.
- I/O, DMA, interrupt handling and access to memory-mapped cards.
- Multiple CPUs and multiple PCI bus platforms support.
- Any USB device support, regardless of manufacturer.
- 64-bit PCI data transfers support.
- Dynamic driver loader.
- Comprehensive documentation and help files.
- Microsoft Windows Certification Program compliant.
- Technical support.
- No run-time fees or royalties.

1.3. Architecture

PCI WinDriver Architecture



USB WinDriver Architecture



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

1.4. Supported Platforms

WinDriver supports the following operating systems:

- Windows 11, 10, 8/8.1, Windows 10 IoT, Server 2016, Server 2012 R2, Server 2012 - henceforth collectively: Windows.
- Linux kernel version 2.6.x or higher, Linux ARM
- MacOS 10.10-12.

The same source code will run on all supported platforms - simply recompile it for the target platform. The source code is binary compatible across Windows 11/10/Server 2016/8.1/Server 2012 R2/8/Server 2012. WinDriver executables can be ported among the binary-compatible platforms without recompilation.

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.

** Attention**

OS-specific support is provided only for operating systems with official vendor support. In order to see which operating system and its version are supported by the vendor, please refer to the vendor's website respectively.

1.5. Linux ARM

WinDriver PCI/USB for Linux ARM automates and simplifies the development of Linux ARM device drivers for PCIe/PCI-X/PCI-104/CompactPCI and USB.

WinDriver was built and tested on the following platforms and OSes. Jungo can offer support for other platforms upon request sent to WinDriver@jungo.com

- ARM Cortex A7 – Broadcom: Raspberry PI 2/3/4 : running Raspbian Stretch with kernel version 4.14.79-v7+. PI 4b: running Raspbian with kernel version 4.19.75-v7l+
- ARM Cortex A9 – Solid Run: I.MX6 MicroSoM i2 (Hummingboard2) running ARMBian's Ubuntu Xenial Desktop Kernel Version 4.11.6-cubox
- ARM Cortex A9 – Boundary: BL-SL-I.MX6 (formerly Sabre LITE) running Debian nitrogen kernel version 4.9.88-6-boundary-14s
- ARM Cortex A15 – NVidia: Jetson TK1 running Ubuntu 16.04 kernel version 3.10.40-g8c4516e
- ARM64 Cortex A53 – Broadcom: Raspberry PI 3b+ – running 64bit Xubuntu 18.04 kernel version 4.15.0-1010-raspi2 or Ubuntu Server 18.04.2 kernel version 4.15.0-1033-raspi2
- ARM64 Cortex A57 – NVidia: Drive PX2 running Ubuntu 18.04 kernel version 4.9.38-rt25-tegra
- ARM64 Cortex A57 – NVidia: Jetson TX2 running Ubuntu 16.04 kernel version 4.4.38-tegra
- ARM64 v8.2 – NVidia: Jetson NX Xavier / Jetson AGX Xavier running Ubuntu 18.04 kernel version 4.9-tegra

1.6. Operating Systems Support Overview

All version information in this document relates to both WinDriver for PCI/ISA and WinDriver for USB, unless otherwise specified.

USB 2.0 is supported beginning with version 5.0.4.

USB 3.0 is supported beginning with version 11.4.0.

USB 3.1 is supported beginning with version 14.2.0.

Operating System	Supported WinDriver Versions
Windows 10 IoT Core	<i>Windows 10 IoT Core</i> (x32/x64/ARM) is supported from version 12.7.0 until version 14.7.0.
Windows Server 2022	<i>Windows Server 2022</i> is supported from version 15.1.0.
Windows Server 2019	<i>Windows Server 2019</i> is supported from version 14.2.1.
Windows Server 2016	<i>Windows Server 2016</i> is supported from version 12.0.0.
Windows 11	<i>Windows 11 x64</i> is supported from version 15.0.0.
Windows 10	<i>Windows 10 x32 and x64</i> are supported from version 12.0.0. <i>Windows 10 Technical Preview build 10162</i> was supported in version 11.9.0.
Windows 8.1	<i>Windows 8.1 x32 and x64</i> are supported from version 11.4.0.
Windows Server 2012 R2	<i>Windows Server 2012 x32 and x64</i> are supported from version 11.4.0.
Windows 8	<i>Windows 8 x32 and x64</i> are supported from version 11.0.0.
Windows Server 2012	<i>Windows Server 2012 x32 and x64</i> are supported from version 11.0.0.
Windows 7	<i>Windows 7 x32 and x64</i> are supported from version 10.1.0. to version 14.3.0. <i>Windows 7 Service Pack 1</i> is supported from version 10.3.1 to version 14.3.0.
Windows Server 2008 R2	<i>Windows Server 2008 R2 x32 and x64</i> are supported from version 10.1.0 to version 14.3.0. <i>Windows Server 2008 R2 Service Pack 1</i> is supported from version 10.3.1 to version 14.3.0.
Windows Vista	<i>Windows Vista x32 and x64</i> are supported from version 8.1.0 to version 12.2.1. <i>Windows Vista Service Pack 1</i> is supported from version 10.0.0 to version 12.2.1.

Operating System	Supported WinDriver Versions
Windows Server 2008	<i>Windows Server 2008 x32 and x64</i> are supported from version 10.0.0 to version 14.3.0.
Windows Server 2003	<i>Windows Server 2003 x64</i> is supported from version 8.0.0 to version 12.2.1. <i>Windows Server 2003 x32</i> is supported from version 6.0.2 to version 12.2.1.
Windows XP	<i>Windows XP x64</i> is supported from version 8.0.0 to version 12.2.1. <i>Windows XP Service Pack 3</i> is supported from version 10.0.0 to version 12.2.1.
Windows CE	<i>Windows Embedded Compact 2013 (a.k.a. Windows CE 8, WEC2013 or Windows CE 2013)</i> is supported on x86 and ARM platforms from version 11.7.0 to version 12.3.0. <i>Windows Embedded Compact 7 (a.k.a. WEC7 or Windows CE 7)</i> is supported from version 10.4.0 to version 12.3.0. <i>Windows Embedded CE 6.0</i> is supported from version 9.0.0 to version 12.3.0. <i>Windows CE 5.0</i> is supported from version 6.2.2 to version 12.3.0. <i>Windows CE 4.x</i> is supported from version 6.0.0 to 12.3.0. <i>Windows CE 2.x–3.x</i> is supported for PCI/ISA from version 4.1.2 to 6.0.3.
MacOS (PCI)	<i>MacOS 10.5 + 10.6</i> is supported for PCI from version 10.2.0 to version 10.4.0.
MacOS (PCIe + USB)	<i>MacOS 10.10-10.15</i> are supported from version 14.5.0. <i>MacOS 10.13-11.0</i> are supported from version 14.6.0. <i>MacOS 11.0 and MacOS 12.0 (ARM64 M1 version)</i> are supported from version 14.8.0.
Linux	<i>PCI/ISA</i> is supported on Linux from version 4.1.2. <i>USB</i> is supported on Linux from version 6.0.0.
Solaris (PCI/ISA)	<i>Solaris</i> is supported for PCI/ISA from version 4.1.4 to version 9.0.1.

** Note**

If you have any questions or would like to evaluate WinDriver on a specific operating system, that is either not listed above, or is supported until a particular version, please send an e-mail to WinDriver@jungo.com, and our team will assist you.

1.7. Limitations of Different Evaluation Versions

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

- Each time WinDriver is activated, an **Unregistered** message appears.
- The Windows evaluation version expires 30 days from the date of installation. When using DriverWizard, a dialogue box with a message stating the remaining evaluation version time will appear on every interaction with the hardware.
- On Linux and MacOS the driver will remain operational for 60 minutes, and will require restarting every time when 60 minutes run out.
- During the evaluation there is free basic technical support available via the support center.

** Note**

If you are interested in purchasing a WinDriver license, wonder what it includes and its conditions, or have any other question or feedback, please send us an e-mail to sales@jungo.com, and our team will respond swiftly. Please note that by offering any suggestions to Jungo, you give Jungo full permission to use them freely.

1.7.1. Limitations on MacOS

Currently, WinDriver for MacOS does not support the following features:

- Out-of-the-box Driver redistribution / Driver renaming.
- Kernel Plugin.
- Server APIs (IPC, Shared Kernel Buffer).
- Some DMA buffer locking flags (Address width specification, preallocation, reserved memory).

Jungo strives to support all these features in the nearest future as well.

1.8. What WinDriver includes

The main files of WinDriver toolkit are:

- **windrvr.h**: declarations and definitions of WinDriver's basic API.
- **wdu_lib.h**: declarations and definitions of the WinDriver USB (WDU) library, which provides convenient wrapper USB APIs.
- **windrvr_int_thread.h**: declarations of convenient wrapper functions to simplify interrupt handling.
- **windrvr_events.h**: declarations of APIs for handling 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 application that diagnoses your hardware and enables you to easily generate code for your driver.
- **Debug Monitor**: a debugging tool that collects information about your driver as it runs. This tool is available both as a fully graphical application - WinDriver/util/wddebug_gui - and as a console-mode application - WinDriver/util/wddebug.
- **WinDriver distribution package (WinDriver/redist)**: the files you include in the driver distribution to customers.
- **This manual**: the full WinDriver manual (this document), in different formats, can be found under the WinDriver/docs directory.

WinDriver also includes the following utilities:

- **usb_diag.exe** (WinDriver/util/usb_diag.exe): enables the user to view the resources of connected USB devices and communicate with the devices - transfer data to/from the device, set the active alternate setting, reset pipes, etc. On Windows the program identifies all devices that have been registered to work with WinDriver using an INF file. On the other supported operating systems the program identifies all USB devices connected to the target platform.
- **pci_dump.exe** (WinDriver/util/pci_dump.exe): used to obtain a dump of the PCI configuration registers of the installed PCI cards.
- **pci_scan.exe** (WinDriver/util/pci_scan.exe): used to obtain a list of the PCI cards installed and the resources allocated for each card.

There are also various code samples available:

- **C samples**: found under the WinDriver/samples/c directory.

- **Python samples:** found under the `WinDriver/samples/python` directory.
- **Java samples:** found under the `WinDriver/samples/java` directory.
- **C#.NET samples :** found under the `WinDriver/samples/csharp.net` directory.
- **PowerShell samples :** found under the `WinDriver/samples/powershell` directory.
- **Visual Basic.NET samples (Windows):** found under the `WinDriver/samples/vb.net` directory.

In addition to the generic samples described above, WinDriver provides custom wrapper APIs and sample code for major PCI chipsets, as outlined in [Enhanced Support for Specific Chipsets](#). The relevant files are provided in the following WinDriver installation directories:

- **PLX 6466, 9030, 9050, 9052, 9054, 9056, 9080 and 9656** - `WinDriver/samples/c/plx`
- **Altera Qsys design** - `WinDriver/samples/c/altera/qsys_design`
- **Altera Avalon Memory Mapped (Avalon-MM) design** - `WinDriver/samples/c/altera/avalonmm`
- **Xilinx Bus Master DMA (BMD) design** - `WinDriver/samples/c/xilinx/bmd_design`
- **Xilinx XDMA design** - `WinDriver/samples/c/xilinx/xdma`
- **Xilinx QDMA design** - `WinDriver/samples/c/xilinx/qdma`

For the Xilinx BMD, XDMA, QDMA and Altera Qsys designs there is also an option to generate customized driver code that utilizes the related enhanced-support APIs.

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.

** Attention**

When using WinDriver you do not need to familiarize yourself with the internal workings of driver development. As explained in [1.1. Introduction](#), 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 provide 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.

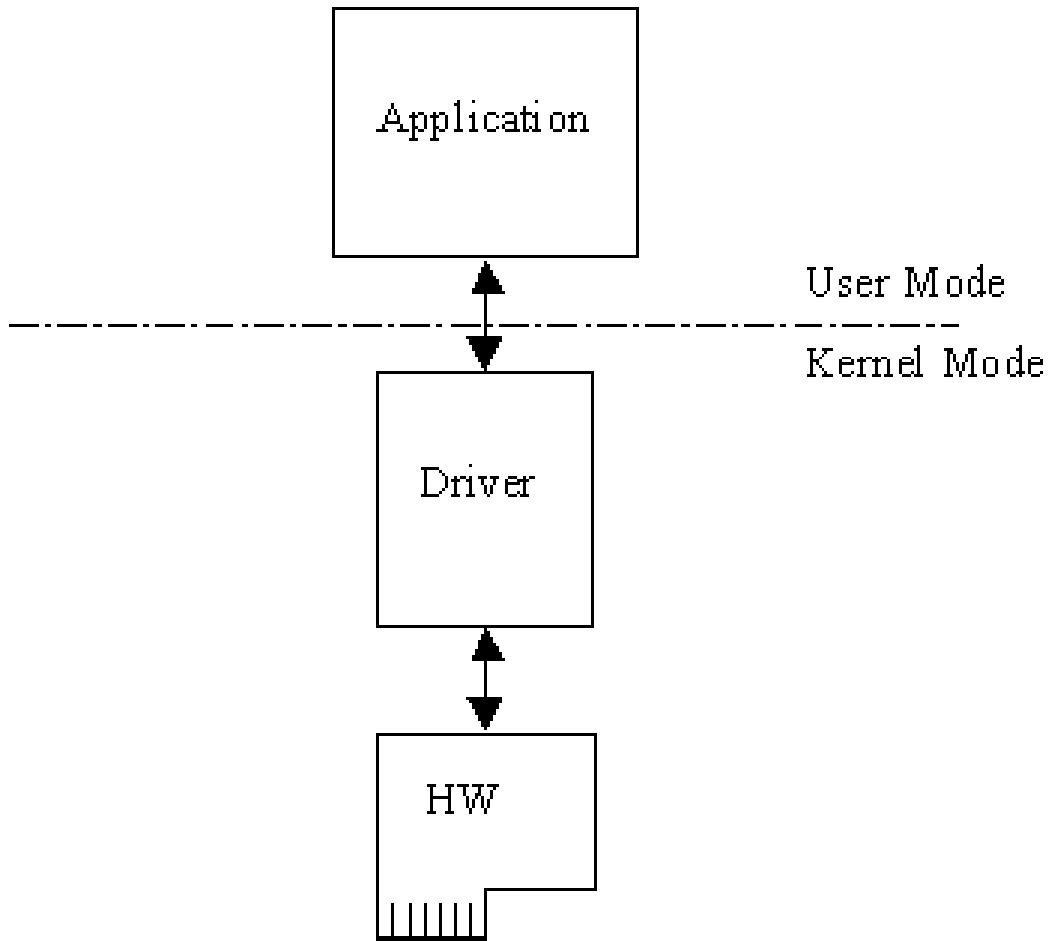
A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and 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 WDK, ETK, DDI/DKI functions.

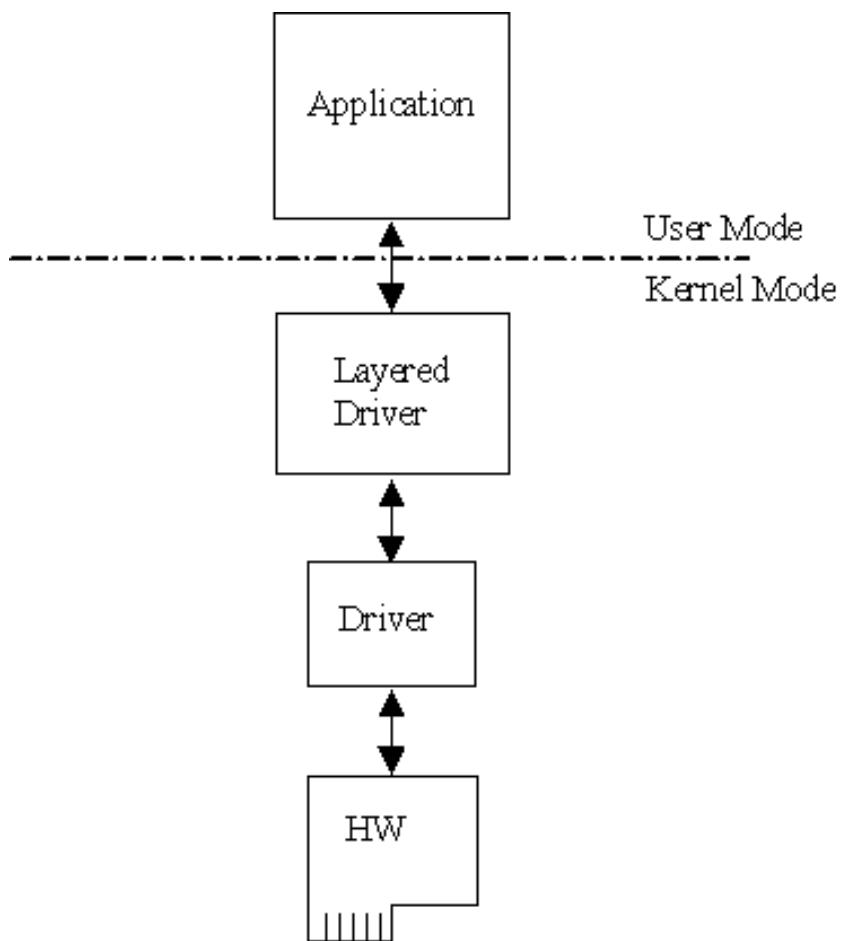


Monolithic drivers are supported in all operating systems including all Windows platforms and all Unix platforms.

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.

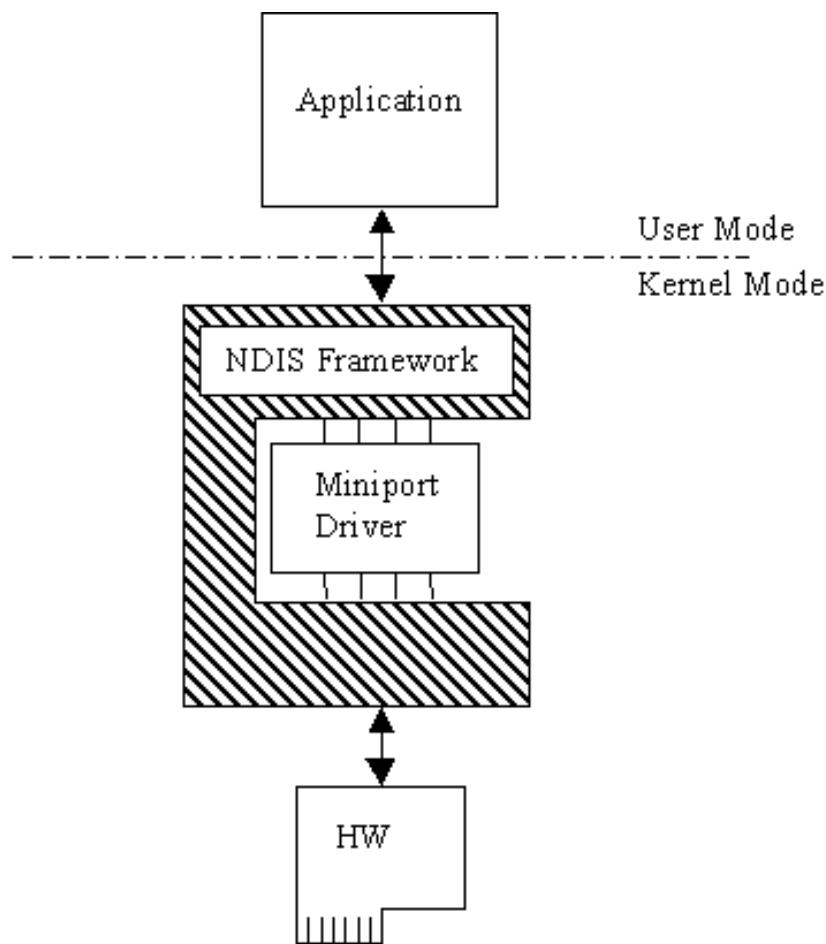
Layered drivers are sometimes also known as filter drivers, and are supported in all operating systems including all Windows platforms and all Unix platforms.



2.2.3. Miniport Drivers

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 7 and higher operating systems.



Windows 7 and higher operating systems 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 framework is used to create network drivers that hook up to Windows's communication stacks, and are therefore accessible to common communication calls used by applications. The Windows 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

Windows Driver Model (WDM) drivers are kernel-mode drivers within the Windows operating systems. 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. WDF Drivers

The Windows Driver Foundation (WDF) is a wrapper around Microsoft Windows Driver Model (WDM) interfaces and is the preferred way to implement Windows drivers today. WDF is a set of Microsoft tools and libraries that aid in the creation of device drivers for Windows. It abstracts away much of the complexity of writing Windows drivers.

** Attention**

From version 5.2.0 to 14.1.1 WinDriver was a WDM driver. From version 14.1.1 WinDriver is a full WDF driver.

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.

2.4. 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 with the Driver

Operating systems differ in the ways they associate a device with a specific driver.

In Windows, the hardware-driver association is performed via an 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, checks its database to identify which INF file is associated with the device, and according to the INF file, calls the driver's entry point.

In Linux, the hardware-driver association is defined in the driver's `init_module()` routine. This routine includes a callback that indicates which 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

Communication between a user-mode application and the driver that drives the hardware, is implemented differently for each operating system, using the custom OS Application Programming Interfaces (APIs).

On Windows, and Linux, the application can use the OS file-access API to open a handle to the driver (e.g., using the Windows `CreateFile()` function or using the Linux `open()` function), and then read and write from/to the device by passing the handle to the relevant OS file-access functions (e.g., the Windows `ReadFile()` and `WriteFile()` functions, or the Linux `read()` and `write()` functions).

The application sends requests to the driver via I/O control (IOCTL) calls, using the custom OS APIs provided for this purpose (e.g., the Windows `DeviceIoControl()` function, or the Linux `ioctl()` function).

The data passed between the driver and the application via the IOCTL calls is encapsulated using custom OS mechanisms. For example, on Windows the data is passed via an I/O Request Packet (IRP) structure, and is encapsulated by the I/O Manager.

Chapter 3

Installing WinDriver

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

To find out how to install a driver you created on target platforms, refer to [Distributing Your Driver](#).

3.1. System Requirements

You can find below system requirements for Windows, Linux, and Mac.

3.1.1. Windows System Requirements

Please make sure that your development machine fits the following requirements:

- Any x86 32-bit or 64-bit (x64: AMD64 or Intel EM64T) processor, ARM processors supported by Microsoft.
- Any compiler or development environment supporting C, .NET, Java or Python.

3.1.2. Linux System Requirements

Any of the following processor architectures, with a 2.6.x or higher Linux kernel:

- 32-bit x86.
- 64-bit x86 AMD64 or Intel EM64T (x86_64)
- ARM Cortex-A7, A9, A15, A53, A57.

** Note**

Jungo strives to support new Linux kernel versions as close as possible to their release. To find out the latest supported kernel version, refer to the WinDriver release notes that can be found online at [WinDriver Release Notes](#).

- A GCC compiler. 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: **glibc2.14.x (or newer)**.
- The following libraries are required for running GUI WinDriver application (e.g., DriverWizard (see more in [Using DriverWizard](#)); Debug Monitor (see more in [8.2. Debug Monitor](#)):
 - libstdc++.so.6
 - libpng12.so.0
 - libQtGui.so.4
 - libQtCore.so.4
 - libQtNetwork.so.4
 - libQt5Gui.so.5.9.5

- libQt5Core.so.5.9.5
- libQt5Network.so.5.9.5
- make, gcc, flex, bison (for installing WinDriver and Kernel Plugins)
- The kernel source of the running Linux kernel Optional dependencies:
- gcc-multilib (for compiling 32-bit applications on 64-bit systems)
- CMake (for compiling user mode samples)

Newer versions of Linux distributions, provide native support of Qt5 instead of Qt4. If your OS supports Qt4 instead of Qt5, use `wdwizard_legacy`, `wddebug_gui_legacy`, `xdma_gui_legacy` instead of the non-prefixed versions. Check out [3.2.2.1. Preparing the System for Installation](#) for instructions how to install the above requirements on various Linux distributions.

3.1.3. MacOS System Requirements

Please make sure that your development machine fits the following requirements:

- An Apple Macintosh computer running MacOS versions 10.10-11.
- XCode Command Line Tools

3.2. WinDriver Installation Process

3.2.1. Windows WinDriver Installation Instructions

** Attention**

Driver installation on Windows requires administrator privileges.

- Run the WinDriver installation - `WD1511.EXE` (for 32-bit) or `WD1511X64.EXE` (for 64-bit) - and follow the installation instructions.
- At the end of the installation, you may be prompted to reboot your computer.

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 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 in the sample Kernel PlugIn projects and makefiles.

If the installation fails with an `ERROR_FILE_NOT_FOUND` error, inspect the Windows registry to see if the `RunOnce` key exists in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the `RunOnce` key is missing, create it; then try installing the INF file again.

The following steps are for registered users only:

To register your copy of WinDriver with the license you received from Jungo, follow these steps:

- Start DriverWizard: **Start | Programs | WinDriver | DriverWizard**.
- Select the **Register WinDriver** option from the **File** menu, and insert the license string you received from Jungo.
- Click the **Activate License** button.
- To register source code that you developed during the evaluation period, refer to the documentation of [WDU_Init\(\)](#) / [WDC_DriverOpen\(\)](#).

When using the low-level `WD_xxx` API instead of the `WDC_xxx` API (which is used by default), refer to the documentation of [WD_License\(\)](#) in this manual.

3.2.2. Linux WinDriver Installation Instructions

3.2.2.1. 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 kernel modules, it must compile with the header files of the Linux kernel during the installation process.

In modern Linux distributions such as Ubuntu, CentOS and Fedora the kernel headers are usually either already installed with the OS, or are easily obtainable via the distribution's package manager. Our suggestion would be to skip to the next section and in case of failure there try the steps mentioned here.

On other less common Linux kernel/distributions, compiling the kernel itself may be necessary. If you're using such a kernel, for more info on compiling it, check out the documentation of the Linux Kernel you're using.

The following instructions may vary according to different versions of the distributions.

CentOS:

```
# Prepare and update apt package manager databases
sudo yum update
# Install prerequisites for kernel development and kernel headers
sudo yum install kernel-devel kernel-headers
sudo yum groupinstall "Development Tools"
# qt5 install for GUI applications to work
sudo yum --enablerepo=extras install epel-release
sudo yum install qt5-qtbase qt5-qtbase-gui
```

Browse the web to find exact installation instructions of CMake for your CentOS version.

Ubuntu/Debian:

```
# Prepare and update apt package manager databases
sudo apt update
# Install prerequisites for kernel development and kernel headers
sudo apt install make gcc bison flex linux-headers-$(uname -r)
# CMake install for compiling user mode samples and generated code
sudo apt install cmake
# qt5 install for GUI applications to work
sudo apt install libqt5gui5
```

- On your development Linux machine, change directory to your preferred installation directory, for example, to your home directory:

```
$ cd ~
```

** Attention**

The path to the installation directory must not contain any spaces.

- Extract the WinDriver distribution file - WD1511LN.tar.gz / WD1511LNx86_64.tar / WD1511LNARM.tar / WD1511LNARM64.tar

```
$ tar -xvzf <file location>/WD1511LN[x86_64/ARM/ARM64].tar.gz
```

3.2.2.2. Installing WinDriver on x86/x86_64 systems

Kindly make sure you've performed the appropriate preparations described in [3.2.2.1. Preparing the System for Installation](#) in order for the following instructions to work.

- Change directory to your WinDriver **redist** directory (the tar automatically creates a WinDriver directory):

```
$ cd <WinDriver directory path>/redist
```
- Install WinDriver:
 - Enter the following:

```
'<WinDriver directory>/redist'$ ./configure
```

** Attention**

The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the `--with-kernel-source=<path>` option, where `<path>` is the full path to the kernel source directory - e.g., `/usr/src/linux`. If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use kbuild to compile the kernel modules. For a full list of the configuration script options, use the `--help` option:

```
./configure --help
```

** Attention**

If you are going to use WinDriver for multiple USB devices with the same VID/ PID, add `--MULTIPLE_SAME_DEVICES` to the `EXTRA_CFLAGS` variable in `makefile.USB.kbuild`.

- Then:

```
<WinDriver directory>/redist$ make
```

- And install the driver (as root user):

```
<WinDriver directory>/redist$ sudo make install
```

- Create a symbolic link so that you can easily launch the DriverWizard GUI:

```
$ ln -s <path to WinDriver>/wizard/wdwizard /usr/bin/wdwizard
```

On older Linux environments such as Ubuntu 16.04 that provide Qt4 instead of Qt5, run `wdwizard_legacy` instead of `wdwizard`.

- Change the read and execute permissions on the file `wdwizard` so that ordinary users can access this program.
- Change the user and group IDs and give read/write permissions to the device file `/dev/windrivr1511`, depending on how you wish to allow users to access hardware through the device. Due to security reasons, by default the device file is created with permissions only for the root user. 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: `windrivr1511:root:root:0666`
- 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 code, and is also used to determine the default directory for saving your generated DriverWizard projects. 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.
- Exit super user mode:

```
# exit
```

- You can now start using WinDriver to access your hardware and generate your driver code!

** Attention**

Use the `WinDriver/util/wdreg` script to load the WinDriver kernel module.

The following steps are for registered users only:

To register your copy of WinDriver with the license you received from Jungo, follow these steps:

- Start DriverWizard:

```
$ <path to WinDriver>/wizard/wdwizard
```

- Select the **Register WinDriver** option from the **File** menu, and insert the license string you received from Jungo.
- Click the **Activate License** button.
- To register source code that you developed during the evaluation period, refer to the documentation of [WDC_DriverOpen\(\)](#). When using the low-level `WD_xxx` API instead of the `WDC_xxx` API (which is used by default), refer to the documentation of [WD_License\(\)](#) in this manual.

3.2.2.3. Installing WinDriver on ARM/ARM64 systems

Kindly make sure you've performed the appropriate preparations described in

[3.2.2.1. Preparing the System for Installation](#) in order for the following instructions to work. The following steps assume you have the ability to compile WinDriver's kernel module on your ARM/ARM64 platform and link it with the running kernel's header. If that is not possible with your platform, see [3.2.2.3.1. Cross compiling the WinDriver kernel module for Linux ARM/ARM64 systems](#)

Starting from WinDriver 14.0.0, Jungo distributes unified ARM and ARM64 versions, in an attempt to support as much platforms as possible. We have tested different platforms to work with WinDriver, but we believe that other platforms with similar credentials should work with these setups.

Therefore we provide an installation script `wd_arm_install.sh` or `wd_arm64_install.sh` that allows the user to either try differently compiled WinDriver kernel modules manually or automatically, until the user hopefully succeeds in installing them.

In order to start:

- Make sure you have all WinDriver dependencies for Linux (gcc, make, kernel sources/headers for your running kernel). Qt applications are currently not supported under ARM, therefore the Qt DLLs are not required.
- Run the following:

```
cd WinDriver/redist && sudo ./wd_arm_install.sh
```

or depending on your platform.

```
cd WinDriver/redist && sudo ./wd_arm64_install.sh
```

- Pick the platform type to install WinDriver for it, or choose automatic install to attempt installation of all available versions of WinDriver on your platform. If installation succeeded, the script will exit. You may modify the script's source code to redistribute the driver if needed.
- If all installations failed – feel free to contact Jungo's support for assistance in getting WinDriver to run on your platform, or for porting WinDriver for it, at sales@jungo.com.

3.2.2.3.1. Cross compiling the WinDriver kernel module for Linux ARM/ARM64 systems Some ARM/ARM64 platforms do not support compilation of kernel modules on the platform itself (which is the classic and recommended way to install WinDriver on Linux). For those cases, we provide a method to cross-compile the WinDriver kernel module on a development machine and then install it on an ARM/ARM64 platform. In order to do this perform the following steps:

On the development machine:

1. Obtain and cross compile your platform's Linux kernel source from the vendor's website. Search for your vendor's instructions on how to do so.
2. Make sure you have installed the relevant gcc compiler for the platform (`arm-linux-gnueabihf-gcc` for ARM, `aarch64-linux-gnu-gcc` for ARM64).
3. Download the ARM/ARM64 WinDriver package to your Linux development machine (even if your development platform is of a different architecture!).
4. Untar the WinDriver package


```
$ tar -xzvf WD1511LNARM.tar.gz
# or
$ tar -xzvf WD1511LNARM64.tar.gz
```
5. Go to the `redist` folder and run the cross compilation script:

for ARM:

```
$ cd WD1511LNARM/redist
$ sudo ./wd_arm_cross_compile.sh PATH_TO_YOUR_COMPILED_KERNEL_SOURCE
```

or for ARM64:

```
$ cd WD1511LNARM64/redist
$ sudo ./wd_arm64_cross_compile.sh PATH_TO_YOUR_COMPILED_KERNEL_SOURCE
```

Follow the instructions in the script and choose one of the options for which type of .ko file to build. Since your platform may be different from the platforms listed in the script (there are endless platforms and kernel types out there), you can try compiling with different options and trying to install the resulting kernel modules on your platform until you find a version that works. If after trying all versions you have not found a version which works with your platform, feel free to contact sales@jungo.com for further assistance.

On the ARM/ARM64 platform:

1. Transfer the WinDriver package to your platform and untar it:

```
$ tar -xzvf WD1511LNARM.tar.gz
# or
$ tar -xzvf WD1511LNARM64.tar.gz
```

2. Go to the redist folder and install the driver:

```
$ cd WD1511LNARM/redist
```

3. Copy the directory WD1511/redist/LINUX.PRECOMPILED.ARM or WD1511/redist/← LINUX.PRECOMPILED.ARM64 from the development machine to WD1511/redist/LINUX.← PRECOMPILED.ARM or WD1511/redist/LINUX.PRECOMPILED.ARM64 on your platform.

4. Install the precompiled kernel module(s):

```
$ ./configure --enable-precompiled
$ sudo make precompiled_install
```

3.2.2.4. Restricting Hardware Access on Linux

Since /dev/windrivr1511 gives direct hardware access to user programs, it may compromise kernel stability on multi-user Linux systems. Please restrict access to DriverWizard and the device file /dev/windrivr1511 to trusted users.

For security reasons the WinDriver installation script does not automatically perform the steps of changing the permissions on /dev/windrivr1460 and the DriverWizard application (wdwizard).

3.2.3. MacOS WinDriver Installation Instructions

3.2.3.1 Preparing the System for Installation

Firstly, disable System Integrity Protection (SIP). The instructions on how to do this may vary between different MacOS versions so find the instructions for your specific version.

On MacOS 11 (Big Sur) in order for WinDriver to be able to load you might also need to disable Apple Mobile File Integrity (AMFI). This could be done by adding "amfi_get_out_of_my_way=1" to your boot-args in the following manner:

```
# Print out the nvram variables to see if boot-args already contain other values
sudo nvram -p
# If boot-args doesn't exist in the list or is empty, run this
sudo nvram boot-args="amfi_get_out_of_my_way=1"
# Otherwise, modify the command such that it keeps all previous values and appends
amfi_get_out_of_my_way=1 as well.
```

**** Attention****

Disabling SIP and AMFI weakens your system's security mechanisms but is mandatory for WinDriver to run on MacOS and to allow Driver Development in general. Disable these mechanisms at your own risk.

Afterwards, install the dependencies, these steps assume you have the Homebrew package manager installed. If it is not yet installed, kindly install it first, and then run the following commands:

Install Qt5 (required for DriverWizard, Debug Monitor)

```
$ brew install qt5
```

Then install CMake (required for compiling generated code and samples)

```
$ brew install cmake
```

3.2.3.2 Installation on MacOS x86_64

Then using a Terminal window, run the following commands:

```
# Unzipping  
$ tar -xvzf WD1511MAC.tar.gz  
# Driver install  
$ cd WD1511MAC/redist/  
$ sudo ./wd_mac_install.sh
```

- On MacOS version 11 (Big Sur) and higher, installing WinDriver using the install script will raise a pop-up message from the Security & Privacy section of the System Preferences to allow installing a kernel extension. In order for WinDriver to run, you must allow it to be installed. This will also require a restart. After restarting the system, you might need to run
\$ sudo ./wd_mac_install.sh
again.

3.2.3.3 Installation on MacOS ARM64 (M1)

Then using a Terminal window, run the following commands:

```
# Unzipping  
$ tar -xvzf WD1511MACARM64.tar.gz  
# Driver install  
$ cd WD1511MACARM64/redist/  
$ sudo ./wd_mac_install.sh
```

- The ARM64 version of WinDriver for MacOS requires that before installation the users will specify the Vendor ID(s) and Product ID(s) of the devices they wish to use. The install script will prompt the users to enter these parameters. For more information, search for Apple's documentation regarding `IOPCIMatch`.
- On MacOS version 11 (Big Sur) and higher, installing WinDriver using the install script will raise a pop-up message from the Security & Privacy section of the System Preferences to allow installing a kernel extension. In order for WinDriver to run, you must allow it to be installed. This will also require a restart. After restarting the system, you might need to run
\$ sudo ./wd_mac_install.sh
again.

** Attention**

If the users enter a VID/PID mask which is too "wide" (covers many devices) this might cause a system crash as MacOS Big Sur and higher does not allow WinDriver to access certain system-reserved devices. The system will reboot and remove the kernel extension and the user will have to run the installation script again and enter different, more specific parameters in order to install WinDriver. Performing a PCI scan on such systems (using the DriverWizard or [WDC_PciScanDevices\(\)](#)) will show only the devices that answer to the parameters provided by the user in this stage.

3.3. Upgrading Your Installation

To upgrade to a new version of WinDriver, follow the installation steps for your target operating system, as outlined in the previous [Section 3.2](#). Download and install a new version of WinDriver that matches your development platform and the operating systems and CPU configurations of the target platforms on which you intend the driver to be used.

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

To upgrade your source code:

- Pass the new license string as a parameter to [WDC_DriverOpen\(\)](#) / [WDU_Init\(\)](#) (or to [WD_License\(\)](#) when using the low-level API).
- Verify that the call to [WD_DriverName\(\)](#) in your driver code (if exists) uses the name of the new driver module - `windrivr1511` or your renamed version of this driver.

If you use the generated DriverWizard code or one of the samples from the new WinDriver version, the code will already use the default driver name from the new version. Also, if your code is based on generated/sample code from an earlier version of WinDriver, rebuilding the code with `windrivr.h` from the new version is sufficient to update the code to use the new default driver-module name (due to the use of the `WD_DEFAULT_DRIVER_NAME_BASE` definition).

If you elect to rename the WinDriver driver module (as outlined in [Chapter 17.2](#)), ensure that your code calls `WD_DriverName()` with your custom driver name. If you rename the driver from the new version to a name already used in your old project, you do not need to modify your code.

3.4. Checking Your Installation

3.4.1. Installation Check

- Start DriverWizard - <path to WinDriver>/wizard/wdwizard. On Windows you can also run DriverWizard from the **Start** menu: **Start | Programs | WinDriver | DriverWizard**.
- If you are a registered user, make sure that your WinDriver license is registered (refer to [Section 3.2] ([3.2. WinDriver Installation Process](#)), which explains how to install WinDriver and register your license).

If you are an evaluation version user, you do not need to register a license.

For PCI cards/USB devices - Insert your card into the PCI bus/your device into the USB slot, and verify that Driver Wizard detects it. For ISA cards (Windows and Linux) - Insert your card into the ISA bus, configure DriverWizard with your card's resources and try to read/write to the card using DriverWizard.

3.5. Uninstalling WinDriver

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

3.5.1. Windows WinDriver Uninstall Instructions

You can select to use the graphical `wdreg_gui.exe` utility instead of `wdreg.exe`.

`wdreg.exe` and `wdreg_gui.exe` are found in the `WinDriver\util` directory (see [Dynamically Loading Your Driver](#) for details regarding these utilities).

To uninstall WinDriver, please follow these steps:

- Close any open WinDriver applications, including DriverWizard, the Debug Monitor, and user-specific applications.
- If you created a Kernel PlugIn driver, uninstall and erase it:
 - If your Kernel PlugIn driver is currently installed, uninstall it using the `wdreg` utility:

```
# The Kernel PlugIn driver name should be specified without the '*.sys' extension.  
wdreg -name <Kernel PlugIn name> uninstall
```

- Erase your Kernel PlugIn driver from the `windir%\system32\drivers` directory.
- Uninstall all Plug-and-Play devices (USB/PCI) that have been registered with WinDriver via an INF file:

```
# Uninstall a device INF using the wdreg utility:  
wdreg -inf <path to the INF file> uninstall
```

- Verify that no INF files that register your device(s) with WinDriver's kernel module (`windrivr1511.sys`) are found in the `windir%\inf` directory.
- Uninstall WinDriver:

- **On the development PC**, on which you installed the WinDriver toolkit: **Start | WinDriver | Uninstall**, OR run the `uninstall.exe` utility from the WinDriver installation directory. The uninstall will stop and unload the WinDriver kernel module (`windrvr1511.sys`); delete the copy of the `windrvr1511.inf` file from the `windir%\inf` directory; delete WinDriver from Windows' **Start** menu; delete the WinDriver installation directory (except for files that you added to this directory); and delete the shortcut icons to the DriverWizard and Debug Monitor utilities from the Desktop.
- **On a target PC**, on which you installed the WinDriver kernel module (`windrvr1511.sys`), but not the entire WinDriver toolkit. Use the `wdreg` utility to stop and unload the driver:
`wdreg -inf <path to windrvr1511.inf> uninstall`

** Attention**

When running this command, `windrvr1511.sys` should reside in the same directory as `windrvr1511.inf`.

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

If you attempt to uninstall WinDriver while there are open handles to the WinDriver service (`windrvr1511.sys` or your renamed driver, or there are connected and enabled Plug-and-Play devices that are registered to work with this service, `wdreg` will fail to uninstall the driver. This ensures that you do not uninstall the driver while it is being used.

You can check if the WinDriver kernel module is loaded by running the Debug Monitor utility (`Win↔Driver\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 the Debug Monitor executables; to use this utility after the uninstallation, create a copy of `wddebug_gui.exe` before performing the `uninstall` procedure.

- If `windrvr1511.sys` was successfully unloaded, erase the following files (if they exist):
 - `windir%\system32\drivers\windrvr1511.sys`
 - `windir%\inf\windrvr1511.inf`
 - `windir%\system32\wdapi1511.dll`
 - `windir%\sysWOW64\wdapi1511.dll` (Windows x64)
- Reboot the computer.

3.5.2. Linux WinDriver Uninstall Instructions

The following commands must be executed with root privileges.

- Verify that the WinDriver driver module is not being used by another program:
 - View the list of modules and the programs using each of them:
`> /sbin/lsmod "`
- Identify any applications and modules that are using the WinDriver driver module. By default, WinDriver module names begin with `windrvr1511`.
- Close any applications that are using the WinDriver driver module.
- If you created a Kernel Plugin driver, unload the Kernel Plugin driver module:

`/sbin/rmmod kp_xxx_module`

- Run the following command to unload the WinDriver driver module:

`/sbin/modprobe -r windrvr1511`

- If you created a Kernel Plugin driver, remove it as well.

- Remove the file `.windriver.rc` from the `/etc` directory:

```
rm -f /etc/.windriver.rc
```

- Remove the file `.windriver.rc` from `$HOME`:

```
rm -f $HOME/.windriver.rc
```

- If you created a symbolic link to `DriverWizard`, remove the link using the command:

```
rm -f /usr/bin/wdwizard
```

- Remove the WinDriver installation directory using the command:

```
rm -rf (path to the WinDriver directory)
```

For example, # `rm -rf ~/WinDriver`.

- Remove the WinDriver shared object file, if it exists:

`/usr/lib/libwdapi1511.so` (32-bit x86) / `/usr/lib64/libwdapi1511.so` (64-bit x86).

3.5.3. MacOS WinDriver Uninstall Instructions

To uninstall WinDriver on MacOS, open a terminal window and type the following commands:

```
$ cd WinDriver-1511-Darwin/redist/  
$ sudo ./wd_mac_uninstall.sh
```

Chapter 4

PCI Express Overview

4.1. 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 consists 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.

4.2. 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.

In addition, WinDriver provides you with a set of APIs for easy access to the PCI Express extended configuration space on target platforms that support such access (e.g., Windows and Linux) - see the description of the [WDC_PciReadCfg8\(\)](#) / [WDC_PciReadCfg16\(\)](#) / [WDC_PciReadCfg32\(\)](#) / [WDC_PciReadCfg64\(\)](#) and [WDC_PciWriteCfg8\(\)](#) / [WDC_PciWriteCfg16\(\)](#) / [WDC_PciWriteCfg32\(\)](#) / [WDC_PciWriteCfg64\(\)](#) functions in this manual, or the description of the lower-level [WD_PciConfigDump\(\)](#) function.

WinDriver interrupt handling APIs also support Message-Signaled Interrupts (MSI) and Extended Message-Signaled Interrupts (MSI-X).

4.3. The pci_dump and pci_scan Utilities

`pci_dump` is a sample utility console-mode program that is provided with the WinDriver tool-kit and enables you to dump the contents of the PCI configuration registers into a readable format. It can be found in the `WinDriver/util` directory.

The source code of this utility is provided as well and can be found at: `WinDriver/samples/c/pci_dump/pci_dump.c`.

`pci_scan` is a sample utility console-mode program that is provided with the WinDriver tool-kit and enables you to scan the PCI device that are connected to your computer in a readable format. It can be found in the `WinDriver/util` directory.

The source code of this utility is provided as well and can be found at: `WinDriver/samples/c/pci_scan/pci_scan.c`.

4.3.1. Dump PCI Configuration Space into a file

To log the `pci_dump` output into a file (`dump.txt`), do the following:

- Open a command prompt window.
- Run `pci_dump` as `pci_dump > dump.txt`.
- Hit **Enter** continuously, until the program's execution is completed.

At the end of this process you will have a `dump.txt` file in the `WinDriver/util` directory (from which `pci_dump` was run).

The same process can similarly be done with `pci_scan` output as well.

** Recommendation**

If you have any technical question, we firstly recommend searching the manual. If you cannot find an answer to your question, please send us an e-mail to windriver@jungo.com, or if you are already in contact with someone from our team, just send an e-mail to that person directly. Please note that in order to ensure a quick and effective support our engineers may request clear detailed description of the steps you performed, specify which step failed and what was the exact nature of the failure or erroneous behavior that you encountered (including complete error messages, screenshots, logs).

4.4. FAQ

4.4.1. Enabling legacy PCI configuration space read/write for identifying PCI devices on Windows

In version 6.2.0 of WinDriver, the PCI configuration space read/write method on Windows was upgraded to a more advanced method. On rare occasions, this method fails to identify some PCI devices. To resolve this problem,

from WinDriver version 10.3.1 you can revert to the legacy PCI configuration space read/write method by doing the following:

Set the `PciCfgRwCompat` registry key flag in the WinDriver driver INF file (`windrvr<version>.inf`(e.g. `windrvrl511.inf`) / `windrvr6.inf` in earlier versions)) - to 1:
HKEY, Parameters, `PciCfgRwCompat`, 0x00010001, 1

Beginning with version 11.1.0 of WinDriver, the driver INF file contains a similar line, which sets the `PciCfgRwCompat` flag to 0 (default), so you only need to modify the value of the flag in the existing line to 1:

Open a command-line prompt and reinstall the driver by running the following commands from the `WinDriver\util\wdregdirectory` (where "WinDriver" is the path to your WinDriver installation directory):

```
$ wdreg -inf <path to your device INF file> uninstall  
$ wdreg -inf <path to the driver INF file> uninstall  
$ wdreg -inf <path to the driver INF file> install  
$ wdreg -inf <path to your device INF file> install
```

4.3.2. How do I access the memory on my PCI card using WinDriver?

First, locate the slot to which your card is connected, using [WDC_PciScanDevices\(\)](#).

Then get the card's information by calling [WDC_PciGetDeviceInfo\(\)](#).

This information includes the memory range chosen for the card by the Plug and Play system.

Now call [WDC_PciDeviceOpen\(\)](#) to install the memory range and map it into both kernel and user mode virtual address spaces.

You can then either access the memory directly from your user mode application (more efficient), by using the user mode mapping of the physical address - returned by [WDC_PciDeviceOpen\(\)](#) in ((`PWDC_DEVICE`) hDev) ->`cardReg.Card.Item[i].I.Mem.pUserDirectAddr` (where `i` is the index number of the memory range in the Item array) - or pass the kernel mode mapping of the memory - ((`PWDC_DEVICE`) hDev) ->`cardReg.Card.Item[i].I.Mem.pTransAddr` - to [WD_Transfer\(\)](#) / [WD_MultiTransfer\(\)](#), in order to access the memory in the kernel. Check out the documentation of [WDC_DEVICE](#) for more info.

This is demonstrated, for example, in the sample code found in the `WinDriver/samples/c/pci_diag` directory, and in the diagnostics DriverWizard code that you can generate for your PCI card.

** Attention**

To access memory directly in the kernel, from within a Kernel Plugin project, you must use the kernel mode mapping of the physical memory address - returned by [WDC_PciDeviceOpen\(\)](#) in ((`PWDC_DEVICE`) hDev) ->`cardReg.Card.Item[i].I.Mem.pTransAddr` - and not the user mode mapping that is used to access the memory directly from your user-mode application.

4.3.3. How do I use WinDriver for PCI Express over Thunderbolt?

Assuming your computer's Thunderbolt adapter is correctly configured and your device is recognized by the operating system, then a PCI Express device connected over Thunderbolt should be recognized by WinDriver as a regular PCI Express device and this should not require special actions on the WinDriver side. Some motherboard chipsets require changes in their BIOS settings to enable Thunderbolt support, consult your vendor's documentation for more info.

Chapter 5

USB Overview

This chapter explores the basic characteristics of the Universal Serial Bus (USB) and introduces WinDriver USB's features and architecture.

** Note**

The references to the WinDriver USB toolkit in this chapter relate to the standard WinDriver USB toolkit for development of USB host drivers.

5.1. Introduction to USB

USB (Universal Serial Bus) is an industry standard extension to the PC architecture for attaching peripherals to the computer. It was originally developed in 1995 by leading PC and telecommunication industry companies, such as Intel, Compaq, Microsoft and NEC. USB was developed to meet several needs, among them the needs for an inexpensive and widespread connectivity solution for peripherals in general and for computer telephony integration in particular, an easy-to-use and flexible method of reconfiguring the PC, and a solution for adding a large number of external peripherals. The USB standard meets these needs.

The USB specification allows for the connection of a maximum of 127 peripheral devices (including hubs) to the system, either on the same port or on different ports.

USB also supports Plug-and-Play installation and hot swapping.

The **USB 1.1** standard supports both isochronous and asynchronous data transfers and has dual speed data transfer: 1.5 Mb/s (megabits per second) for **low-speed** USB devices and 12 Mb/s for **full-speed** USB devices (much faster than the original serial port). Cables connecting the device to the PC can be up to five meters (16.4 feet) long. USB includes built-in power distribution for low power devices and can provide limited power (up to 500 mA of current) to devices attached on the bus. The **USB 2.0** standard supports a signalling rate of 480 Mb/s, known as **'high-speed'**, which is 40 times faster than the USB 1.1 full-speed transfer rate.

USB 2.0 is fully forward- and backward-compatible with USB 1.1 and uses existing cables and connectors. USB 2.0 supports connections with PC peripherals that provide expanded functionality and require wider bandwidth. In addition, it can handle a larger number of peripherals simultaneously. USB 2.0 enhances the user's experience of many applications, including interactive gaming, broadband Internet access, desktop and Web publishing, Internet services and conferencing.

The **USB 3.2 Gen 1** standard supports a signalling rate of up to 5 Gb/s in a new mode known as **SuperSpeed USB**. The **USB 3.2 Gen 2** standard supports a signalling rate of up to 10 Gb/s in a new mode known as **SuperSpeed USB 10Gbps**. The **USB 3.2 Gen 2x2** standard supports a signalling rate of up to 20 Gb/s in newer modes also named **SuperSpeed USB 20Gbps** whilst preserving backwards compatibility with previous USB versions.

New name	Old name	Original name	SuperSpeed name	Max name
USB 3.2 Gen 2x2	N/A	USB 3.2	SuperSpeed USB 20Gbps	20Gbps
USB 3.2 Gen 2	USB 3.1 Gen 2	USB 3.1	SuperSpeed USB 10Gbps	10Gbps
USB 3.2 Gen 1	USB 3.1 Gen 1	USB 3.0	SuperSpeed USB	5Gbps

Because of its benefits USB is currently enjoying broad market acceptance.

5.2. WinDriver USB Benefits

This section describes the main benefits of the USB standard and the WinDriver USB toolkit, which supports this standard:

- External connection, maximizing ease of use
- Self identifying peripherals supporting automatic mapping of function to driver and configuration
- Dynamically attachable and re-configurable peripherals
- Suitable for device bandwidths ranging from a few Kb/s to hundreds of Mb/s
- Supports isochronous as well as asynchronous transfer types over the same set of wires
- Supports simultaneous operation of many devices (multiple connections)
- Supports a data transfer rate of up to 10 Gbp/s (super-speed+) for USB 3.1, up to 5 Gbp/s(super-speed) for USB 3.0, up to 480 Mb/s (high-speed) for USB 2.0 and up to 12 Mb/s (full-speed) for USB 1.1. The supported speed also depends on the operating system and hardware utilised
- Guaranteed bandwidth and low latencies; appropriate for telephony, audio, etc. (isochronous transfer may use almost the entire bus bandwidth)
- Flexibility: supports a wide range of packet sizes and a wide range of data transfer rates
- Robustness: built-in error handling mechanism and dynamic insertion and removal of devices with no delay observed by the user
- Synergy with PC industry; Uses commodity technologies
- Optimized for integration in peripheral and host hardware
- Low-cost implementation, therefore suitable for development of low-cost peripherals
- Low-cost cables and connectors
- Built-in power management and distribution
- Specific library support for custom USB HID devices

5.3. USB Components

The Universal Serial Bus (USB) consists of the following primary components:

USB Host

The USB host platform is where the USB host controller is installed and where the client software/device driver runs. The *USB Host Controller* is the interface between the host and the USB peripherals. The host is responsible for detecting the insertion and removal of USB devices, managing the control and data flow between the host and the devices, providing power to attached devices and more.

USB Hub

A USB device that allows multiple USB devices to attach to a single USB port on a USB host. Hubs on the back plane of the hosts are called *root hubs*. Other hubs are called external hubs.

USB Function

A USB device that can transmit or receive data or control information over the bus and that provides a function. A function is typically implemented as a separate peripheral device that plugs into a port on a hub using a cable. However, it is also possible to create a *compound device*, which is a physical package that implements multiple functions and an embedded hub with a single USB cable. A compound device appears to the host as a hub with one or more non-removable USB devices, which may have ports to support the connection of external devices.

5.4. Data Flow in USB Devices

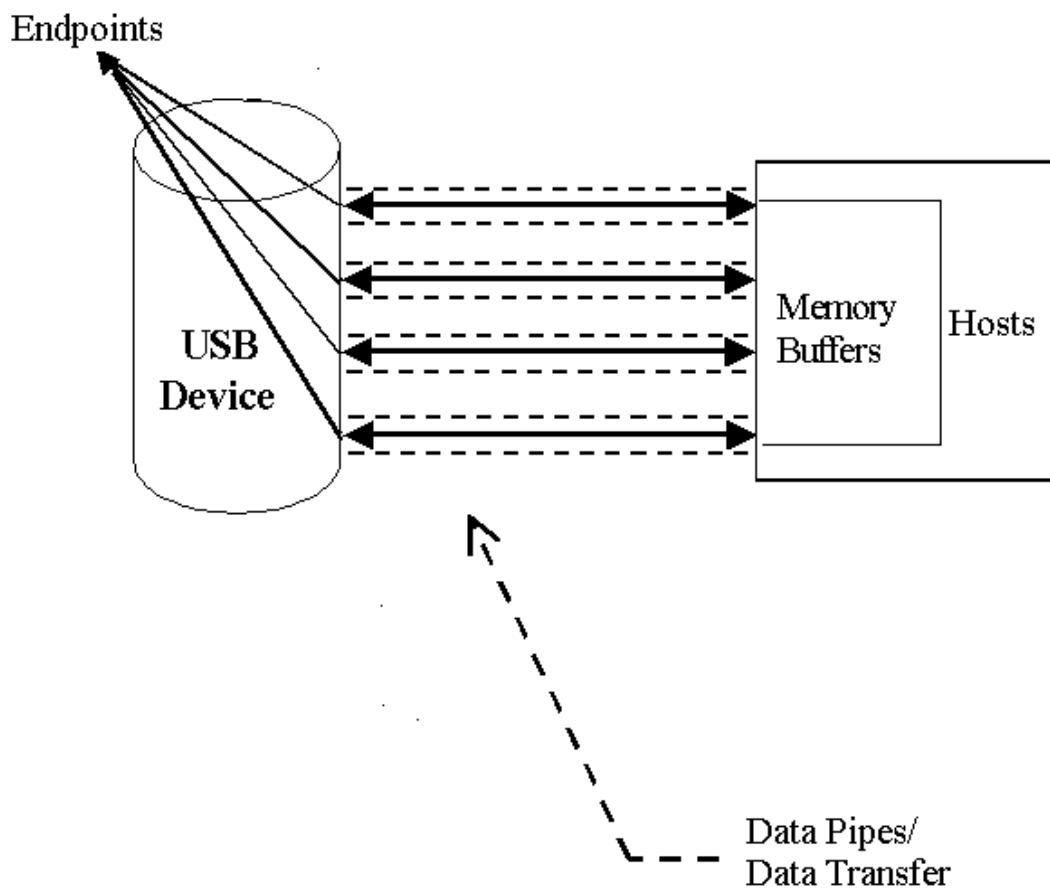
During the operation of a USB device, the host can initiate a flow of data between the client software and the device. Data can be transferred between the host and only one device at a time (*peer to peer communication*). However, two hosts cannot communicate directly, nor can two USB devices (with the exception of On-The-Go (OTG) devices, where one device acts as the master (host) and the other as the slave.)

The data on the USB bus is transferred via pipes that run between software memory buffers on the host and endpoints on the device.

Data flow on the USB bus is half-duplex, i.e., data can be transmitted only in one direction at a given time.

An **endpoint** is a uniquely identifiable entity on a USB device, which is the source or terminus of the data that flows from or to the device. Each USB device, logical or physical, has a collection of independent endpoints. The three USB speeds (low, full and high) all support one bi-directional control endpoint (endpoint zero) and 15 unidirectional endpoints. Each unidirectional endpoint can be used for either inbound or outbound transfers, so theoretically there are 30 supported endpoints. Each endpoint has the following attributes: bus access frequency, bandwidth requirement, endpoint number, error handling mechanism, maximum packet size that can be transmitted or received, transfer type and direction (into or out of the device).

USB Endpoints



A **pipe** is a logical component that represents an association between an endpoint on the USB device and software on the host. Data is moved to and from a device through a pipe. A pipe can be either a stream pipe or a message pipe, depending on the type of data transfer used in the pipe. *Stream pipes* handle interrupt, bulk and isochronous transfers, while *message pipes* support the control transfer type.

5.5. USB Data Exchange

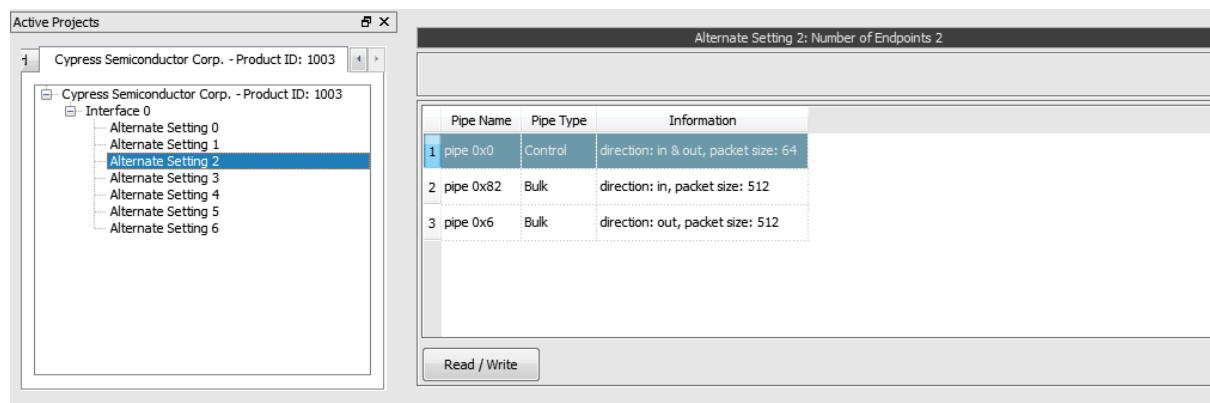
The USB standard supports two kinds of data exchange between a host and a device: functional data exchange and control exchange.

Functional Data Exchange is used to move data to and from the device. There are three types of USB data transfers: Bulk, Interrupt and Isochronous.

Control Exchange is used to determine device identification and configuration requirements, and to configure a device; it can also be used for other device-specific purposes, including control of other pipes on the device. Control exchange takes place via a control pipe - the default *pipe 0*, which always exists. The control transfer consists of a *setup stage* (in which a setup packet is sent from the host to the device), an optional *data stage* and a *status stage*.

The figure below depicts a USB device with one bi-directional control pipe (endpoint) and two functional data transfer pipes (endpoints), as identified by WinDriver's DriverWizard utility.

USB Pipes



5.6. USB Data Transfer Types

The USB device (function) communicates with the host by transferring data through a pipe between a memory buffer on the host and an endpoint on the device. USB supports four different transfer types. A type is selected for a specific endpoint according to the requirements of the device and the software. The transfer type of a specific endpoint is determined in the endpoint descriptor.

The USB specification provides for the following data transfer types:

5.6.1. Control Transfer

Control Transfer is mainly intended to support configuration, command and status operations between the software on the host and the device.

This transfer type is used for low-, full- and high-speed devices.

Each USB device has at least one control pipe (default pipe), which provides access to the configuration, status and control information.

Control transfer is bursty, non-periodic communication.

The control pipe is bi-directional - i.e., data can flow in both directions.

Control transfer has a robust error detection, recovery and retransmission mechanism and retries are made without the involvement of the driver.

The maximum packet size for control endpoints can be only 8 bytes for low-speed devices; 8, 16, 32, or 64 bytes for full-speed devices; and only 64 bytes for high-speed devices.

For more in-depth information regarding USB control transfers and their implementation, refer to [USB Advanced Features](#) of the manual.

5.6.2. Isochronous Transfer

Isochronous Transfer is most commonly used for time-dependent information, such as multimedia streams and telephony.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Isochronous transfer is periodic and continuous.

The isochronous pipe is unidirectional, i.e., a certain endpoint can either transmit or receive information. Bi-directional isochronous communication requires two isochronous pipes, one in each direction.

USB guarantees the isochronous transfer access to the USB bandwidth (i.e., it reserves the required amount of bytes of the USB frame) with bounded latency, and guarantees the data transfer rate through the pipe, unless there is less data transmitted.

Since timeliness is more important than correctness in this type of transfer, no retries are made in case of error in the data transfer. However, the data receiver can determine that an error occurred on the bus.

5.6.3. Interrupt Transfer

Interrupt Transfer is intended for devices that send and receive small amounts of data infrequently or in an asynchronous time frame.

This transfer type can be used for low-, full- and high-speed devices.

Interrupt transfer type guarantees a maximum service period and that delivery will be re-attempted in the next period if there is an error on the bus.

The interrupt pipe, like the isochronous pipe, is unidirectional and periodical.

The maximum packet size for interrupt endpoints can be 8 bytes or less for low-speed devices; 64bytes or less for full-speed devices; and 1,024 bytes or less for high-speed devices.

5.6.4. Bulk Transfer

Bulk Transfer is typically used for devices that transfer large amounts of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners.

This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.

Bulk transfer is non-periodic, large packet, bursty communication.

Bulk transfer allows access to the bus on an "as-available" basis, guarantees the data transfer but not the latency, and provides an error check mechanism with retries attempts. If part of the USB bandwidth is not being used for other transfers, the system will use it for bulk transfer.

Like the other stream pipes (isochronous and interrupt), the bulk pipe is also unidirectional, so bi-directional transfers require two endpoints.

The maximum packet size for bulk endpoints can be 8, 16, 32, or 64 bytes for full-speed devices, and 512 bytes for high-speed devices.

5.7. USB Configuration

Before the USB function (or functions, in a compound device) can be operated, the device must be configured. The host does the configuring by acquiring the configuration information from the USB device. USB devices report their attributes by descriptors. A **descriptor** is the defined structure and format in which the data is transferred. A complete description of the USB descriptors can be found in [14.2. USB Control Transfers Overview](#).

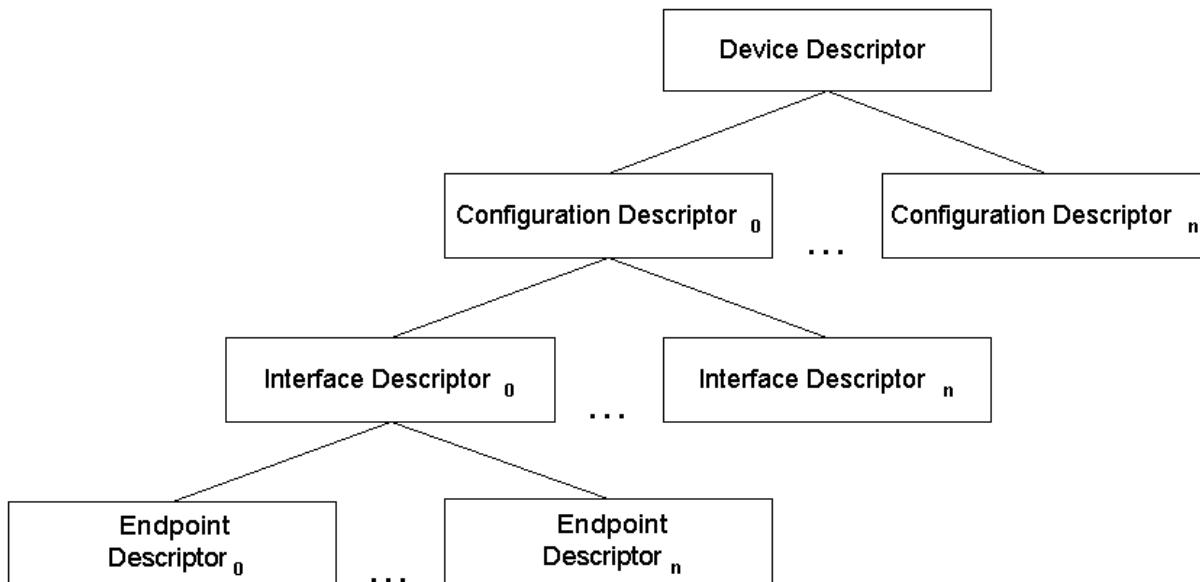
It is best to view the USB descriptors as a hierarchical structure with four levels:

- The Device level
- The Configuration level

- The Interface level (this level may include an optional sub-level called Alternate Setting)
- The Endpoint level

There is only one device descriptor for each USB device. Each device has one or more configurations, each configuration has one or more interfaces, and each interface has zero or more endpoints.

Device Descriptors



Device Level

The device descriptor includes general information about the USB device, i.e. global information for all of the device configurations. The device descriptor identifies, among other things, the device class (HID device, hub, locator device, etc.), subclass, protocol code, vendor ID, device ID and more. Each USB device has one device descriptor.

Configuration Level

A USB device has one or more configuration descriptors. Each descriptor identifies the number of interfaces grouped in the configuration and the power attributes of the configuration (such as self-powered, remote wakeup, maximum power consumption and more). Only one configuration can be loaded at a given time. For example, an ISDN adapter might have two different configurations, one that presents it with a single interface of 128 Kb/s and a second that presents it with two interfaces of 64 Kb/s each.

Interface Level

The interface is a related set of endpoints that present a specific functionality or feature of the device. Each interface may operate independently. The interface descriptor describes the number of the interface, the number of endpoints used by this interface and the interface-specific class, subclass and protocol values when the interface operates independently.

In addition, an interface may have **alternate settings**. The alternate settings allow the endpoints or their characteristics to be varied after the device is configured.

Endpoint Level

The lowest level is the endpoint descriptor, which provides the host with information regarding the endpoint's data transfer type and maximum packet size. For isochronous endpoints, the maximum packet size is used to reserve the required bus time for the data transfer - i.e., the bandwidth. Other endpoint attributes are its bus access frequency, endpoint number, error handling mechanism and direction. The same endpoint can have different properties (and consequently different uses) in different alternate settings.

Seems complicated? Not at all! WinDriver automates the USB configuration process. The included Driver Wizard utility and USB diagnostics application scan the USB bus, detect all USB devices and their configurations, interfaces, alternate settings and endpoints, and enable you to pick the desired configuration before starting driver development.

WinDriver identifies the endpoint transfer type as determined in the endpoint descriptor. The driver created with WinDriver contains all configuration information acquired at this early stage.

5.8. WinDriver USB

WinDriver USB enables developers to quickly develop high-performance drivers for USB-based devices without having to learn the USB specifications and operating system internals, or use the operating system development kits. For example, Windows drivers can be developed without using the Windows Driver Kit (WDK) or learning the Windows Driver Model (WDM).

The driver code developed with WinDriver USB is binary compatible across the supported Windows platforms and source code compatible across all supported operating systems. For an up-to-date list of supported operating systems, see [1.4. Supported Platforms](#).

WinDriver USB is a generic tool kit that supports all USB devices from all vendors and with all types of configurations.

WinDriver USB encapsulates the USB specification and architecture, letting you focus on your application logic. WinDriver USB features the graphical DriverWizard utility, which enables you to easily detect your hardware, view its configuration information, and test it, before writing a single line of code: DriverWizard first lets you choose the desired configuration, interface and alternate setting combination, using a friendly graphical user interface. After detecting and configuring your USB device, you can proceed to test the communication with the device - perform data transfers on the pipes, send control requests, reset the pipes, etc. - in order to ensure that all your hardware resources function as expected.

After your hardware is diagnosed, you can use DriverWizard to automatically generate your device driver source code. WinDriver USB provides user-mode APIs, which you can call from within your application in order to implement the communication with your device. The WinDriver USB API includes USB-unique operations such as reset of a pipe or a device. The generated DriverWizard code implements a diagnostics application, which demonstrates how to use WinDriver's USB API to drive your specific device. In order to use the application you just need to compile and run it. You can jump-start your development cycle by using this application as your skeletal driver and then modifying the code as needed to implement the desired driver functionality for your specific device.

DriverWizard also automates the creation of an INF file that registers your device to work with WinDriver, which is an essential step in order to correctly identify and handle USB devices using WinDriver.

With WinDriver USB, all development is done in the user mode, using familiar development and debugging tools and your favorite compiler or development environment (such as MS Visual Studio, CMake, GCC, Windows GCC).

5.9. WinDriver USB Architecture

To access your hardware, your application calls the WinDriver kernel module using functions from the WinDriver USB API. The high-level functions utilize the low-level functions, which use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application. The WinDriver kernel module accesses your USB device resources through the native operating system calls.

There are two layers responsible for abstracting the USB device to the USB device driver. The upper layer is the **USB Driver (USBD)** layer, which includes the USB Hub Driver and the USB Core Driver. The lower level is the **Host Controller Driver (HCD)** layer. The division of duties between the HCD and USBD layers is not defined and is operating system dependent. Both the HCD and USBD are software interfaces and components of the operating system, where the HCD layer represents a lower level of abstraction.

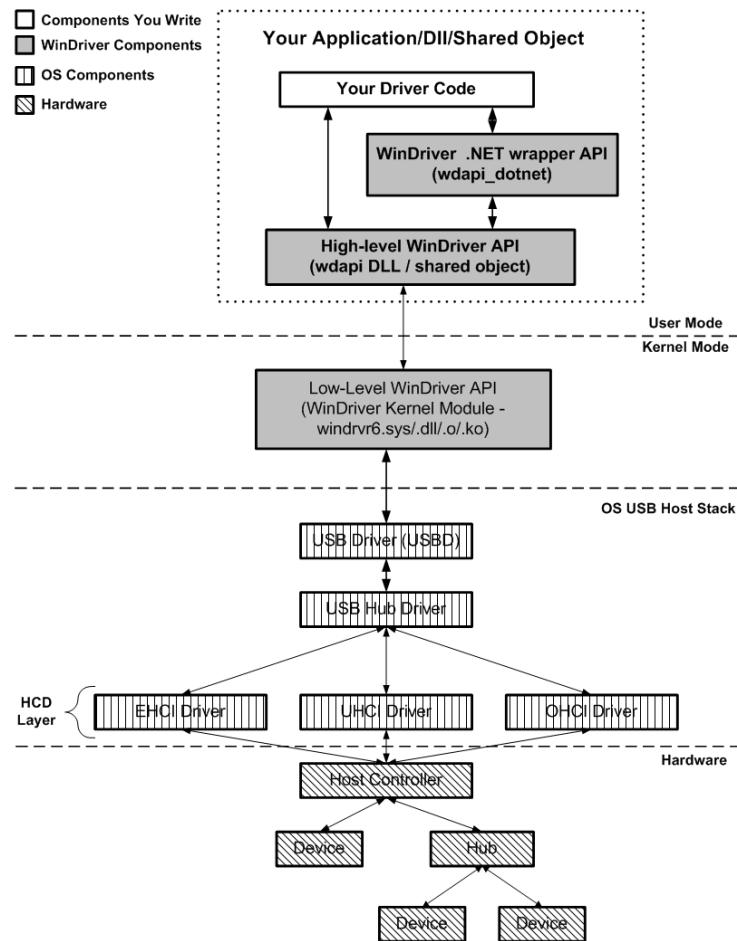
The **HCD** is the software layer that provides an abstraction of the host controller hardware, while the **USBD** provides an abstraction of the USB device and the data transfer between the host software and the function of the USB device.

The **USBD** communicates with its clients (the specific device driver, for example) through the USB Driver Interface (**USBDI**). At the lower level, the Core Driver and USB Hub Driver implement the hardware access and data transfer by communicating with the HCD using the Host Controller Driver Interface (**HCDI**).

The USB Hub Driver is responsible for identifying the addition and removal of devices from a particular hub. When the Hub Driver receives a signal that a device was attached or detached, it uses additional host software and the USB Core Driver to recognize and configure the device. The software implementing the configuration can include the hub driver, the device driver, and other software.

WinDriver USB abstracts the configuration procedure and hardware access described above for the developer. With WinDriver's USB API, developers can perform all the hardware-related operations without having to master the lower-level implementation for supporting these operations.

WinDriver USB Architecture



5.10. WinDriver USB (WDU) Library Overview

This section provides a general overview of WinDriver's USB Library (WDU), including:

- An outline of the WDU_xxx API calling sequence.
- Instructions for upgrading code developed with the previous WinDriver USB API, used in version 5.22 and earlier, to use the improved WDU_xxx API. If you do not need to upgrade USB driver code developed with an older version of WinDriver, simply skip this section.

The WDU library's interface is found in the `WinDriver/include/wdu_lib.h` and `WinDriver/include/windrvr.h` header files, which should be included from any source file that calls the WDU API (`wdu_lib.h` already includes `windrvr.h`).

5.10.1. Calling Sequence for WinDriver USB

The WinDriver WDU_xxx USB API is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

You can implement the three user callback functions specified in the next section: `WDU_ATTACH_CALLBACK`, `WDU_DETACH_CALLBACK` and `WDU_POWER_CHANGE_CALLBACK` (at the very least `WDU_ATTACH_CALLBACK`). These functions are used to notify your application when a relevant system event occurs, such as the attaching or detaching of a USB device. For best performance, minimal processing should be done in these functions.

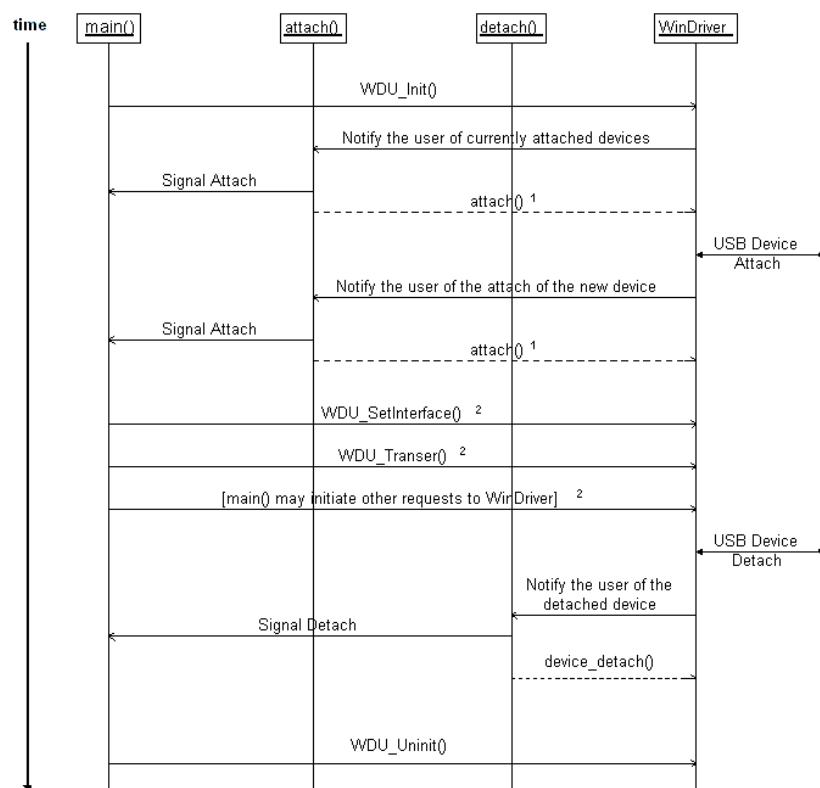
Your application calls `WDU_Init()` and provides the criteria according to which the system identifies a device as relevant or irrelevant. The `WDU_Init()` function must also pass pointers to the user callback functions.

Your application then simply waits to receive a notification of an event. Upon receipt of such a notification, processing continues. Your application may make use of any functions defined in the high- or low-level APIs below. The high-level functions, provided for your convenience, make use of the low-level functions, which in turn use IOCTLs to enable communication between the WinDriver kernel module and your user-mode application.

When exiting, your application calls `WDU_Uninit()` to stop listening to devices matching the given criteria and to unregister the notification callbacks for these devices.

The following figure depicts the calling sequence described above. Each vertical line represents a function or process. Each horizontal arrow represents a signal or request, drawn from the initiator to the recipient. Time progresses from top to bottom.

WinDriver USB Calling Sequence



¹ If the `WD_ACKNOWLEDGE` flag was set in the call to `WDU_Init()`, the `attach()` callback should return TRUE to accept control of the device or FALSE otherwise.

² Only possible if the `attach()` callback returned TRUE.

The following piece of meta-code can serve as a framework for your user-mode application's code:

```

attach()
{
    ...
    if this is my device
    /*
     * Set the desired alternate setting ;
     * Signal main() about the attachment of this device
     */
    return TRUE;
    else
        return FALSE;
}
  
```

```

}
detach()
{
    ...
    signal main() about the detachment of this device
    ...
}
main()
{
    WDU_Init(...);
    ...
    while (...){
        /* wait for new devices */
        ...

        /* issue transfers */
        ...
    }
    ...
    WDU_Uninit();
}

```

5.10.2. Upgrading from the WD_xxx USB API to the WDU_xxx API

The WinDriver WDU_xxx USB API, provided beginning with version 6.0.0, is designed to support event-driven transfers between your user-mode USB application and USB devices. This is in contrast to earlier versions, in which USB devices were initialized and controlled using a specific sequence of function calls.

As a result of this change, you will need to modify your USB applications that were designed to interface with earlier versions of WinDriver to ensure that they will work with WinDriver versions 6.X on all supported platforms and not only on Microsoft Windows.

You will have to reorganize your application's code so that it conforms with the framework illustrated by the piece of meta-code provided in [5.10.1. Calling Sequence for WinDriver USB](#)

In addition, the functions that collectively define the USB API have been changed. The new functions provide an improved interface between user-mode USB applications and the WinDriver kernel module. Note that the new functions receive their parameters directly, unlike the old functions, which received their parameters using a structure.

The table below lists the legacy functions in the left column and indicates in the right column which function or functions replace(s) each of the legacy functions. Use this table to quickly determine which new functions to use in your new code.

Low Level API	High Level API
<i>Previous Function</i>	<i>New Function</i>
WD_Open() , WD_Version() , WD_UsbScanDevice()	WDU_Init()
WD_UsbDeviceRegister()	WDU_SetInterface()
WD_UsbGetConfiguration()	WDU_GetDeviceInfo()
WD_UsbDeviceUnregister()	WDU_Uninit()
WD_UsbTransfer()	WDU_Transfer() , WDU_TransferDefaultPipe() , WDU_TransferBulk() , WDU_TransferIsoch() , WDU_TransferInterrupt()
USB_TRANSFER_HALT option	WDU_HaltTransfer()
WD_UsbResetPipe()	WDU_ResetPipe()
WD_UsbResetDevice() , WD_UsbResetDeviceEx()	WDU_ResetDevice()

5.11. FAQ

5.11.1 How do I reset my USB device using WinDriver?

You can use [WDU_ResetDevice\(\)](#).

Chapter 6

Using DriverWizard

This chapter describes the WinDriver DriverWizard utility and its hardware diagnostics and driver code generation capabilities.

6.1. An Overview

DriverWizard (included in the WinDriver toolkit) is a graphical user interface (GUI) tool that is targeted at two major phases in the hardware and driver development:

Hardware diagnostics

DriverWizard enables you to write and read hardware resources before writing a single line of code. After the hardware has been built, insert your device into the appropriate bus slot on your machine, view its resources and configuration - and verify the hardware's functionality.

Code generation

Once you have verified that the device is operating to your satisfaction, use DriverWizard generate skeletal driver source code with functions to view and access your hardware's resources.

If you are developing a driver for a device that is based on an enhanced-support PCI chipset (PLX 6466, 9030, 9050, 9052, 9054, 9056, 9080 and 9656; Altera Qsys, Avalon-MM designs; Xilinx BMD, XDMA, QDMA designs), we recommend that you first read [Enhanced Support for Specific Chipsets](#) to understand your development options.

On Windows, DriverWizard can also be used to generate an INF file for your hardware.

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 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 solution/makefile that you can use to automatically load all of the project information and files into your development environment.

6.2. DriverWizard Walkthrough

To use DriverWizard, follow these steps.

6.2.1. Attach your hardware to the computer

Attach the card to the appropriate bus slot on your computer.

For a virtual PCI device, you have the option to use DriverWizard to generate code without having the actual device installed, by selecting the **PCI Virtual Device** DriverWizard option (see information in Step 2). When selecting this option, DriverWizard will generate code for your virtual PCI device.

** Note**

Use the virtual PCI device option when you are unable to connect your PCI device to your development computer. This will generate code that is less customized for your device, but you will be able to later add the customizations for your device to the generated code manually.

6.2.2. Run DriverWizard and select your device

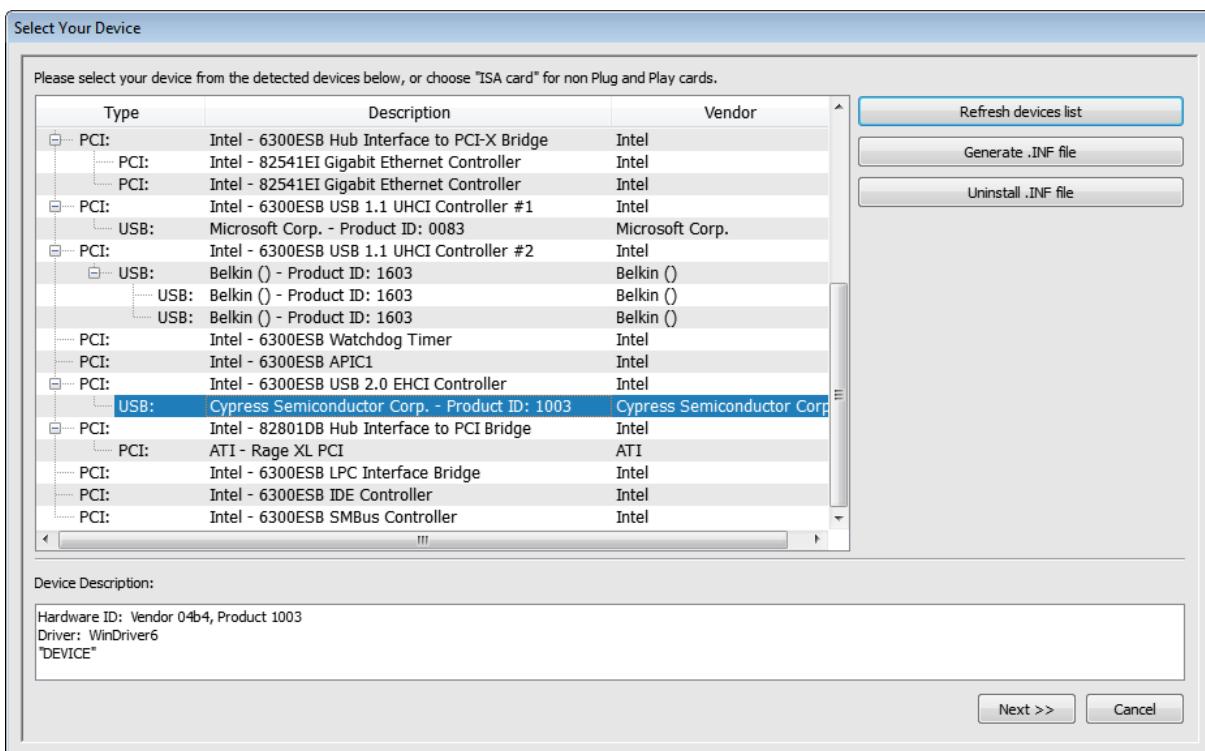
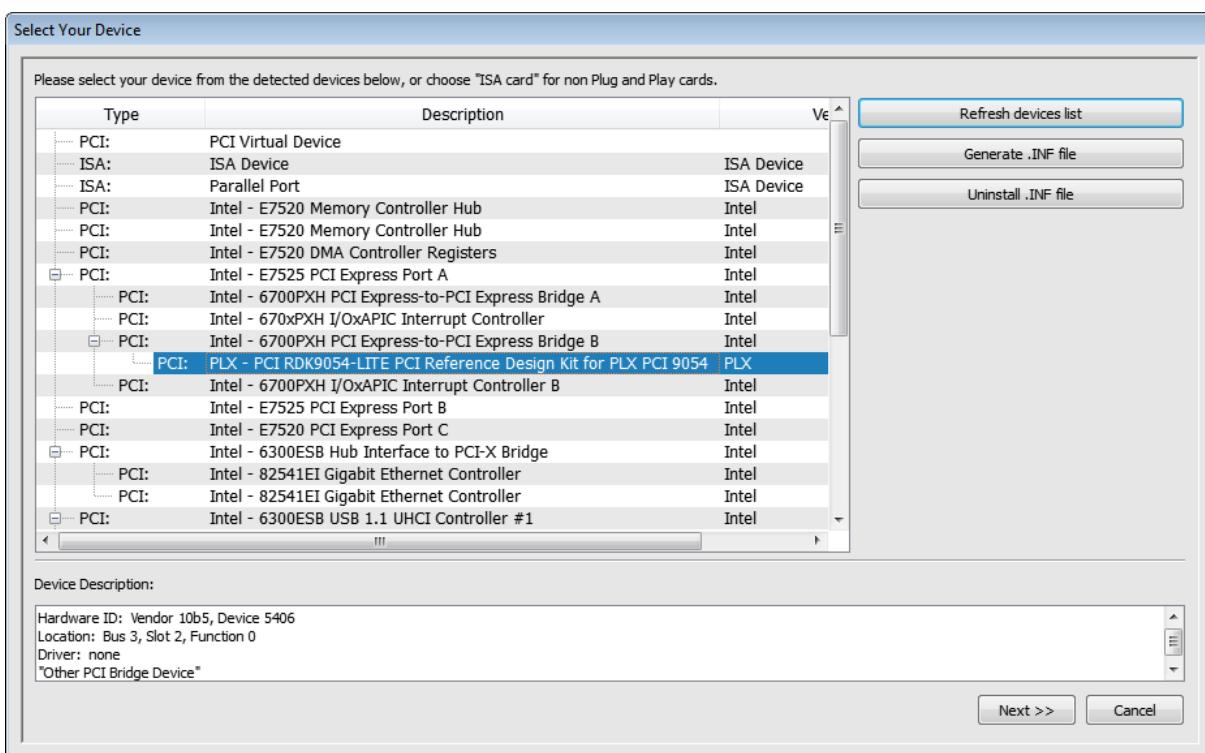
- Start DriverWizard - <path to WinDriver>/wizard/wdwizard. On Windows you can also run DriverWizard from the **Start** menu: **Start | Programs | WinDriver | DriverWizard**. On Windows and Linux you must run DriverWizard as administrator/root.
- Click **New host driver project** to start a new project, or Open an existing project to open a saved session.

Create or Open a Driver Project



- Select your Plug-and-Play card from the list of devices detected by DriverWizard.

Select Your 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. When selecting the PCI Virtual Device option, DriverWizard allows you to define the device's resources. By specifying the I/O and/or 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.

** Attention**

If a PCI device is attached physically to the development computer, The IRQ number and the size of the I/O and memory ranges are irrelevant, since these will be automatically detected by DriverWizard. Until WinDriver 14.40 - the maximum [WD_CARD_ITEMS] ([WD_CARD](#)) that were available was 20, from WinDriver 14.50 onwards the maximum is 128.

6.2.3. Generate and install an INF file for your device (Windows)

If you don't need to generate and install an INF file (e.g., if you are using DriverWizard on a non-Windows OS), skip this step.

On the supported Windows operating systems, the driver for Plug-and-Play devices is installed by installing an INF file for the device. DriverWizard enables you to generate an INF file that registers your device to work with WinDriver (i.e., with the `windrivr1511.sys` driver). The INF file generated by DriverWizard should later be distributed to your Windows customers, and installed on their PCs. The INF file that you generate in this step is also designed to enable DriverWizard to diagnose your device on Windows (for example, when no driver is installed for your PCI/USB device). Additional information concerning the need for an INF file is provided in [17.1.1. Why Should I Create an INF File?](#).

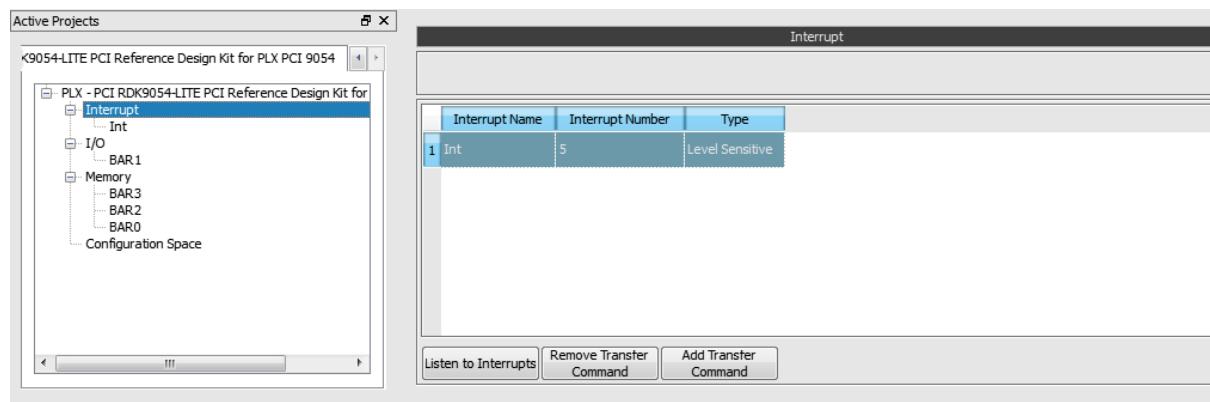
** Attention**

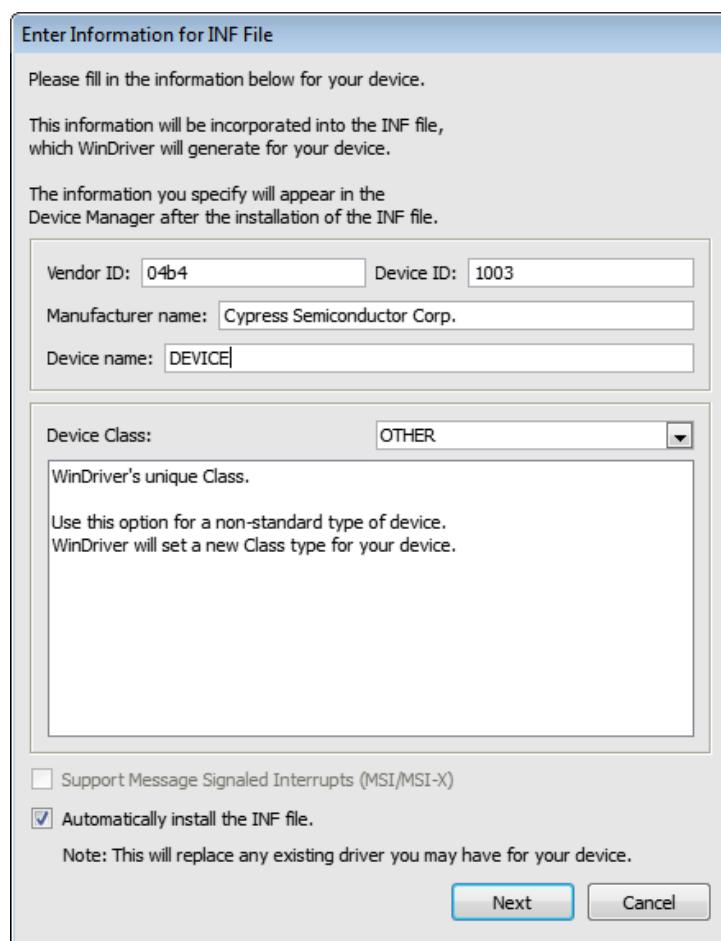
On Windows 10 or higher you must disable digital signature enforcement or enter test mode in order to install unsigned INF files generated by the DriverWizard. See [17.3.4. Temporary disabling digital signature enforcement in Windows 10](#). **Make sure that this was done prior to performing the following steps, otherwise they will not work.**

To generate and install the INF file with DriverWizard, do the following:

- In the Select Your Device screen (see Step 2), click the **Generate .INF file** button or click **Next**.
- DriverWizard will display information detected for your device - Vendor ID, Device ID, Device Class, manufacturer name and device name - and allow you to modify this information.

DriverWizard INF File Information





** Attention**

For multiple-interface USB devices, you can select to generate an INF file either for the composite device or for a specific interface. When selecting to generate an INF file for a specific interface of a multi-interface USB device the INF information dialogue will indicate for which interface the INF file is generated. When selecting to generate an INF file for a composite device of a multi-interface USB device, the INF information dialogue provides you with the option to either generate an INF file for the root device itself, or generate an INF file for specific interfaces, which you can select from the dialogue. Selecting to generate an INF file for the root device will enable you to handle multiple active interfaces simultaneously.

- 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. You can choose to automatically install the INF file by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation dialogue. If the automatic INF file installation fails, DriverWizard will notify you and provide manual installation instructions (refer also the manual INF file installation instructions in [17.1. Windows INF Files](#)).

For further information on Preallocating DMA Buffers in Windows, please refer to [11.2.2.4. Preallocating Contiguous DMA Buffers on Windows](#).

Handling of PCI Message-Signaled Interrupts (MSI) and Extended Message- Signaled Interrupts (MSI-X) requires specific configuration in the device's INF file, as explained in [10.1.7.1. Windows MSI/MSI-X Device INF Files](#).

On Windows, if your hardware supports MSI or MSI-X, the Support Message Signaled Interrupts option in the DriverWizard's INF generation dialogue will be enabled and checked by default. When this option is checked, the generated DriverWizard INF file for your device will include support for MSI/ MSI-X handling. However, when this option is not checked, PCI interrupts will be handled using the legacy level-sensitive interrupts method, regardless of whether the hardware and OS support MSI/MSI-X.

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

6.2.4. Uninstall the INF file of your device (Windows)

On Windows, you can use DriverWizard to uninstall a previously installed device INF file. This will unregister the device from its current driver and delete the copy of the INF file in the Windows INF directory.

In order for WinDriver to correctly identify the resources of a Plug-and-Play device and communicate with it - including for the purpose of the DriverWizard device diagnostics outlined in the next step - the device must be registered to work with WinDriver via an INF file.

If you do not wish to uninstall an INF file, skip this step.

To uninstall the INF file, do the following:

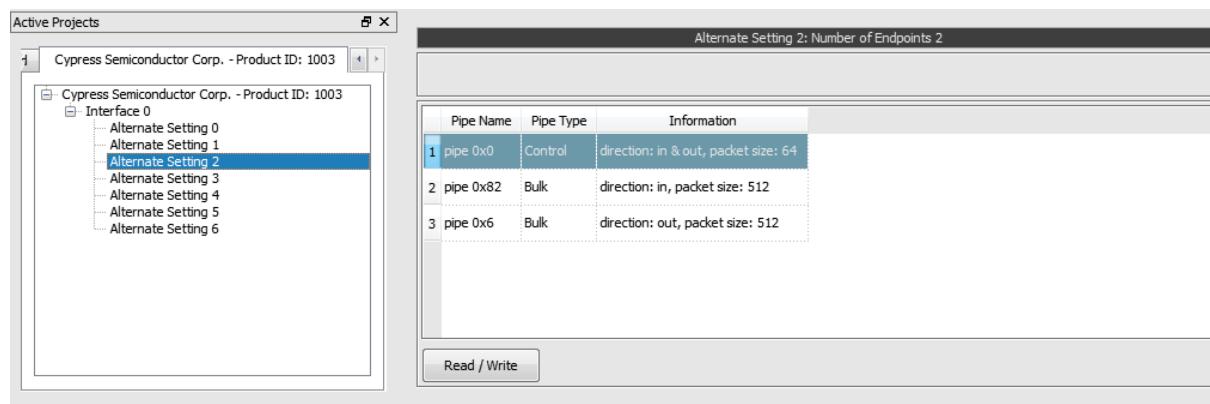
- In the Select Your Device screen (see Step 2), click the **Uninstall .INF file** button.
- Select the INF file to be removed.

6.2.5. Select the desired alternate setting (USB)

Skip this step if your device is not a USB device

DriverWizard detects all the device's supported alternate settings and displays them. Select the desired alternate setting from the displayed list.

Select Device Interface



DriverWizard will display the pipes information for the selected alternate setting. For USB devices with only one alternate setting configured, DriverWizard automatically selects the detected alternate setting and therefore the Select Device Interface dialogue will not be displayed.

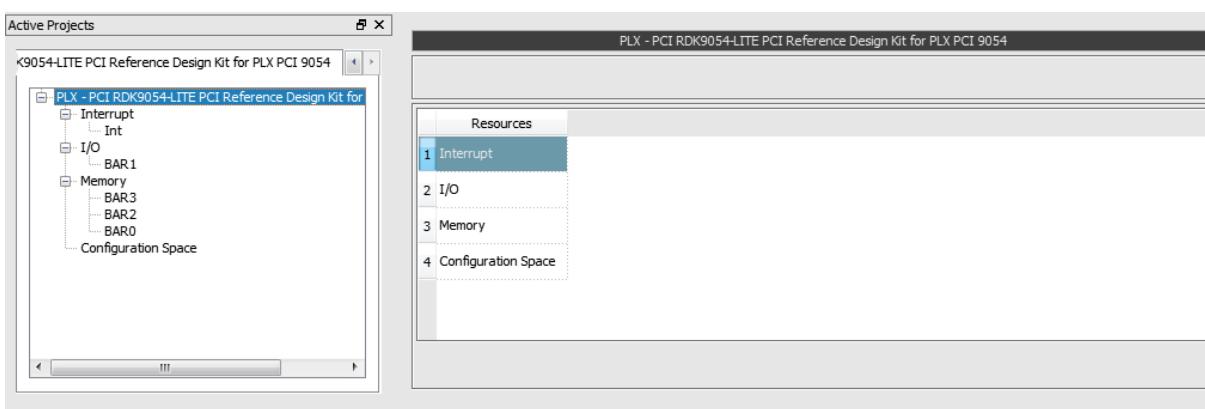
6.2.6. Diagnose your device (PCI)

Skip this step if your device is not a PCI 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.

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

- DriverWizard will automatically detect your Plug-and-Play hardware resources: I/O ranges, memory ranges, and interrupts.

PCI Resources

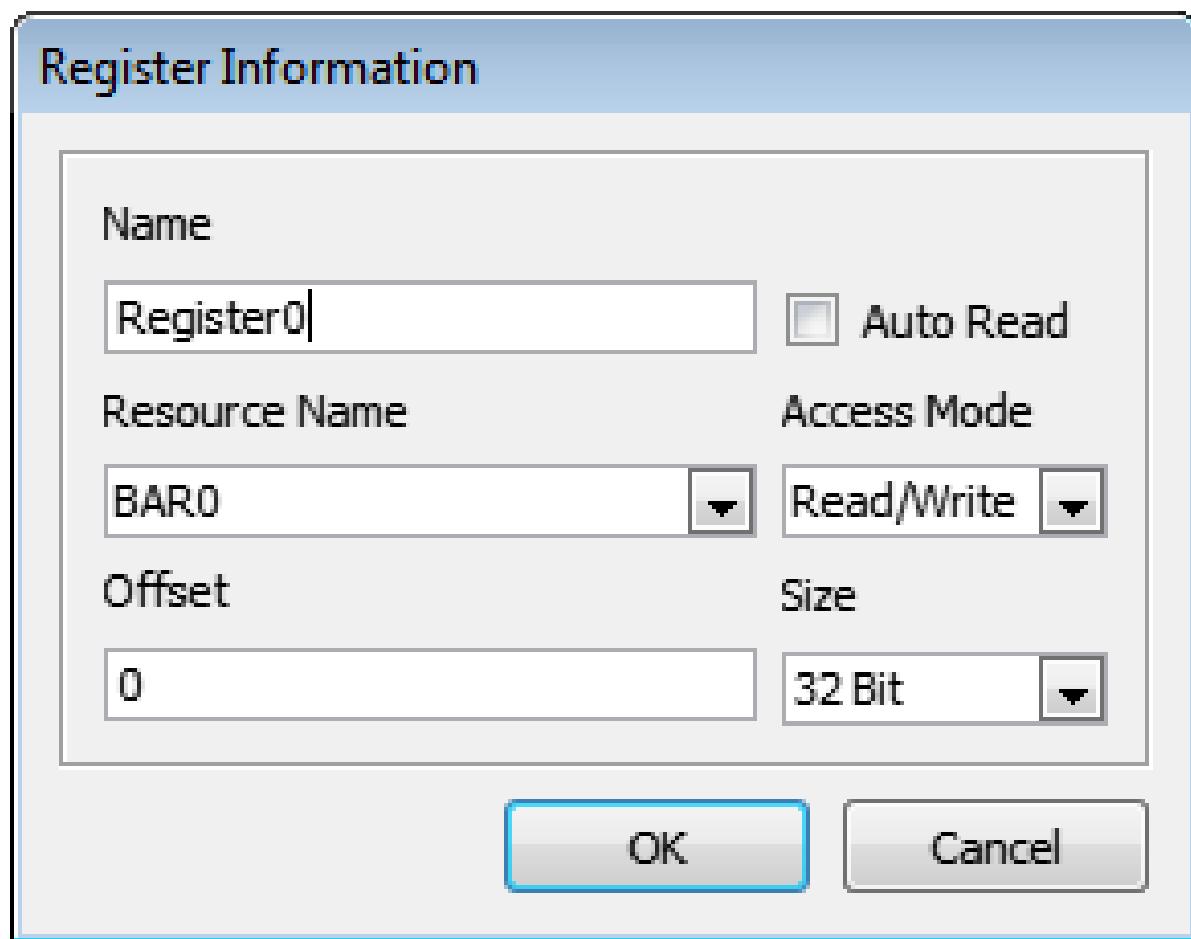


For non-Plug-and-Play hardware, define your hardware's resources manually.

On Windows, you may need to register an IRQ with WinDriver before you can assign it to your non-Plug-and-Play hardware see [10.1.3. Registering IRQs for Non-Plug-and-Play Hardware](#).

You can also manually define hardware registers, as demonstrated below.

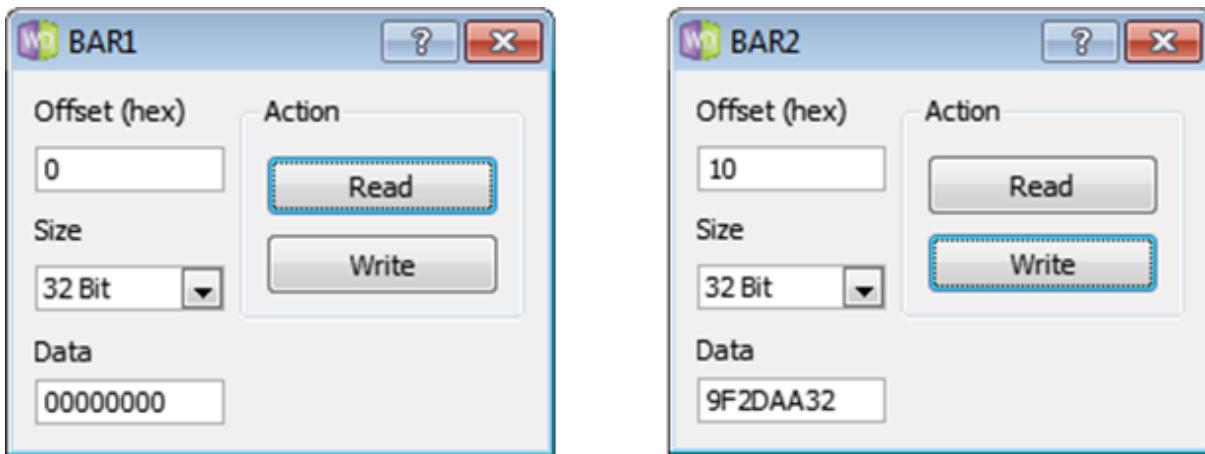
Define Registers



When defining registers, you may check the **Auto Read** box in the **Register Information** window. Registers marked as **Auto Read** will automatically be read for any register read/write operation performed from DriverWizard. The read results will be displayed in the wizard's Log window.

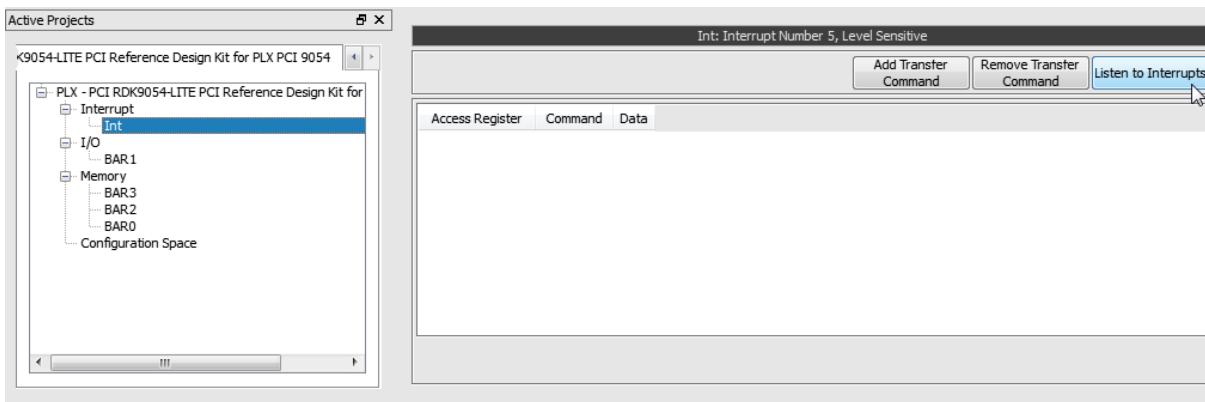
- Read and write to the I/O ports, memory space and your defined registers, as demonstrated below.

Read/Write Memory and I/O



- 'Listen' to your hardware's interrupts.

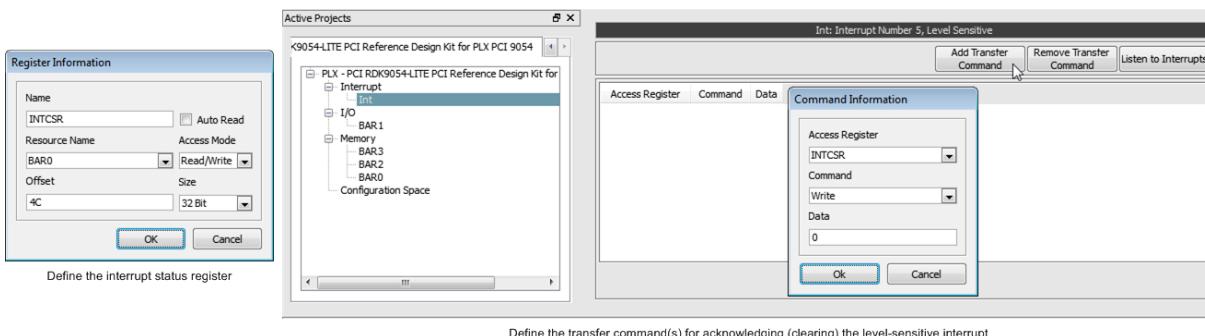
Listen to Interrupts



For level-sensitive interrupts, such as legacy PCI interrupts, you must use DriverWizard to define the interrupt status register and assign the read/write command(s) for acknowledging (clearing) the interrupt, before attempting to listen to the interrupts with the wizard, otherwise the OS may hang!

Figure below demonstrates how to define an interrupt acknowledgment command for a defined INTCSR hardware register. Note, however, that interrupt acknowledgment information is hardware-specific.

Define Transfer Commands for Level-Sensitive Interrupts



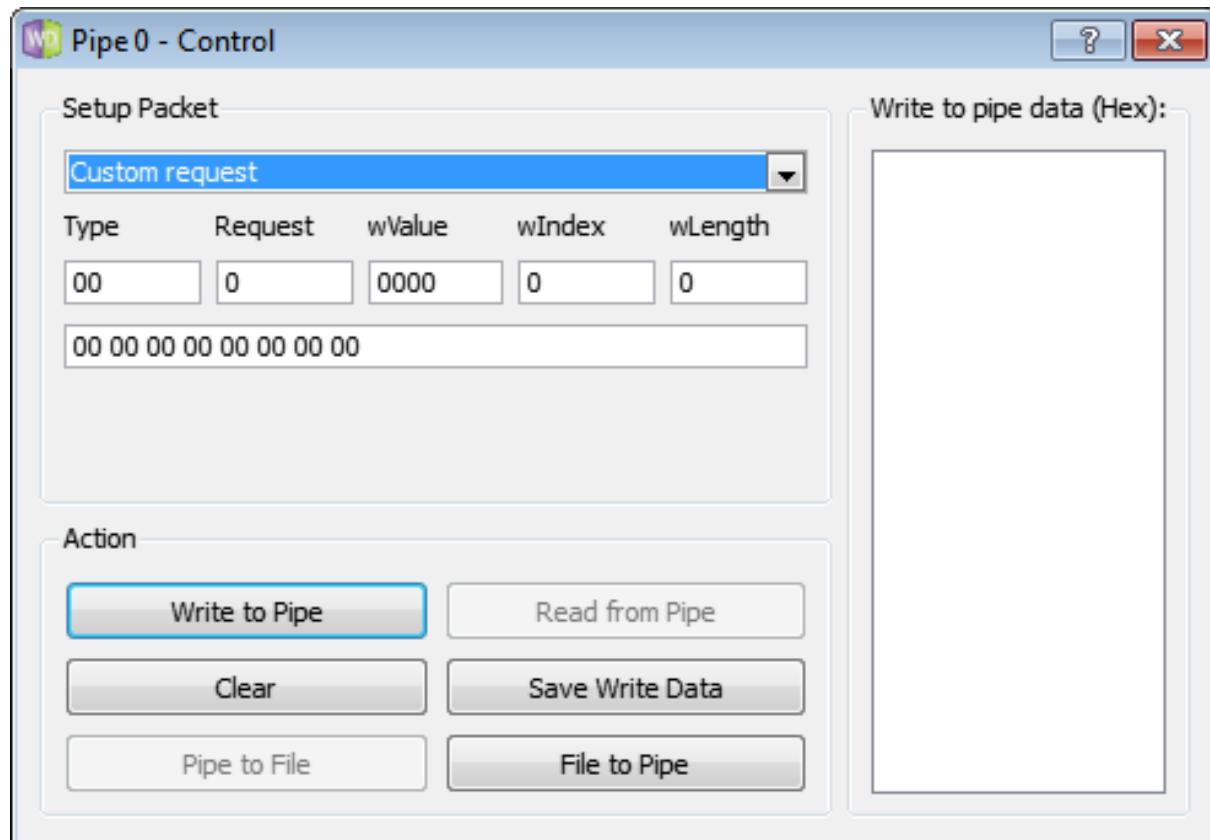
6.2.7. Diagnose your device (USB)

Skip this step if your device is not a USB 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.

- Test your USB device's pipes DriverWizard shows the pipes detected for the selected alternate setting. To perform USB data transfers on the pipes, follow these steps:

- Select the desired pipe.
- For a control pipe (a bidirectional pipe), click Read / Write. A new dialogue will appear, allowing you to select a standard USB request or define a custom request, as demonstrated below.

USB Control Transfers

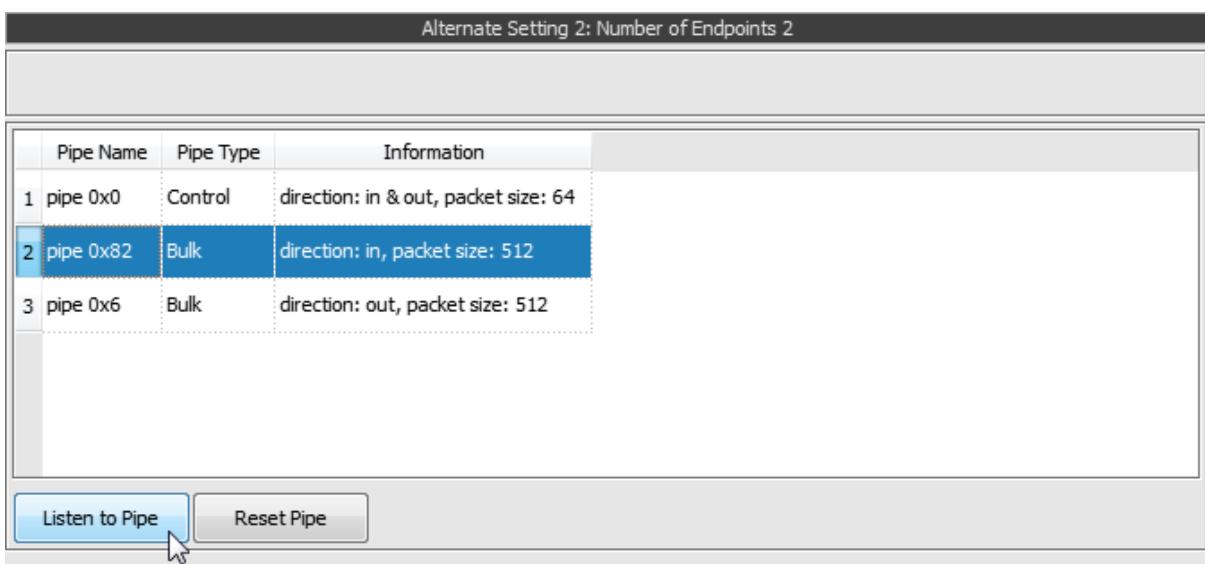


When you select one of the available standard USB requests, the setup packet information for the selected request is automatically filled and the request description is displayed in the Request Description box. For a custom request, you are required to enter the setup packet information and write data (if exists) yourself. The size of the setup packet should be eight bytes and it should be defined using little endian byte ordering. The setup packet information should conform to the USB specification parameters (bmRequestType, bRequest, wValue, wIndex, wLength).

For an input pipe (moves data from device to host) click **Listen to Pipe**. To successfully accomplish this operation with devices other than HID, you need to first verify that the device sends data to the host. If no data is sent after listening for a short period of time, DriverWizard will notify you that the Transfer Failed.

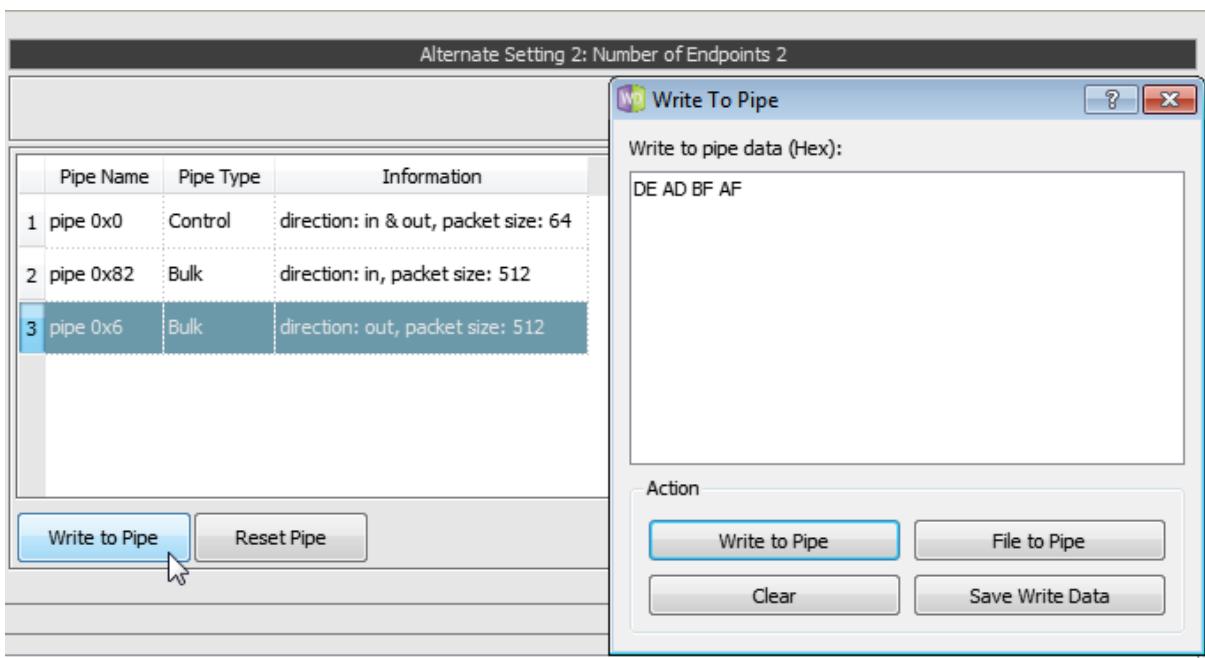
To stop reading, click **Stop Listen to Pipe**.

Listen to Pipe



For an output pipe (moves data from host to device), click **Write to Pipe**. A new dialogue box will appear asking you to enter the data to write. The DriverWizard log will contain the result of the operation.

Write to Pipe



You can reset input and output pipes by pressing the **Reset Pipe** button for the selected pipe.

6.2.8. Import Register Information from CSV file

DriverWizard allows users to import register information from a Comma Separated Values(CSV) file instead of having to type it manually for each register. This register information is used later in order to automatically generate code.

Use a text or spreadsheet editor to create a CSV in the following format:

- The first line must be: Name,Resource Name,Offset,Access Mode,Size
- The next lines must include the following values formatted the following way (the following line is an example of one register): MY_REGISTER,BAR2,200,0,32

- Name: The register's desired name.
 - Resource Name: The name of the resource as it appears in DriverWizard.
 - Offset: The register's offset in the resource.
 - Access Mode: Allowed values are: 0 = Read/Write, 1 = Read, 2 = Write
 - Size: Size of register.
- Choose any resource from the DriverWizard side tab (a BAR, etc...)
 - Click the "Import registers" on the top left tab, browse and choose your .csv file. Registers should be added to the DriverWizard resource tabs. If the input file was faulty, DriverWizard will alert and still try to add as much registers it was able to.

6.2.9. Samples And 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.

6.2.9.1. Samples Or Code Generation

You can generate code by selecting this option either via DriverWizard's Generate Code toolbar icon or from the Wizard's **Project | Generate Code** menu. After you select a code generation option of your choice, DriverWizard will generate the source code for your driver, and save it together with the Wizard driver-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 code generation dialogue.

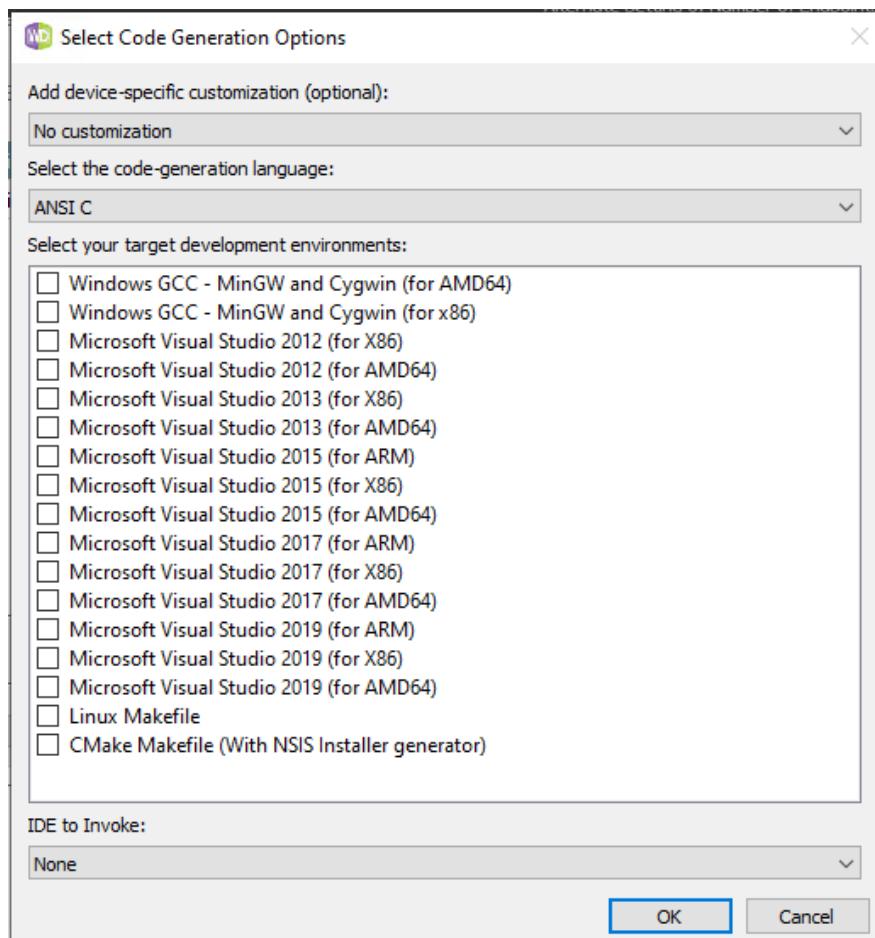
** Attention**

Close DriverWizard to avoid device overlap errors.

Use this code as a starting point for your device driver. Modify where needed to perform your driver's specific functionality.

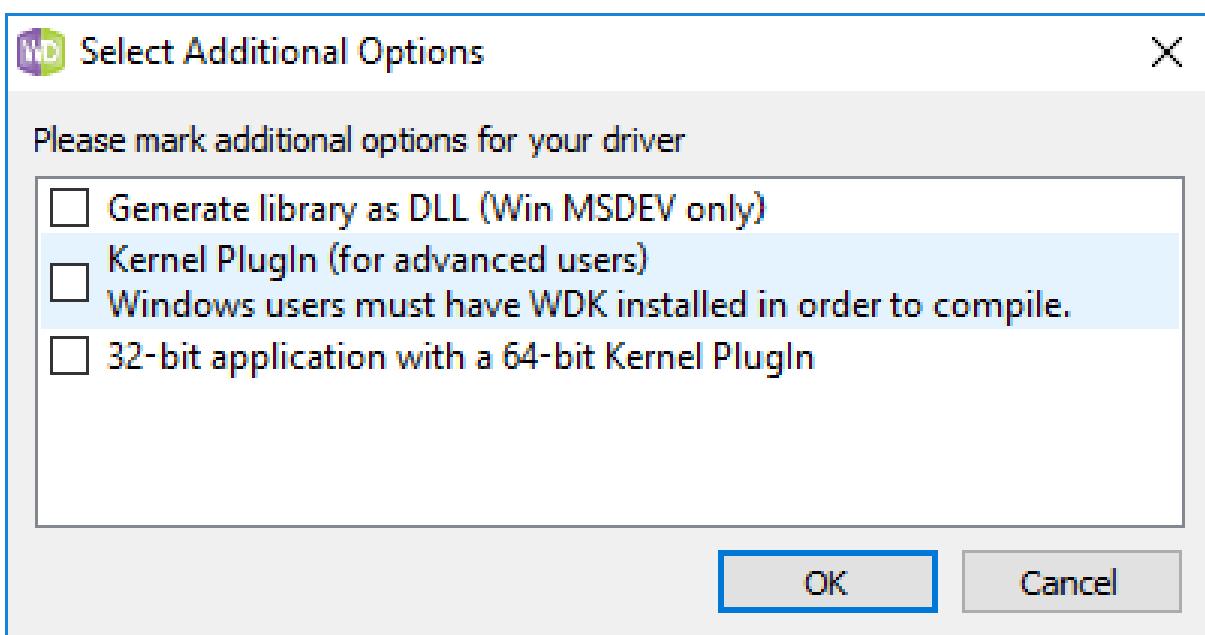
If you choose to generate code either via the **Generate Code** toolbar icon or from the **Generate Code** menu, in the **Select Code Generation Options** dialogue box that will appear, you may optionally select to generate additional customized code for one of the supported devices. Then you choose the code language and development environment(s) for the generated code and select **Next** to generate the code.

PCI Code Generation Options



Then click **Next** and select whether to handle Plug-and-Play and power management events from within your driver code, whether to generate Kernel Plugin code (see [Understanding the Kernel Plugin](#)) (and what type of related application to create), and whether to build your project's library as a DLL (for MS Visual Studio Windows projects).

PCI Additional Driver Options



Kernel Plugin Windows Project Notes:

- To compile the generated Kernel Plugin code, the Windows Driver Kit (WDK) must be installed.
- To successfully build a Kernel Plugin project using MS Visual Studio, the path to the project directory must not contain any spaces.
- Save your project (if required) and click OK to open your development environment with the generated driver.
- Close DriverWizard to avoid device overlap errors.

After you choose the code, you can compile and run it:

- 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 a variety of compilers, and will run on all supported platforms without modification.

In addition, WinDriver includes a variety of samples that demonstrate how to use WinDriver's API to communicate with your device and perform various driver tasks:

- C samples: found under the `WinDriver/samples/c` directory.
- Python samples: found under the `WinDriver/samples/python` directory.
- Java samples: found under the `WinDriver/samples/java` directory.
- C#.NET samples: found under the `WinDriver/samples/csharp.net` directory.
- Visual Basic.NET samples (Windows): found under the `WinDriver/samples/vb.net` directory.
- .NET PowerShell samples: found under the `WinDriver/samples/powershell` directory.

6.2.9.2. The Generated PCI/ISA and USB C 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.

** Attention**

For USB you will only have a `xxx_diag.c` source file. This file implements a diagnostic USB application, which demonstrates how to use WinDriver's USB API to locate and communicate with your USB device(s), including detection of Plug-and-Play events (device insertion/removal, etc.), performing read/write transfers on the pipes, resetting the pipes and changing the device's active alternate setting. The generated application supports handling of multiple identical USB devices.

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` - the implementation of the hardware-specific API (declared in `xxx_lib.h`), using the WinDriver Card (WDC) API .
 - `xxx_lib.h` - a 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 to use the API generated by 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 suits your needs.

6.2.9.3. The Generated PCI/ISA Python Code

After generating Python code in the DriverWizard, in the source code directory you now have a new `xxx_lib.py` file, which contains type definitions and function implementations for the API created for you by the DriverWizard. In addition, you will find an `xxx_diag.py` 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 `wplib` subdirectory includes shared Python code files that implement the WinDriver API structs, import the WinDriver user API DLL file, define variables, and implement user I/O functions required for the user application to run. 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.py` - type definitions and the implementation of the hardware-specific API, using the WinDriver Card (WDC) API.
- A diagnostics program that utilizes the generated DriverWizard API (declared in `xxx_lib.py`) to communicate with your device(s): `xxx_diag.py` - 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, run it with your favorite Python interpreter. Change the function `main()` of the program so that the functionality suits your needs.

6.2.9.4. The Generated USB Python Code

After generating Python code in the DriverWizard, in the source code directory you now have a new `xxx_diag.py` source file (where `xxx` is the name you selected for your DriverWizard project). This file implements a diagnostic USB application, which demonstrates how to use WinDriver's USB API to locate and communicate with your USB device(s), including detection of Plug-and-Play events (device insertion/removal, etc.), performing read/write transfers on the pipes, resetting the pipes and changing the device's active alternate setting. The generated application supports handling of multiple identical USB devices.

The `wplib` subdirectory includes shared Python code files that implement the WinDriver API structs, import the WinDriver user API DLL file, define variables, and implement user I/O functions required for the user application to run.

6.2.9.5. The Generated PCI/ISA Java Code

After generating Java code in the DriverWizard, in the source code directory you now have a new `xxxLib.java` file, which contains type definitions and function implementations for the API created for you by the DriverWizard. In addition, you will find an `xxxDiag.java` 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 `com\jungo\shared` subdirectory and the `xxx_installation\redist\wdapi_java1511.jar` includes shared Java code files that implement the WinDriver API structs, import the WinDriver user API DLL file, define variables, and implement user I/O functions required for the user application to run.

A very important file is the Java Native Interface (JNI) file `wdapi_java1511.dll` that provides the interface between the Java layer and your native `wdapi1511.dll` file provided with WinDriver. Make sure that the JNI file corresponds with the architecture and bits of the Java Virtual Machine (JVM) you'll be using to run the application.

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): `xxxLib.java` - type definitions and the implementation of the hardware-specific API, using the WinDriver Card (WDC) API.
- A diagnostics program that utilizes the generated DriverWizard API (declared in `xxxLib.java`) to communicate with your device(s): `xxxDiag.java` - 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, you can either compile and run it from the command line as described below or open it with the Eclipse IDE. Change the function `main()` of the program so that the functionality suits your needs.

6.2.9.6. The Generated USB Java Code

After generating Java code in the DriverWizard, in the source code directory you now have a new `xxxDiag.java` source file (where `xxx` is the name you selected for your DriverWizard project). This file implements a diagnostic USB application, which demonstrates how to use WinDriver's USB API to locate and communicate with your USB device(s), including detection of Plug-and-Play events (device insertion/removal, etc.), performing read/write transfers on the pipes, resetting the pipes and changing the device's active alternate setting. The generated application supports handling of multiple identical USB devices.

The `com\jungo\shared` subdirectory and the `xxx_installation\redist\wdapi_java1511.jar` includes shared Java code files that implement the WinDriver API structs, import the WinDriver user API DLL file, define variables, and implement user I/O functions required for the user application to run.

A very important file is the Java Native Interface (JNI) file `wdapi_java1511.dll` that provides the interface between the Java layer and your native `wdapi1511.dll` provided with WinDriver. Make sure that the JNI file corresponds with the architecture and bits of the Java Virtual Machine (JVM) you'll be using to run the application.

After creating your code, you can either compile and run it from the command line as described below, or open it with the Eclipse IDE.

6.2.9.7. The Generated PCI/ISA and USB .NET Code

You can use WinDriver's DriverWizard utility to generate C# (USB and PCI) and Visual Basic.NET (USB only). To develop a C# driver with WinDriver, either use DriverWizard to generate a driver application for your device, or use the WinDriver .NET samples, which utilize the WinDriver .NET API DLL, that most matches your design, and then modify the generated/sample code in accordance with your hardware specification and desired driver functionality. Alternatively, you can use the generated/sample code as a reference for writing your own WinDriver C# driver:

PCI:

- The `WinDriver\samples\csharp.net\pci_sample` directory contains a .NET PCI library (`pci←_lib.dll`) and a sample GUI-based PCI diagnostics application (`pci_sample.exe`), both implemented in C#. This sample is based on the Windows Forms framework, only supported natively on Windows.
- The `WinDriver\samples\c\plx\dotnet` directory contains a C# library (`plx_lib_dotnet.←_dll`) and sample diagnostics application (`PLX_Sample.exe`), designed specifically for handling PLX devices. This sample is based on the Windows Forms framework, only supported natively on Windows.

- The `WinDriver\samples\csharp\pci_diag` directory contains a .NET PCI Console sample similar to the `console pci_diag` sample Jungo provides in other languages. This sample is supported on Windows, Linux and MacOS.

USB:

- The `WinDriver\samples\csharp.net\usb_sample` directory contains a .NET USB library (`usb_lib_dotnet.dll`) and a sample USB diagnostics application (`csharp_usb_sample.exe`), both implemented in C#.
- The `WinDriver\samples\vb.net\usb_sample` directory contains a sample .NET USB diagnostics application (`vb_usb_sample.exe`), implemented in VB.NET. This sample is similar to the sample C# USB diagnostics application and also uses the sample C# USB library (`usb_lib_dotnet.dll`).

If using the DriverWizard in the source code directory you now have a new `lib` subdirectory, which contains type definitions and function implementations for the API created for you by the DriverWizard. This subdirectory outputs a DLL file that is compiled by the `xxx_diag` project. In addition, you will find a `diag` subdirectory, a sample diagnostics application that utilizes the generated DriverWizard API to communicate with your device. The `xxx_diag.cs` file that includes a `Main()` function. A list of all files created can be found at `xxx_files.txt`.

After creating your code, compile it with your favorite C# compiler. Change the function `Main()` of the program so that the functionality suits your needs.

6.3. Compiling the Generated Code

6.3.1. C/C#/VB Windows Compilation

As explained above, on Windows you can select to generate project, solution, and make files for the supported compilers and development environments - MS Visual Studio, Windows GCC(MinGW/Cygwin). For integrated development environments (IDEs), such as MS Visual Studio, you can also select to automatically invoke your selected IDE from the wizard. You can then proceed to immediately build and run the code from your selected IDE.

You can also build the generated code using any other compiler or development environment that supports the selected code language and target OS. Simply create a new project or make file for your selected compiler/environment, include the generated source files, and run the code. For Windows, the generated compiler/environment files are located under an `x86` directory - for 32-bit projects - or an `amd64` directory - for 64-bit projects. To build a Kernel PlugIn project (on Windows), follow the instructions in [13.8.1. Windows Kernel PlugIn Driver Compilation](#).

6.3.1.1. C Compilation with a Linux Makefile

Use the makefile that was created for you by DriverWizard in order to build the generated code using your favorite compiler, preferably GCC. To build a Kernel PlugIn project, follow the instructions in [13.8.2. Linux Kernel PlugIn Driver Compilation](#).

6.3.1.2. C Compilation with CMake

The recommended way is using CMake, which simplifies compilation and portability between IDEs and platforms.

** Attention**

MacOS Code generation with WinDriver works only with CMake.

Open a terminal window:

```
# Go to directory of the generated code
$ cd <GENERATED_CODE_DIRECTORY>
# Prepare makefile
$ cmake .
# Compile
$ make
```

6.3.1.2.1 Generating a WinDriver C project for Xcode (MacOS)

On MacOS, to create an XCode project instead of a make project, run CMake in the following manner:

```
$ cmake . -G Xcode .
```

6.3.1.2.2 Generating a WinDriver C project for MinGW (Windows)

Although compiling on Windows is preferable with Visual Studio, your project can also be compiled using MinGW.

Compile the wdapi shared library:

```
$ cd WinDriver1511\src\wdapi
$ mkdir build
$ cd build
$ cmake .. -G "MinGW Makefiles" -DCMAKE_C_COMPILER="gcc" -DCMAKE_MAKE_PROGRAM="mingw32-make"
$ mingw32-make
```

Copy the library to the bin directory of your MinGW installation folder (i.e C:\msys64\mingw64\bin):

```
$ cp WIN32\libwdapi1511* C:\msys64\mingw64\bin
```

In PATH_TO_YOUR_PROJECT/CMakeLists.txt, change the line target_link_libraries(your←_project \${WDAPI_LIB}) to:

```
find_library(WDAPI libwdapi1511.dll.a)
target_link_libraries(xxx_diag ${WDAPI})
```

Compile your user application:

```
$ cmake .. -G "MinGW Makefiles" -DCMAKE_C_COMPILER="gcc" -DCMAKE_MAKE_PROGRAM="mingw32-make"
$ mingw32-make
```

the program should compile and be able to run now.

6.3.2. Java Compilation

6.3.2.1. Opening the DriverWizard generated Java code with Eclipse IDE

Please follow these steps:

- Make sure you have a Java Development Kit (JDK) and the Eclipse IDE installed and set up for Java Development. Under Linux make sure to run Eclipse as sudo.
- Go to **File | New | Java Project**.
- Uncheck “Use default location”.
- Click on “Browse” and select the directory of the generated code. For example WinDriver\my_projects\xxx.
- Click on “Next”.
- Click on “Finish”. You will now be able to compile, run and edit your code.

6.3.2.2. Compiling and running the DriverWizard generated Java code from the command line

Please follow the next steps:

- Make sure you have a Java Development Kit (JDK) installed.
- Make sure your \$PATH environment variable contains the path to the JDK's bin subdirectory (for example C:\Program Files \Java\jdkX.X.X_XXX\bin, replace the Xs with your JDK version).

In Windows:

```
# set PATH=%PATH%;"C:\Program Files\Java\jdkX.X.X_XXX\bin\"
```

Compile the generated code:

In Windows:

```
# javac -cp ".;xxx_installation/redist/wdapi_javal511.jar" xxxDiag.java
```

In Linux/MacOS:

```
# sudo javac -cp .:xxx_installation/lib/wdapi_java1511.jar xxxDiag.java
```

Run the generated code:

In Windows:

```
# java -cp ".;xxx_installation/redist /wdapi_java1511.jar" -Djava.library.path=xxx_installation/redist/xxxDiag.java
```

In Linux/MacOS:

```
# sudo java -Djava.library.path=xxx_installation/lib/ -cp .:xxx_installation/lib/wdapi_java1511.jar xxxDiag
```

6.4. FAQ

6.4.1. If a variable requires a pointer to be assigned to it, as in pBuffer = &dwVal, how do I do it in C# dotNET/Java/Python?

The main difference between C#/Java/Python and C is that the former languages have a garbage collector which automatically manages the memory, whereas C is totally "manual" - meaning that it is the user's responsibility to allocate and free memory from the heap (using [malloc\(\)](#) and [free\(\)](#)). Some WinDriver APIs that are written in C require usage of unmanaged memory.

For each "managed" language that WinDriver supports, WinDriver provides a sample of this kind of use:

- C#.NET: WinDriver/samples/csharp.net/pci_sample/lib/dma.cs ([Using the System.Runtime.InteropServices.Marshal class](#))
- Python: WinDriver/samples/python/pci_diag.py ([Using global variables and the ctypes class](#))
- Java: WinDriver/samples/java/com/jungo/PciDiag.Java ([Using the java.nio.ByteBuffer class](#)).

Chapter 7

Developing a Driver

This chapter takes you through the WinDriver driver development cycle.

7.1. Using DriverWizard to Build a Device Driver

You can use DriverWizard to diagnose your device and verify that it operates as expected, as well as to generate skeletal code for your device in C, C#, Visual Basic (USB Only), Java and Python. For more information about DriverWizard, refer to [Using DriverWizard](#).

** Attention**

For devices based on the Altera Qsys, Avalon-MM designs, Xilinx BMD, XDMA, QDMA designs, you can use DriverWizard to generate customized device-specific code, which utilizes the enhanced-support sample APIs. For additional information, refer to [Enhanced Support for Specific Chipsets](#). This is the most recommended way to develop a driver using WinDriver as it saves you the most time and hassle.

Use any C/C#.NET/Python/Java compiler/interpreter or development environment (depending on the code you created) to build the skeletal driver you need. WinDriver provides specific support for the following environments and compilers: MS Visual Studio, CMake, GNU Make/Eclipse (for Java projects).

That is all you need to do in order to create your user-mode driver. If you discover that better performance is needed, refer to [Improving PCI Performance](#).

To learn how to perform operations that DriverWizard cannot automate, refer to [PCI Advanced Features](#).

7.2. Using a code sample to Build a Device Driver

If you are using an enhanced-support PCI device (PLX 6466, 9030, 9050, 9052, 9054, 9056, 9080 and 9656; Altera Qsys, Avalon-MM designs; Xilinx BMD, XDMA, QDMA designs), you may want to use the related WinDriver sample as the basis for your development instead of generating code with DriverWizard.

This method is relevant if you are unable to get DriverWizard to work on your platform or if it is not available for your platform and you wish to quickly start developing.

These samples can be found in 'WinDriver/samples/c'. These directories include both the source code and project files to compile them, and also a precompiled version of the sample source code.

** Attention**

Note that after your evaluation period has expired the precompiled binaries of the sample will not work, and you will have to recompile them with your new license key.

7.3. Writing the Device Driver Without DriverWizard

There may be times when you choose to write your driver directly, without using DriverWizard.

In such cases, either follow the steps outlined in this section to create a new driver project, or select a WinDriver sample that most closely resembles your target driver and modify it to suit your specific requirements.

This method is recommended only for expert users that are already familiar with the WinDriver API and the required compilation flags needed in order to compile WinDriver.

7.3.1. Include the Required WinDriver Files

Firstly, include the relevant WinDriver header files in your driver project. All header files are found under the `WinDriver/include` or `WinDriver/samples/c/shared` (for C projects) directory. All WinDriver projects require the `windrvr.h` header file.

When using the `WDC_xxx` API / `WDU_xxx` API , include the `wdc_lib.h` / `wdu_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/c/shared` directory, which provide convenient diagnostics functions.)

Then include the relevant header files from your source code. For example, to use API from the `windrvr.h` header file, add the following line to the code:

```
#include "windrvr.h"
```

Afterwards, link your code with the WDAPI library (Windows) / shared object (Linux):

- For Windows: `WinDriver/lib/amd64/wdapi1511.lib`, for compiling 64-bit binaries for x64 platforms, or `WinDriver/lib/amd64/x86/wdapi1511_32.lib` for compiling 32-bit binaries for x64 platforms
- For Linux: From the `WinDriver/lib` directory - `libwdapi1511.so` (for 64-bit binaries for x64 platforms) or `libwdapi1511_32.so` (for 32-bit applications targeted at 64-bit platforms).
- For MacOS: From the `WinDriver/lib` directory - `libwdapi1511.dylib`

(note that WinDriver for MacOS currently doesn't support 32 bit applications on 64-bit platforms).

You can also include the library's source files in your project instead of linking the project with the library. The C source files are located under the `WinDriver/src/wdapi` directory.

When linking your project with the WDAPI library/framework/shared object, you will need to distribute this binary with your driver.

- For Windows, get `wdapi1511.dll` / `wdapi1511_32.dll` (for 32-bit applications targeted at 64-bit platforms) from the `WinDriver/redist` directory.
- For Linux, get `libwdapi1511.so` / `libwdapi1511_32.so` (for 32-bit applications targeted at 64-bit platforms) from the `WinDriver/lib` directory.
- For MacOS, get `libwdapi1511.dylib` from the `WinDriver/lib` directory.

** Note**

WinDriver for MacOS currently doesn't support 32 bit applications on 64-bit platforms.

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

7.3.2. Write Your Code (PCI/ISA)

This section outlines the calling sequence when using the `WDC_xxx` API:

- Call `WDC_DriverOpen()` 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).
 - For PCI/PCI Express devices, call `WDC_PciScanDevices()` to scan the PCI bus and locate your device.
 - For PCI/PCI Express devices, call `WDC_PciGetDeviceInfo()` to retrieve the resources information for your selected device.
 - For ISA devices, define the resources yourself within a `WD_CARD` structure.

- Call [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) (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.
- Communicate with the device using the WDC_xxx API.
 - To enable interrupts, call [WDC_IntEnable\(\)](#).
 - To register to receive notifications for Plug-and-Play and power management events, call [WDC_EventRegister\(\)](#).
- When you are done:
 - Call [WDC_IntDisable\(\)](#) to disable interrupt handling (if previously enabled)
 - Call [WDC_EventRegister\(\)](#) to unregister Plug-and-Play and power management event handling (if previously registered)
 - Call [WDC_PciDeviceClose\(\)](#) / [WDC_IsaDeviceClose\(\)](#) (depending on your device) in order to close the handle to the device.
 - Call [WDC_DriverClose\(\)](#) to close the handles to WinDriver and the WDC library.

7.3.3. Write Your Code (USB)

This section outlines the calling sequence when using the WDU_xxx / WDC_xxx API:

- Call [WDC_DriverOpen\(\)](#) 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).
- Call [WDU_Init\(\)](#) at the beginning of your program to initialize WinDriver for your USB device, and wait for the device-attach callback. The relevant device information will be provided in the attach callback.
- Communicate with the device using the WDU_xxx API.
 - Once the attach callback is received, you can start using one of the [WDU_Transfer\(\)](#) functions family to send and receive data.
- When you are done:
 - Call [WDU_Uninit\(\)](#) to unregister from the device.
 - Call [WDC_DriverClose\(\)](#) to close the handles to WinDriver and the WDC library.

7.3.4. Configure and Build Your Code

After including the required files and writing your code, make sure that the required build flags and environment variables are set, then build your code.

** Attention**

When developing a driver for a 64-bit platform, your project or makefile must include the KERNEL_64BIT preprocessor definition. In the makefiles, the definition is added using the -D flag: -DKERNEL_64BIT. The sample and wizard-generated Linux and Windows GCC makefiles and the Windows MS Visual Studio projects, in the 64-bit WinDriver toolkit, already include this definition.

Before building your code, verify that the WD_BASEDIR environment variable is set to the location of the WinDriver installation directory. On Windows and Linux you can define the WD_BASEDIR environment variable globally: for Windows - refer to the Windows WD_BASEDIR note in [3.2.1. Windows WinDriver Installation Instructions](#); for Linux - refer to [3.2.2. Linux WinDriver Installation Instructions](#).

** Attention**

>If you have renamed your driver make sure you've added -DWD_DRIVER_NAME_CHANGE to your makefile (More information on Driver Renaming is available on [Driver Installation - Advanced Issues](#)).

7.4. Upgrading a Driver

This section is meant for users that already have code written using previous versions of WinDriver and wish to upgrade it to link against the current version of WinDriver.

7.4.1. Regenerating code and gradually merging

If your code is very old, the **most recommended** method is to generate a new project for your device using the DriverWizard of the new WinDriver version and gradually merge all device-specific code from your legacy code into the generated skeletal code.

This is the recommended method as it allows you to make sure that the driver still works and compiles correctly step by step, and stop and debug at any moment that behavior is not as desired (As opposed to trying to compile an old code against the new WinDriver version, where it may be harder to just get it to compile). Most of the steps in this chapter will happen by themselves/be skipped if you choose to follow the first method.

7.4.2. Checklist for Driver Code upgrade

If you must not regenerate code with the wizard as described in the previous section, the following is a list of action items to verify when upgrading a WinDriver based code.

7.4.2.1 Register Your New License

If you are using a registered version of WinDriver, contact Jungo Connectivity at wd_license@jungo.com to acquire a WinDriver license registration string for the new version. Then register the new license from DriverWizard File/Register WinDriver and from your code.

** Note**

If you have a valid license subscription you are entitled to receive a new license free of charge. In case you do not have such a plan, contact sales@jungo.com to request a temporary license that will allow you to evaluate the new version.

- Modify the driver code to register your new license - i.e., replace the license string in the call to [WDU_Init\(\)](#) ([USB](#) / [WDC_DriverOpen\(\)](#) ([PCI/ISA](#) - [WDC API](#)) / [WD_License\(\)](#) (low-level API) from your code.
- PCI users - if you created a Kernel PlugIn driver, make sure to also update the license string in your Kernel PlugIn code.

7.4.2.2 Conform to API Updates

Some functions and APIs have changed in the course of the years, make sure your WDAPI function calls are correctly formulated.

Linking your projects with the high-level WinDriver-API DLL / shared object - `wdapi<version>` (version 8.x+) / `wd_utils` (version 7.x) - frees you of the need to include the source files from the `WinDriver/src/wdapi` directory (version 8.x+) / `WinDriver/src` directory (version 7.x) in your project.

In version 8.0.0 the name of the DLL/shared object module was changed from `wd_utils` to `wdapi` (`version`) (e.g. `wdapi1440` in version 14.4.0) as part of the addition of versioning support to this module. This enables you to upgrade your driver, including the DLL/shared object, without worrying about the possible effects on other drivers, developed with earlier versions of WinDriver, which may be using the same module.

On Windows, in version 8.x and newer, you can use the WDAPI DLL - `wdapi<version>.dll` (found in the `WinDriver\redist` directory) by linking your project with the `WinDriver\lib\<CPU>\wdapi<version>.lib` library (e.g. `WinDriver\lib\x86\wdapi1511.lib`) - for MS Visual Studio(Visual C++) projects. Similarly, the name of the WinDriver .NET API DLL changed in version 8.0.0 from `wdapi_dotnet.dll` to

wdapi<version>_dotnet.dll (e.g. wdapi1511_dotnet.dll), and the DLL was moved to the WinDriver\lib\<CPU>\<.NET version>\directory (e.g., WinDriver\lib\x86\4\).

On Unix based OSes, such as Linux, MacOS and Solaris (Solaris was supported until 9.0.1), in version 8.x and newer you can use libwdapi<version>.so by linking your driver project with WinDriver/lib/libwdapi<version>.so (e.g. libwdapi800.so in WinDriver version 8.0.0). To link your Linux project with this shared object, add wdapi<version> to the makefile's link flag (LFLAGS += -l wdapi<version>; e.g., LFLAGS += -l wdapi800), instead of listing all the source files from the WinDriver/src/wdapi directory (previously WinDriver/src/ - see below) in the makefile (under the SRCS flag). On all platforms, the sample and generated DriverWizard projects demonstrate how to correctly link the project with the relevant DLL/shared object for your WinDriver version and target OS.

** Attention**

If your code uses the high-level WinDriver-API DLL / shared object, you will need to distribute wdapi<version>.dll (version 8.x+) / wd_utils.dll (version 7.x) - for Windows, or wdapi<version>.so (version 8.x+) / libwd_utils.so (version 7.x) - for Linux and Solaris, with your driver. Windows .NET users should also distribute wdapi<version>_dotnet.dll.

WinDriver source files location changes

In version 8.0.0 the WinDriver C source files were moved from the WinDriver/src directory to the WinDriver/src/wdapi/ directory. The .NET source files were moved from the WinDriver/wdapi.net/ directory to the WinDriver/src/wdapi.net/ directory.

If you have selected to upgrade your version 6.2.x project to use the wdapi (version 8.x+) DLL/shared object, this should not normally affect you. However, if your project directly includes WinDriver source files, you may need to modify your project/make file to point to the new source files location.

Update your project's files search paths

Beginning with version 7.0.1, the include path in the WinDriver project/make files contains the path to the WinDriver/ and WinDriver/include/ directories, and the #include statements in the WinDriver source files and generated DriverWizard code were consequently modified to indicate only the name of the header file to include, instead of the full/relative path to the file (as done in earlier versions). In light of these changes, when rebuilding a driver project from version 7.0.0 or earlier of WinDriver with the source files from version 7.0.1 or newer, you may need to modify your project/make file and add the path to the WinDriver/ and WinDriver/include/ directories to the project's include path in order to successfully build the project.

For USB, beginning with version 7.0.0 of WinDriver, if you have created a console driver application/DLL/shared object that calls functions implemented in WinDriver/samples/c/shared/usb_diag_lib.c (as is the case for the sample and generated WinDriver USB diagnostic driver projects), to build your project with the usb_diag_lib.c file from the new version you must add the new WinDriver/samples/c/shared/diag_lib.c file to your project. For PCI/ISA users, beginning with version 7.0.0 WinDriver features the high-level WDC library, which provides convenient wrapper APIs to the standard WinDriver PCI/ISA APIs. (This library is part of the wdapi<version> (v8.x+) / shared object (see above; the source files are found under the WinDriver/src/wdapi directory (v8.x+). The WDC APIs are documented in this manual. The generated DriverWizard v7.x+ projects use the WDC APIs instead of the low-level WD_xxx APIs. The WDC APIs are also used from the v7.x+ pci_diag, pci_dump and PLX samples. Since WDC mainly provides wrappers to the standard WinDriver APIs, which are still supported, you do not need to modify your old code to use the new WDC library. Should you select to upgrade your code to use the WDC APIs, you can examine the new samples and generated code and compare them to those from your old WinDriver version for a better understanding of how to use the new APIs.

** Attention**

To use the WDC APIs you will need to either include the relevant wdc_xxx.c source files from the WinDriver/src/wdapi directory in your project/makefile; or link your project with the wdapi<version> WinDriver high-level API DLL/shared object.

7.4.2.3 64-bit OS upgrade (Windows and Linux)

When porting your driver from a 32-bit platform to a 64-bit platform, your project or makefile must include the KERNEL_64BIT preprocessor definition. In the makefiles, the definition is added using the -D flag: -DKERNEL↔_64BIT. The sample and wizard-generated Linux and Windows GCC makefiles and Windows MS Visual Studio projects in the 64-bit WinDriver toolkit already add this definition.

7.4.2.4 Rename your driver (Windows and Linux)

To avoid conflicts with other WinDriver-based drivers on the target platforms, we highly recommend that you rename the default WinDriver driver module - windrvr(VERSION).sys (e.g., windrvr1511.sys) on Windows / windrvr(VERSION).o/.ko (e.g., windrvr1511.o/.ko) on Linux (or windrvr6.sys / windrvr6↔o/.ko in versions 11.8.0 and older) - to a unique name, by following the instructions in ([Distributing Your Driver](#)). The Linux USB GPL driver - windrvr(VERSION)_usb.o/.ko (or windrvr6_usb.o/.ko in version 11↔8.0 and older) is automatically renamed when renaming the main WinDriver Linux driver. When creating a PCI Kernel PlugIn driver, select a unique name as well.

7.4.2.5 Ensure that your code uses the correct driver module

Verify that the call to [WD_DriverName\(\)](#) in your driver code (if exists) uses the new driver-module name - windrvr(VERSION) or your renamed version of this driver. In version 11.9.0 of WinDriver the default Win↔Driver driver-module name changed from windrvr6 to windrvr(VERSION) (e.g., windrvr1511). Consequently, when using the default driver-module name old projects need to be updated to use the default name from the newer version. If you use the generated DriverWizard code or one of the samples from the new WinDriver version, the code will already use the default driver name from the new version. Also, if your code is based on generated/sample code from an earlier version of WinDriver, rebuilding the code with [windrvr.h](#) from the new version is sufficient to update the code to use the new default driver-module name (due to the use of the WD↔_DEFAULT_DRIVER_NAME_BASE definition). If you elect to rename the WinDriver driver module, ensure that your code calls [WD_DriverName\(\)](#) with your custom driver name. If you rename the driver from the new version to a name already used in your old project, you do not need to modify your code. To apply a driver name change - whether using the default driver name or a custom name - your user-mode driver project must be built with the WD_DRIVER_NAME_CHANGE preprocessor flag (e.g., -DWD_DRIVER_NAME_CHANGE).

7.4.2.6 Rebuild your updated driver

Rebuild your updated driver project with the source files **from the new version**.

PCI users who created a Kernel PlugIn driver **must rebuild it with the files from the new version** as well.

7.4.2.7 Upgrade Your Device INF File (Windows)

On Windows, you must create and install a new INF file for your device, which registers it with the driver module from the new version windrvr(VERSION).sys (e.g., windrvr1511.sys) / windrvr6.sys in version 11.8.0 and older - or your renamed version of this driver (in version 9.x and newer). You can use DriverWizard from the new version to generate the new INF file, or change the driver version in your old INF file.

7.4.2.8 Digitally Sign Your Driver Files (Windows)

Microsoft requires that kernel drivers to be digitally signed. Therefore, if you use any of the following driver files you must digitally sign them. A renamed version of the WinDriver kernel driver (the default Win↔Driver driver - windrvr(VERSION).sys / windrvr6.sys in version 11.8.0 and older - is already digitally signed), a Plug-and-Play device INF file, and/or a PCI Kernel PlugIn driver. For more info see [17.3. Windows Digital Driver Signing and Certification](#).

** Note**

If you have any questions in regard to driver digital signature, please send an e-mail to WinDriver@jungo.com. Our team will answer any questions you may have and provide full assistance.

7.4.2.9 Upgrade Your Driver Distribution/Installation Package

Create a new driver installation package that contains the relevant files from the new WinDriver distribution, depending on your hardware and target OS. Hardware and OS-specific driver distribution instructions can be found in [Distributing Your Driver](#).

** Attention**

Do not try to use WinDriver related files (shared libraries, etc.) from older versions in your new distribution package as they will not work with the newer version.

7.4.2.10 Check for changes in variable and struct sizes

Upon switching platforms, operating systems or environments, the default sizes of basic types such as `int`, `long` etc. may change. If you are changing a platform, make sure that these changes do not affect your code by careful debugging. When upgrading code from WinDriver versions older than 14.9, make sure to address the changes made to the `DWORD` typedef, which is widely used in WinDriver's sample and generated code. For more information, see [7.5.2 64-Bit and 32-Bit Data Types](#).

7.5. 32-Bit Applications on 64-Bit Windows and Linux Platforms

By default, applications created using the 64-bit versions of WinDriver are 64-bit applications, and application created using the 32-bit versions of WinDriver are 32-bit applications. However, you can also use the 64-bit WinDriver versions to create 32-bit applications that will run on the supported Windows and Linux 64-bit platforms.

** Attention**

WinDriver for MacOS currently supports only 64-bit applications.

We recommend avoiding compiling your applications in this manner unless you have to, as this might introduce a performance decrease for your driver.

In the following documentation, (WD64) signifies the path to a 64-bit WinDriver installation directory for your target operating system, and (WD32) signifies the path to a 32-bit WinDriver installation directory for the same operating system.

To create a 32-bit application for 64-bit Windows or Linux platforms, using the 64-bit version of WinDriver, do the following:

- Create a WinDriver application, as outlined in [Using DriverWizard](#) (e.g., by generating code with DriverWizard, or using one of the WinDriver samples).
- Build the application with an appropriate 32-bit compiler for your target OS, using the following configuration:
 - Make sure that `KERNEL_64BIT` preprocessor definition is added to your project or makefile. In the makefiles, the definition is added using the `-D` flag `-DKERNEL_64BIT`. The sample and wizard-generated Linux and Windows GCC makefiles and the Windows MS Visual Studio projects, in the 64-bit WinDriver toolkit, already include this definition.
 - Link the application with the specific version of the WinDriver-API library/shared object for 32-bit applications executed on 64-bit platforms - (WD64) \lib\amd64\x86\wdapi1511_32.lib on Windows / (WD64) /lib/libwdapi1511_32.so on Linux.

The sample and wizard-generated project and make files for 32-bit applications in the 64-bit WinDriver toolkit already link to the correct library:

On Windows, the MS Visual Studio project files, CMake or Windows GCC makefiles are defined to link with `(WD64)\lib\amd64\x86\wdapi1511.lib`. On Linux, the installation of the 64-bit WinDriver toolkit on the development machine creates a `libwdapi1511.so` symbolic link in the `/usr/lib` directory - which links to `(WD64)/lib/libwdapi1511_32.so` - and in the `/usr/lib64` directory - which links to `(WD64)/lib/libwdapi1511.so` (the 64-bit version of this shared object).

The sample and wizard-generated WinDriver makefiles rely on these symbolic links to link with the appropriate shared object, depending on whether the code is compiled using a 32-bit or 64-bit compiler.

When distributing your application to target 64-bit platforms, you need to provide with it the WinDriver-API DLL/shared object for 32-bit applications executed on 64-bit platforms - `(WD64)\redist\wdapi1511_32.dll` on Windows / `(WD64)/lib/libwdapi1511_32.so` on Linux. The installation on the target should copy the renamed DLL/shared object to the relevant OS directory - `\windir%\sysWOW64` on Windows or `/usr/lib` on Linux. All other distribution files are the same as for any other 64-bit WinDriver driver distribution.

An application created using the method described in this section will not work on 32-bit platforms. A WinDriver application for 32-bit platforms needs to be compiled without the `KERNEL_64BIT` definition; it needs to be linked with the standard 32-bit version of the WinDriver-API library/shared object from the 32-bit WinDriver installation(`(WD32)\lib\x86\wdapi1511.lib` on Windows / `(WD32)/lib/libwdapi1511.so` on Linux); and it should be distributed with the standard 32-bit WinDriver-API DLL/shared object (`(WD32)\redist\wdapi1511.dll` on Windows / `(WD32)/lib/libwdapi1511.so` on Linux) and any other required 32-bit distribution file.

7.5.1. Developing a 32-Bit Application for Both 32-Bit and 64-Bit Platforms

If you have both 64-bit and 32-bit WinDriver installations, you can also create a single 32-bit application that can be executed on both 32-bit and 64-bit platforms. This can be done using the following method:

- Create a DLL (on Windows) or a shared object (on Linux) that concentrates the calls to the WinDriver APIs. If you created a WinDriver application using the generated DriverWizard code or one of the WinDriver samples, convert this application to a DLL/shared object.

** Attention**

When using DriverWizard to generate PCI driver code for Windows MS Visual Studio, you have the option to generate the library code as a DLL (for both 32-bit and 64-bit environments).

- Compile two versions of your DLL/shared object:

A version for 32-bit platforms: This version should be compiled using a 32-bit compiler, without the `KERNEL_64BIT` definition, and linked with the standard 32-bit WinDriver-API library/shared object from the 32-bit WinDriver installation - `<WD32>\lib\x86\wdapi<ver>.lib` on Windows (e.g.,`C:\WinDriver\lib\x86\wdapi1200.lib`) /`<WD32>/lib/libwdapi<ver>.so` on Linux (e.g.,`~/WinDriver/lib/libwdapi1200.so`). A version for 64-bit platforms: This version should be compiled using a 32-bit compiler, with the `KERNEL_64BIT` definition, and linked with the standard 32-bit WinDriver-API library/shared object from the 64-bit WinDriver installation - `<WD64>\lib\amd64\x86\wdapi<ver>_32.lib` on Windows (e.g., `C:\WinDriver64\lib\x86\wdapi1200.lib`)/`<WD64>/lib/libwdapi<ver>_32.so` on Linux (e.g., `~/WinDriver64/lib/libwdapi1200_32.so`) (see details above regarding compilation of 32-bit applications for 64-bit platforms).

- Write a 32-bit application that communicates with WinDriver via the DLL/shared object that you created: the application should be implemented to load the relevant version of the DLL/shared object - 32-bit or 64-bit - depending on the platform on which it is run.

** Attention**

This application should be compiled using an appropriate 32-bit compiler, without the `KERNEL_64BIT` definition.

When distributing a driver that was developed using this method, be sure to distribute the relevant files for each target platform:

For 32-bit platforms, distribute the application together with the 32-bit version of your WinDriver-wrapper DLL/shared object, and with the standard 32-bit files from the 32-bitWinDriver installation, including the 32-bit WinDriver DLL/shared object

(<WD32>\redist\wdapi1511.dll on Windows / <WD32>/lib/libwdapi1511.so on Linux).

For 64-bit platforms, distribute the application together with the 64-bit version of your WinDriver-wrapper DLL/shared object, and with the standard 64-bit files from the 64-bitWinDriver installation, including the 64-bit WinDriver DLL/shared object

(<WD64>\redist\wdapi1511.dll on Windows / <WD64>/lib/libwdapi1511.so on Linux).

7.5.2 64-Bit and 32-Bit Data Types

** Attention**

Since WinDriver 15.00 The DWORD typedef is **ALWAYS** 32 bits wide.

On all previous versions the previous table has applied:

Operating System	DWORD width
Windows (32 and 64 bits)	4 bytes
Linux (32 bits)	4 bytes
Linux (64 bits)	8 bytes
MacOS (64 bits)	8 bytes

When upgrading existing code from a version lower than 15.00, make sure that your code does not use DWORD variables for holding addresses or hardware related data.

- If a variable used to be 64 bits wide in older versions and you need to keep it 64 bit wide, define it as `UINT64` instead.
- If a variable is used to hold data which is related to a WinDriver API of size `DWORD` (i.e. `dwOptions`), you could keep it defined as `DWORD`.

7.6. WinDriver .NET APIs in PowerShell

Microsoft defines PowerShell as “a task-based command-line shell and scripting language built on .NET. PowerShell helps system administrators and power-users rapidly automate tasks that manage operating systems and processes.”

The advantages of using WinDriver with PowerShell are:

- PowerShell comes built-in in all modern Windows systems. No need to install any additional software to use it.
- No need to compile your code as it is a scripting language.
- The ability to easily combine WinDriver with your scripts or with any other .NET based library.

WinDriver has been providing its .NET wrapper DLL `wdapi_dotnet1511.dll` (1511 stands for the version number) for many years. Using PowerShell's built-in .NET support you can easily open this DLL and call API functions by using the following code from PowerShell's command line:

- Load the WinDriver .NET DLL using namespace `Jungo.wdapi_dotnet`:

```
$wdapi_dotnet = [Reflection.Assembly]::LoadFile($Env:WD_BASEDIR + "\lib\amd64\wdapi_netcore1511.dll")
```

- Open WinDriver:

```
[windrvr_decl]:::WD_DriverName("YOUR_DRIVER_NAME")
[wdc_lib_decl]:::WDC_DriverOpen([wdc_lib_consts]:::WDC_DRV_OPEN_DEFAULT, "YOUR_LICENSE_STRING")
```

- Open a device - by default the slot will be set to 0,0,0,0:

```
$slot = New-Object -TypeName "Jungo.wdapi_dotnet.WD_PCI_SLOT"
$DeviceInfo = New-Object -TypeName "Jungo.wdapi_dotnet.WD_PCI_CARD_INFO"
$DeviceInfo.pciSlot = $slot
[wdc_lib_decl]:::WDC_PciGetDeviceInfo($DeviceInfo)
```

- Device context will be set to NULL for simplicity of sample:

```
$pDevCtx = $NULL
$dev = New-Object -TypeName "Jungo.wdapi_dotnet.WDC_DEVICE"
[wdc_lib_decl]:::WDC_PciDeviceOpen([ref]$dev, $pDeviceInfo, $pDevCtx)
```

- Read from the device (or any other WinDriver API usage):

```
[wdc_lib_decl]:::WDC_ReadAddr32($dev.hDev, 0, 0x60, 0x1)
```

- Cleanup:

```
[wdc_lib_decl]:::WDC_PciDeviceClose($dev.hDev)
[wdc_lib_decl]:::WDC_DriverClose()
```

There is a complete PowerShell Script utilizing WinDriver's API. Jungo provides a sample for PowerShell based on our pci_diag sample application. It is located in WinDriver\samples\powershell\pci_diag. Make sure you allow Powershell to run scripts, then do as follows:

- Open PowerShell as an administrator and run:

```
Set-ExecutionPolicy RemoteSigned

cd $Env:WD_BASEDIR\samples\powershell\pci_diag./pci_diag

** Attention**
```

Currently the sample does not yet support the following features: Power events, IPC, Kernel PlugIn, USB, Shared Kernel Buffer. WinDriver's API and pci_diag work on both the PowerShell command line and the PowerShell ISE IDE. Starting with WinDriver 14.70, WinDriver's .NET 5 API can be used in on Linux or MacOSX as well, using PowerShell 7 or higher.

7.7. WinDriver Server API

The following WinDriver Server APIs are not part of the standard WinDriver API set. Also, they are not included in the standard version of WinDriver.

However, they are part of “WinDriver for Server” API and require “WinDriver for Server” separate license. “WinDriver for Server” APIs are included in WinDriver evaluation version for **evaluation only**.

IPC API (Inter-process communication):

- [WDS_IpcRegister\(\)](#)
- [WDS_IpcUnRegister\(\)](#)
- [WDS_IsIpcRegistered\(\)](#)
- [WDS_IpcScanProcs\(\)](#)
- [WDS_IpcMulticast\(\)](#)

- `WDS_IpcSubGroupMulticast()`
- `WDS_IpcUidUnicast()`

Buffer sharing API:

- `WDC_DMABufGet()`
- `WDS_SharedBufferAlloc()` (when using `KER_BUF_ALLOC_NON_CONTIG` flag)
- `WDS_SharedBufferGet()` (IPC license)

PCI devices scanning:

- `WDC_PciScanDevices()` (for more than 100 devices)
- `WDC_PciScanDevicesByTopology()` (for more than 100 devices)
- `WDC_PciScanRegisteredDevices()` (for more than 100 devices)

SR-IOV API(Linux Only):

- `WDC_PciSriovEnable()`
- `WDC_PciSriovDisable()`
- `WDC_PciSriovGetNumVFs()`

7.8. FAQ

7.8.1. Using WinDriver to build a GUI Application

Although most samples provided in the WinDriver package are console-mode applications, it is possible to integrate WinDriver's API into Graphic User Interface (GUI) applications. Some examples of GUI applications that were built with WinDriver are our DriverWizard and Debug Monitor (built with the Qt Framework), and also our .NET GUI samples/generated codes.

7.8.2. Can WinDriver handle multiple devices, of different or similar types, at the same time?

It is possible. For the sake of simplicity, our C, Java and Python PCI samples only allow opening one device at a time, but it is indeed possible in these languages as well. Our USB samples for C, Java and Python allow opening multiple devices at the same time. Our .NET samples also allow opening multiple devices at the same time.

Generally, each device controlled by WinDriver has its own unique handle, allowing applications to open multiple handles at the same time if needed.

7.8.3. Can I run two different device drivers, both developed with WinDriver, on the same machine?

Yes. You can run several WinDriver-based applications simultaneously on the same machine. It is strongly recommended that the WinDriver Kernel Module will be renamed in that case.

7.8.4. Can WinDriver group I/O and memory transfers?

Yes. Using WinDriver, you can group I/O and memory transfers by calling [WD_MultiTransfer\(\)](#), which can perform multiple transfers in one call. However, Jungo generally recommends using the High-Level API (WDC_xxx) for I/O.

Note that you can also access the memory directly from your user-mode application using the virtual user-mode mapping of the physical address, which is returned by or [WDC_PciDeviceOpen\(\)](#) in (((PWDC_DEVICE) h←Dev) → cardReg.Card.Item[i].I.Mem.pUserDirectAddr or [WD_CardRegister\(\)](#) in cardReg.←Card.Item[i].I.Mem.pUserDirectAddr, where i is the index item of the relevant memory item in the [WD_ITEMS](#) Item array.

7.8.5. I need to define more than 20 "hardware items" (I/O, memory, and interrupts) for my ISA card. Therefore, I increased the value of [WD_CARD_ITEMS](#) in the [windrvr.h](#) header file (due to the definition of the Item member of the [WD_CARD](#) structure as an array of [WD_CARD_ITEMS](#) [WD_ITEMS](#) structures). But now [WD_CardRegister\(\)](#) will not work. Why?

If you need to define more than [WD_CARD_ITEMS](#) items for your card (currently 128 items, according to the definition of [WD_CARD_ITEMS](#) in [windrvr.h](#)), do not modify the value of [WD_CARD_ITEMS](#) in the code, but instead, simply call [WD_CardRegister\(\)](#) several times from your code, with different items each time. It is not mandatory to lock all the resources on a specific card with a single [WD_CardRegister\(\)](#) call. Alternatively, consider grouping several memory/I/O address ranges into a single BAR definition, so that the overall resources item count does not exceed the default 20 items limit.

We highly recommend against changing anything in [windrvr.h](#). The effect will certainly not be what you expect and it could be potentially disastrous.

7.8.6. I have a WinDriver based application running on a certain operating system, how do I port my code to a different operating system?

WinDriver generally rids you of the need to do any changes in your program's code. Of course some OS specific functions that you have been using that are not from the WinDriver API might not work anymore (i.e. Win32 functions that are not provided by POSIX, or the other way around), but this is why we recommend trying to stick only to cross-platform standard functions in your C code and to the WinDriver API. There are about 10% of WinDriver's API features that are not cross platform, but usually this could be overcome with different alternatives. If this could not be overcome, contact Jungo Support for assistance.

The recommended and easiest way to shift between platforms is to generate a `CMakeLists.txt` file for your project using the DriverWizard. This will allow you to port the code over all supported OSes and makes the porting a much easier task, assuming you've installed CMake on your target development machine.

On supported languages other than C, porting should be even easier as these languages support cross-platform use inherently.

Assuming you're using a renamed driver - a step you will have to go through is creating a new project from the DriverWizard in your new operating system. This will create a renamed driver for your target platform (i.e. if you had a `.sys` file for your driver on a Windows system, then on Linux the wizard would create a `.ko` file, etc.).

As described in [7.4.1. Regenerating code and gradually merging](#), the best way for this kind of porting is first to generate new code with the DriverWizard, make sure that it works without any modifications, and then gradually merge your code from the other operating system into your new OS code.

Chapter 8

Debugging Drivers

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

8.1. User-Mode Debugging

Since WinDriver is accessed from the user mode, we recommend that you first debug your code using your standard debugging software, such as the compiler's debugger.

Debug Monitor is a powerful graphical- and console-mode tool for monitoring all activities handled by the WinDriver kernel. You can use this tool to monitor how each command sent to the kernel is executed.

In addition, WinDriver enables you to print your own debug messages to the Debug Monitor, using the [WD_DebugAdd\(\)](#) function or the high-level [PrintDbgMessage\(\)](#) function. Note that this function can be called from both user-mode and kernel-mode code (created using WinDriver's Kernel PlugIn feature) and can also be called from within functions running at HIGH IRQL, such as the Kernel PlugIn [KP_IntAtlrql\(\)](#) function.

The Debug Monitor comes in two versions:

- `wddebug_gui` - a GUI version
- `wddebug` - a console-mode version

Note that `wddebug_gui` is available for all supported Desktop operating system but is currently not available for Linux ARM/ARM64. For these platforms, use the console-mode `wddebug`.

Both Debug Monitor versions are provided in the `WinDriver/util` directory.

Please note that [WD_DebugAdd\(\)](#) function can be called from both user-mode and kernel-mode code (created using WinDriver's Kernel PlugIn feature) and can also be called from within functions running at HIGH IRQL, such as the Kernel PlugIn [KP_IntAtlrql\(\)](#) function.

You can also select to send the debug messages to a kernel debugger, instead of to the Debug Monitor log, as we explain further. This can enable you, for example, to log the debug message on another PC in case the system crashes (BSOD) or hangs on the development PC.

Most of the WinDriver functions have a return value to indicate success or a relevant error code, which can assist you in the debugging process. The return status codes are defined in the `[WD_ERROR_CODES]` ([WD_ERROR_CODES](#)) enum.

When developing code in the kernel using the Kernel PlugIn, you can also use any kernel debugger in the debugging process (e.g., WinDbg - which is distributed with the Windows Driver Kit (WDK) and is part of the Debugging Tools for Windows package, distributed via the Microsoft website). As mentioned above, you can also select to direct the debug message from the Debug Monitor to a kernel debugger of your choice.

8.2. Debug Monitor

The Debug Monitor utility logs debug messages from WinDriver's kernel-mode and usermode APIs. You can also use WinDriver APIs to send your own debug messages to the Debug Monitor log.

When using WinDriver's API (such as [WD_Transfer\(\)](#)), to read/write memory ranges on the card in the kernel, while the Debug Monitor is activated, WinDriver's kernel module validates the memory ranges, i.e., it verifies that the reading/writing from/to the memory is in the range that is defined for the card.

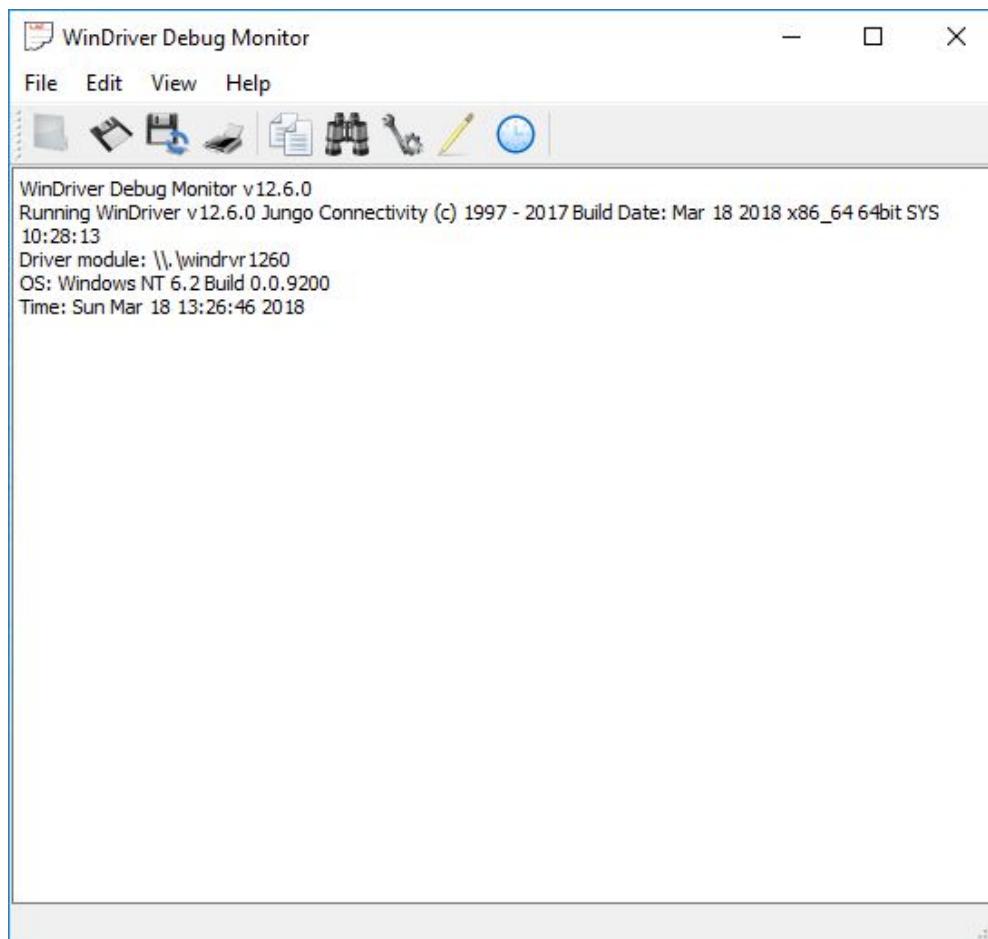
Use DriverWizard to check values of memory, registers and configuration in the debugging process.

8.2.1. The wddebug_gui Utility

wddebug_gui is a fully graphical (GUI) version of the Debug Monitor utility for Windows and Linux.

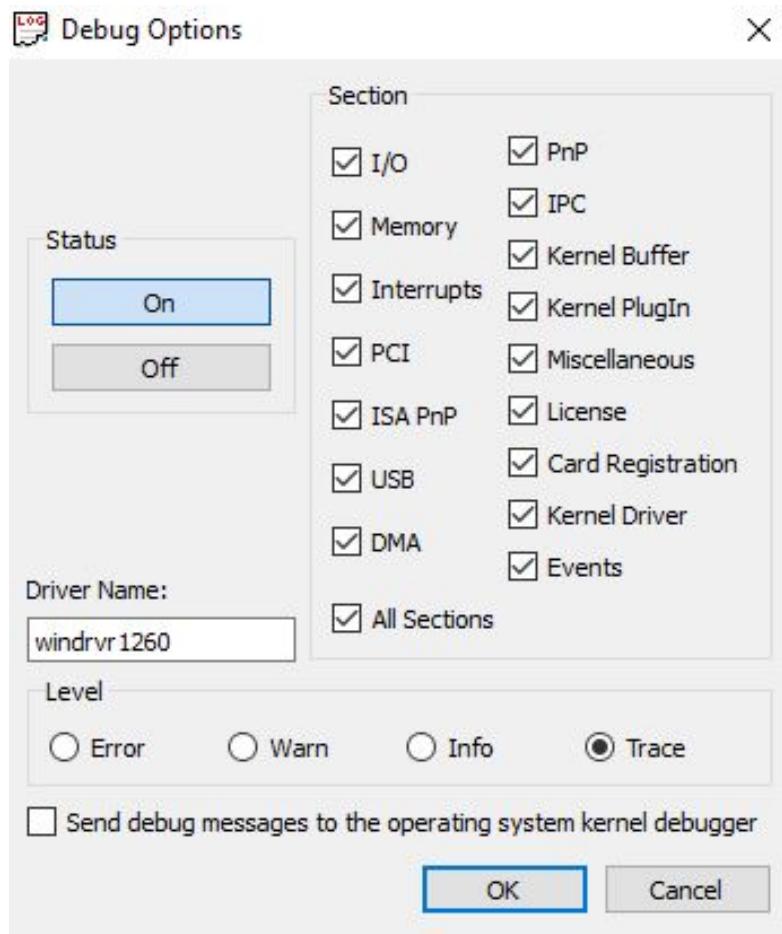
- Run the Debug Monitor using either of the following methods:
 - Run WinDriver/util/wddebug_gui.
 - Run the Debug Monitor from DriverWizard's Tools menu.
 - On Windows, choose **Start | Programs | WinDriver | Debug Monitor**.

Start Debug Monitor



- Set the Debug Monitor's status, trace level and debug sections information from the **Debug Options** dialogue, which is activated either from the Debug Monitor's **View | Debug Options** menu or the **Debug Options** toolbar button.

Debug Options



Status - Set trace on or off. **Section** - Choose what part of the WinDriver API you would like to monitor.

For example, if you are experiencing problems with the interrupt handler on your PCI card, select the PCI and Interrupts sections. USB developers should select the USB section.

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 trace level, resulting in minimum output to the screen. Trace is the highest trace level, displaying every operation the WinDriver kernel performs. **Send debug messages to the operating system kernel debugger** - Select this option to send the debug messages received from the WinDriver kernel module to an external kernel debugger, in addition to the Debug Monitor.

On Windows, the first time that you enable this option you will need to restart the PC. A free Windows kernel debugger, WinDbg, is distributed with the Windows Driver Kit (WDK) and is part of the Debugging Tools for Windows package, distributed via the Microsoft web site. **Driver Name** - This field displays the name of the driver currently being debugged. Changing it allows you to debug a renamed WinDriver driver.

- Once you have defined what you want to trace and on what level, click **OK** to close the **Debug Options** window.
- Optionally make additional configurations via the Debug Monitor menus and toolbar.

When debugging OS crashes or hangs, it's useful to auto-save the Debug Monitor log, via the **File | Toggle Auto-Save** menu option (available also via a toolbar icon), in addition to sending the debug messages to the OS kernel debugger.

- Run your application (step-by-step or in one run).

You can use the **Edit | Add Custom Message...** menu option (available also via a toolbar icon) to add custom messages to the log. This is especially useful for clearly marking different execution sections in the log.

- Watch the Debug Monitor log (or the kernel debugger log, if enabled) for errors or any unexpected messages.

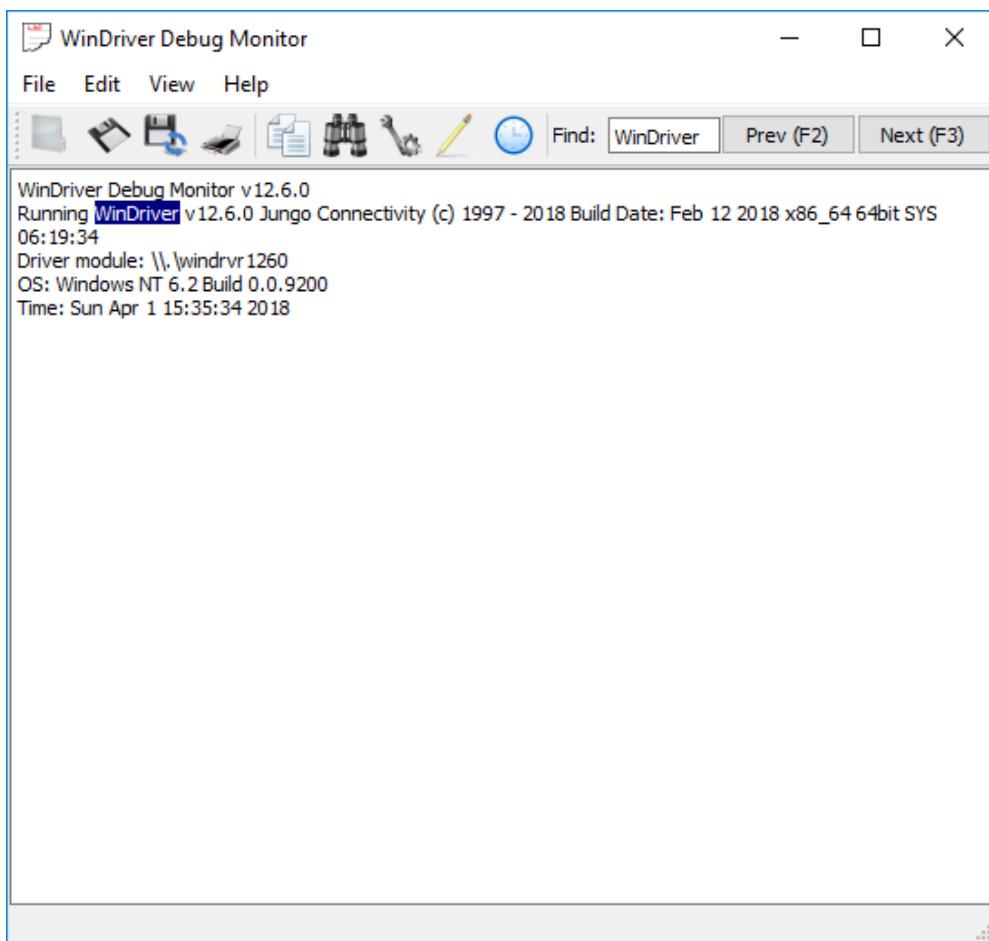
8.2.1.1. Search in wddebug_gui

wddebug_gui allows to find expressions in the Debug output in 4 ways:

- Starting to type an expression with the keyboard immediately searches for it in the Debug output.
- Pressing CTRL+F displays the Find field in the toolbar.
- Clicking the Find icon in the toolbar.
- Going to **Edit | Find...**

If an expression is found it is marked. It is possible to browse between occurrences of an expression using **Prev (F2)** and **Next (F3)**.

Search in wddebug_gui



8.2.1.2. Opening Windows kernel crash dump with wddebug_gui

In driver development, it is not uncommon to experience kernel crashes that lead to the Blue Screen of Death (BSOD). In order to assist developers in debugging and solving these crashes, WinDriver allows saving its kernel debug messages in the memory dump that Windows automatically creates when the system crashes. These crash dumps can later be opened in wddebug_gui after the system was restarted.

To direct the debug messages to a kernel debugger, follow these steps:

- Open your selected kernel debugger.

- Run the Debug Monitor and select to direct the WinDriver debug messages to a kernel debugger, using either of the following methods:

- Open the GUI version of the Debug Monitor - `wddebug_gui` (available on Windows, Linux, and Mac OS), check the “Send debug messages to the operating system kernel debugger” option in the Debug Options window, and press ‘OK’.
- Run the console-mode version of the Debug Monitor - `wddebug` (available on all the supported operating systems) - with the `dbg_on` command:

```
WDDEBUG [<driver_name>] dbg_on [<level>] \
[<sections>]
```

- Load up the WinDriver symbols by going to **File | Symbol File Path...** and loading your driver's symbols MyDriver.pdb (the file name for the default driver is `lib/windrivr1511.pdb`). Do not skip this step, otherwise the kernel dump might not be comprehensible.
- Load the kernel crash dump (.dmp) file by going to **File | Process Kernel Dump...**. `wddebug_gui` will display the output from the crash dump.

Windows users can use Microsoft’s WinDbg tool, for example, this tool is distributed with the Windows Driver Kit (WDK) and is part of the Debugging Tools for Windows package, distributed via the Microsoft web site.

8.2.1.3. Running `wddebug_gui` for a Renamed Driver

By default, `wddebug_gui` logs messages from the default WinDriver kernel module - `windrivr1511.sys/.dll/.o/.ko`. However, you can also use `wddebug_gui` to log debug messages from a renamed version of this driver see [Driver Installation - Advanced Issues](#), in two ways:

- From within `wddebug_gui`, by going to **View | Debug Options** and changing the value of the "Driver name" field to your own renamed driver's name.
- By running `wddebug_gui` from the command line with the `driver_name` argument:

```
'wddebug_gui <driver_name>'
```

The driver name should be set to the name of the driver file without the file's extension, e.g., `windrivr1511`, not `windrivr1511.sys` (on Windows) or `windrivr1511.o` (on Linux). For example, if you have renamed the default `windrivr1511.sys` driver on Windows to `my_driver.sys`, you can log messages from your driver by running the Debug Monitor using the following command:

```
'wddebug_gui my_driver'
```

8.2.2. The `wddebug` Utility

8.2.2.1. Console-Mode `wddebug` Execution

The `wddebug` version of the Debug Monitor utility can be executed as a console-mode application on all supported operating systems: Windows, and Linux. To use the console-mode Debug Monitor version, run `WinDriver/util/wddebug` in the manner explained below.

wddebug console-mode usage

```
wddebug [<driver_name>] [<command>] [<level>] [<sections>]
```

The `wddebug` arguments must be provided in the order in which they appear in the usage statement above.

- <driver_name> - The name of the driver to which to apply the command.

The driver name should be set to the name of the WinDriver kernel module - `windrivr1511` (default), or a renamed version of this driver (refer to the explanation in [17.2. Renaming the WinDriver Kernel Driver](#)). We remind that the driver name should be set to the name of the driver file without the file's extension, e.g., `windrivr1511`, not `windrivr1511.sys` (on Windows) or `windrivr1511.o` (on Linux).

- <command>- The Debug Monitor command to execute.

Activation commands:

Command	Description
on	Turn the Debug Monitor on.
off	Turn the Debug Monitor off.
dbg_on	Redirect the debug messages from the Debug Monitor to a kernel debugger and turn the Debug Monitor on (if it was not already turned on). On Windows, the first time that you enable this option you may need to restart the PC. The on and dbg_on commands can be used together with the <level> and <sections> arguments.
dbg_off	Stop redirecting debug messages from the Debug Monitor to a kernel debugger.
dump	Continuously send ("dump") debug information to the command prompt, until the user selects to stop (by following the instructions displayed in the command prompt).
status	Display information regarding the running driver (<driver_name>), the current Debug Monitor status - including the active debug level and sections (when the Debug Monitor is on) - and the size of the debug-messages buffer.
clock_on	Add a timestamp to each debug message. The timestamps are relative to the driver-load time, or to the time of the last clock_reset command.
clock_off	Do not add timestamps to the debug messages.
clock_reset	Reset the debug-messages timestamps clock.
sect_info_on	Add section(s) information to each debug message.
sect_info_off	Do not add section(s) information to the debug messages.
help	Display usage instructions.
No arguments (including no commands)	This is equivalent to running 'wddebug help'.

The following arguments are applicable only with the on or dbg_on commands:

Argument	Description
<level>	The debug trace level to set - one of the following flags: ERROR, WARN, INFO, or TRACE (default). ERROR is the lowest trace level and TRACE is the highest level (displays all messages). When the <sections> argument is set, the <level> argument must be set as well (no default).
<sections>	The debug sections - i.e., the WinDriver API sections - to monitor.

This argument can be set either to ALL (default) - to monitor all the supported debug sections - or to a quoted string that contains a combination of any of the supported debug-section flags (run 'wddebug help' to see the full list).

Usage Sequence

To log messages using wddebug, use the following sequence:

- Turn on the Debug Monitor by running wddebug with either the on or dbg_on command; the latter redirects the debug messages to the OS kernel debugger before turning on the Debug Monitor.

You can use the <level> and <sections> arguments to set the debug level and sections for the log. If these arguments are not explicitly set, the default values will be used; (note that if you set the sections you must also set the level). You can also log messages from a renamed WinDriver driver by preceding the command with the name of the driver (default: windrvr1511) - see the <driver_name> argument.

- If you did not select to redirect the debug messages to the OS kernel debugger (using the dbg_on command), run wddebug with the dump command to begin dumping debug messages to the command prompt.

You can turn off the display of the debug messages, at any time, by following the instructions displayed in the command prompt.

- Run applications that use the driver, and view the debug messages as they are being logged to the command prompt/the kernel debugger.
- At any time while the Debug Monitor is running, you can run wddebug with the following commands:
 - status, clock_on, clock_off, clock_reset, sect_info_on, or sect_info_off.
 - on or dbg_on with different <level> and/or <sections> arguments.
 - dbg_on and dbg_off - to toggle the redirection of debug messages to the OS kernel debugger.
 - dump - to start a new dump of the debug log to the command prompt; (the dump can be stopped at any time by following the instructions in the prompt)
- When you are ready, turn off the Debug Monitor by running wddebug with the off command.

The status command can be used to view information regarding the running WinDriver driver even when the Debug Monitor is off.

Example

The following is an example of a typical wddebug usage sequence. Since no <driver_name> is set, the commands are applied to the default driver - windrvrl511.

Turn the Debug Monitor on with the highest trace level for all sections:

```
wddebug on TRACE ALL
```

This is the same as running wddebug on TRACE, because ALL is the default <sections> value. Then dump the debug messages continuously to the command prompt or a file, until the user selects to stop:

```
wddebug dump  
wddebug dump <filename>
```

Following that use the driver and view the debug messages in the command prompt. Turn the Debug Monitor off:
wddebug off

8.2.2.2. Debugging on a test machine

In order to debug your WinDriver application and driver a machine that does not have WinDriver installed on it:

- Copy WinDriver\util\wddebug.exe to your testing machine.
- Run wddebug dump from your target machine's command line.
- Run your application on a different window.
- Stop wddebug utility using CTRL+C.

8.3. FAQ

8.3.1. Should I use wddebug_gui or wddebug?

So why use wddebug_gui?

The GUI version offers easier search options and is more comfortable to use on modern desktop operating systems.

And why use wddebug?

- The GUI version depends on Qt, and runs slower than the console-version, and in some cases might make your application run slower when they run at the same time.
- If you wish to debug a WinDriver-based user application on a machine that does not have WinDriver installed on it (a redistribution machine)- it would be more complicated to run wddebug_gui on it because you will have to install or copy all Qt DLL/.so dependencies to that machine for the Debug Monitor to work. In contrast, wddebug only requires the wdapi shared library (wdapi1511.dll/.so).

- wddebug's source code is available at `WinDriver/samples/c/wddebug`. This allows you greater flexibility in modifying this application for your specific debugging needs. Jungo does not provide the source code for `wddebug_gui`.
- wddebug is easier to use in scripts, or remote console sessions such as PowerShell, SSH, etc.

8.3.2. Can I debug WinDriver-based code easily using MS Visual Studio (Visual C++)?

Yes! The code of the device driver you write runs in normal Win32 user mode. Therefore, you can compile and debug your code using MS Visual Studio. Generally, WinDriver-based user-code can be debugged on any IDE that allows compiling C/C++ applications in Windows.

8.3.3. How would you recommend debugging a WinDriver-based application?

Besides using `wddebug_gui` / `wddebug` as described in this chapter, we also recommend using the following tools when needed:

- Basic user-mode development: Use IDEs such as Visual Studio / Visual Studio Code / XCode (MacOS) and the debuggers that they offer. Other recommended tools are `valgrind` (Linux) / Dr. Memory (Windows) for locating memory leaks.
- Debugging into calls to the WDAPI shared library: You can use its source code which is available in `WinDriver/src/wdapi`.
- Kernel debugging (For Kernel-Plugin or identifying issues with WinDriver): In Windows you can use `WinDbg` / `WinDbg Preview`. More info on how to set these up is available on MSDN. In Linux/MacOS you can use `dmesg` to view kernel panics. MacOS also provides crash dumps for every application or kernel crash in a designated directory (Usually `~/Library/Logs`).

Chapter 9

Enhanced Support for Specific Chipsets

9.1. Enhanced Support for Specific Chipsets Overview

In addition to the standard WinDriver APIs and the DriverWizard code generation capabilities described in this manual, which support development of drivers for any PCI/ISA device, WinDriver features enhanced support for specific PCI chipsets. This enhanced support includes custom APIs, customized code generation (for some of the chipsets), and sample diagnostics code, which are all designed specifically for these chipsets.

WinDriver's enhanced support is currently available for the following PCI chipsets: PLX 6466, 9030, 9050, 9052, 9054, 9056, 9080 and 9656; Altera Qsys design; Xilinx BMD, XDMA, QDMA designs.

Customized code generation is available for the Altera Qsys design, Xilinx BMD, XDMA, QDMA designs chipsets.

9.2. Developing a Driver Using the Enhanced Chipset Support

When developing a driver for a device based on one of the enhanced-support chipsets described above, you can use WinDriver's chipset-set specific support in the following manner: if your device is based on the Altera Qsys, Avalon-MM designs, Xilinx BMD, XDMA, QDMA designs, you can generate customized code for the device by selecting this option in the DriverWizard code generation options dialogue (see [6.2. DriverWizard Walkthrough](#)). Alternatively, or if you are using another enhanced-support device, follow the steps below to use one the enhanced-support WinDriver samples as the starting point for your development:

- Locate the sample diagnostics program for your device under the `WinDriver/samples/language/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` subdirectory. The program's executable is found under a subdirectory for your target operating system (e.g., `WIN32` for Windows.)

- Run the custom diagnostics program to diagnose your device and familiarize yourself with the options provided by the sample program.
- 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/samples/language/chip_vendor/lib` directory.
- 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 [Improving PCI Performance](#).

9.3. The XDMA sample code

Starting from version 12.3, WinDriver supplies a user-mode sample code of a diagnostic utility that demonstrates several features of Xilinx PCI Express cards programmed with a Xilinx DMA IP.

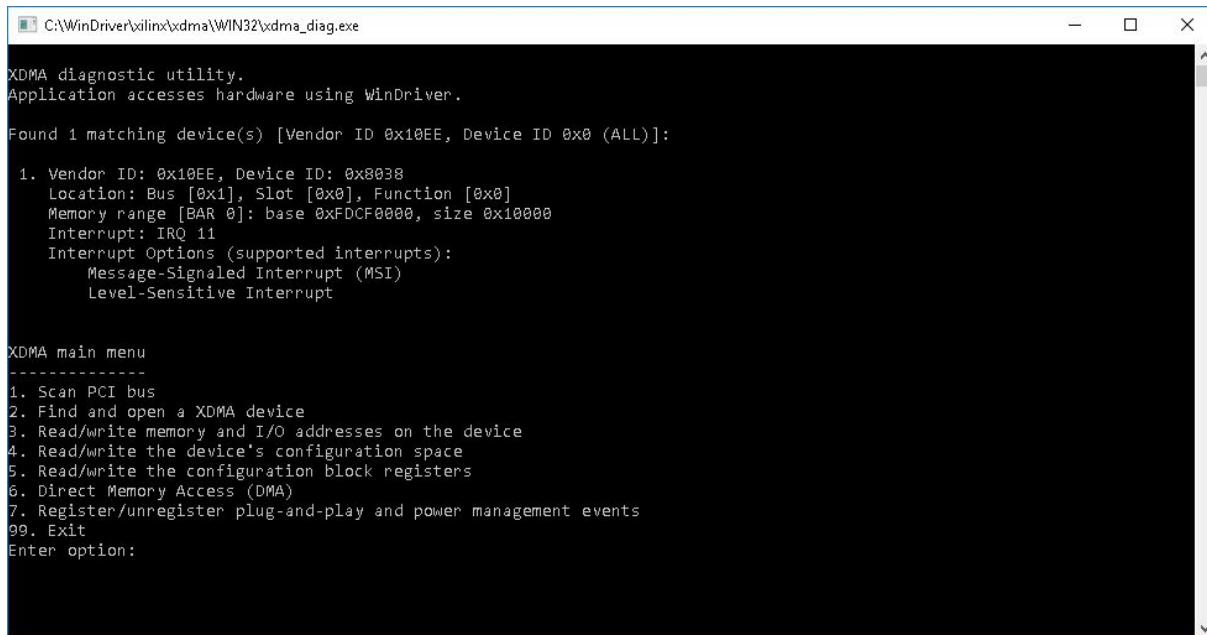
The sample source code and the pre-compiled sample can be found in the `WinDriver/samples/c/xilinx/xdma` directory.

Before running the diagnostic utility, make sure that DriverWizard is closed, to avoid a “resource overlap” error.

** Attention**

If you have a proper XDMA supporting device attached, the diagnostic utility might open it automatically on startup, as shown here. Otherwise, you can choose option 2 and to try to open a device yourself.

The xdma_diag utility



C:\WinDriver\xilinx\xdma\WIN32\xdma_diag.exe

XDMA diagnostic utility.
Application accesses hardware using WinDriver.

Found 1 matching device(s) [Vendor ID 0x10EE, Device ID 0x0 (ALL)]:

1. Vendor ID: 0x10EE, Device ID: 0x8038
Location: Bus [0x1], Slot [0x0], Function [0x0]
Memory range [BAR 0]: base 0xFDCF0000, size 0x10000
Interrupt: IRQ 11
Interrupt Options (supported interrupts):
 Message-Signaled Interrupt (MSI)
 Level-Sensitive Interrupt

XDMA main menu

1. Scan PCI bus
2. Find and open a XDMA device
3. Read/write memory and I/O addresses on the device
4. Read/write the device's configuration space
5. Read/write the configuration block registers
6. Direct Memory Access (DMA)
7. Register/unregister plug-and-play and power management events
99. Exit

Enter option:

9.3.1. Performing Direct Memory Access (DMA) tests

After choosing option 6 from the XDMA main menu, the user can either Perform a DMA transfer or Measure DMA Performance.

The DMA transfer option allows the user to actually read or write data from device.

The writing option prompts the user to type a hexadecimal 32 bit packet of data, and this packet is repeatedly written to the device's memory, according to the user entered Number of packets to transfer. The reading option prints out the contents of a certain memory area, according to the user entered FPGA offset.

A simple test can be:

- Writing a certain 32 bit packet to the device's memory.
- Reading from the same offset and making sure whether the data previously written to the device is in fact the data that is now read.

XDMA Transfer example

The DMA performance option allows the user to test the speed of the device. The user is prompted to choose a transfer direction (to device, from device, or simultaneous bi-directional transfers). Then the user is prompted to enter a buffer size for each transfer that will be made during the test, and the test's duration. Afterwards the test takes place and in the end of the test the results will be printed out.

XDMA Performance test example

```
C:\WinDriver\xilinx\xdma\WIN32\xdma_diag.exe
DMA performance
-----
1. DMA host-to-device performance
2. DMA device-to-host performance
3. DMA host-to-device and device-to-host performance running simultaneously
99. Exit menu

Enter option: 3

Select DMA completion method:
-----
1. Interrupts
2. Polling
99. Cancel
Enter option: 2

Enter single transfer buffer size in KBs (to cancel press 'x'): 30

Enter test duration in seconds (to cancel press 'x'): 3

Running DMA bi-directional performance test, wait 3 seconds to finish...

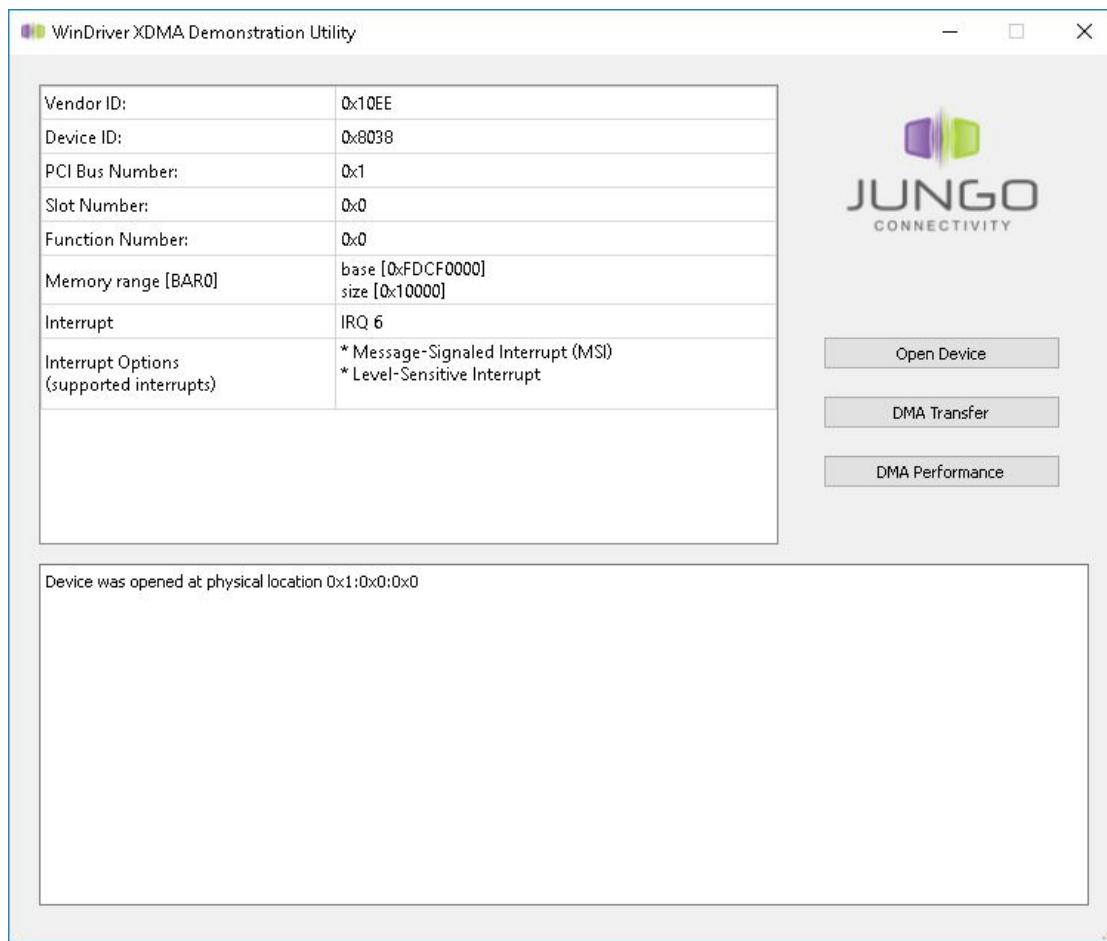
DMA host-to-device performance results:
-----
Transferred 2721[MB], test duration 3000[ms], rate 907[MB/sec]

DMA device-to-host performance results:
-----
Transferred 2745[MB], test duration 3000[ms], rate 915[MB/sec]
```

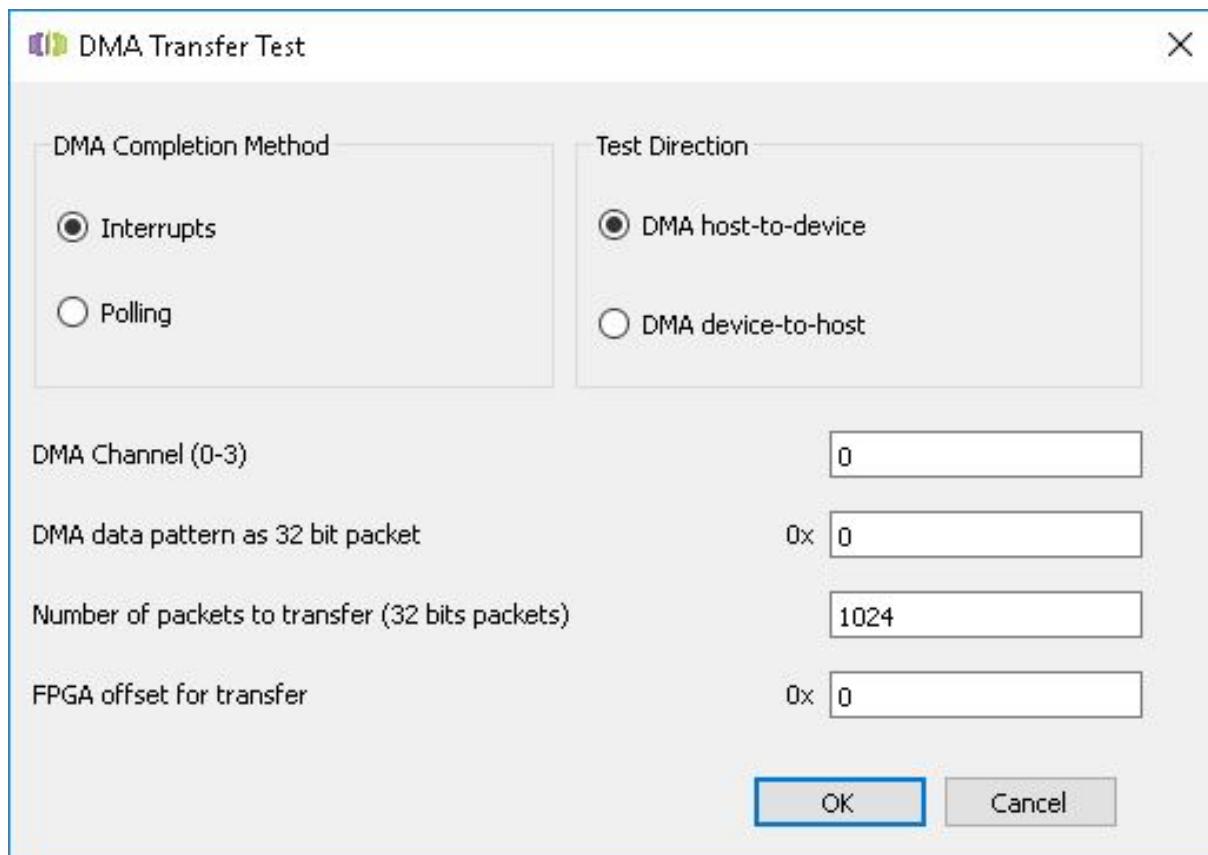
9.3.2. The XDMA GUI utility

Starting from WinDriver version 12.5, WinDriver also supplies a GUI utility, based upon the same `xdma_diag` source code, showcasing the above mentioned DMA Transfer and DMA Performance tests. The `xdma_gui` utility can be found in the `WinDriver/samples/c/xilinx/xdma/gui` directory. Similar to the console `xdma←_diag` program, the `xdma_gui` utility will try to open an XDMA device automatically, and if it will fail then it will be required to first select and open an XDMA-supported device before being able to perform the tests.

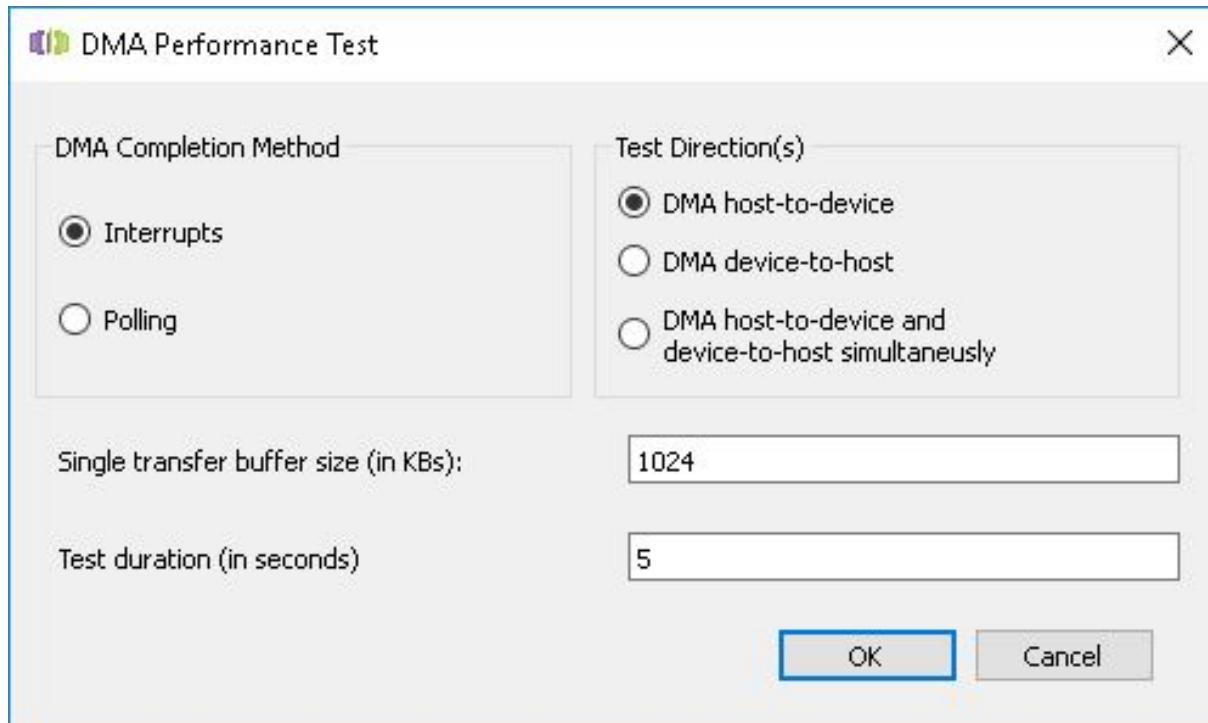
The xdma_gui Utility



DMA Transfer test in xdma_gui



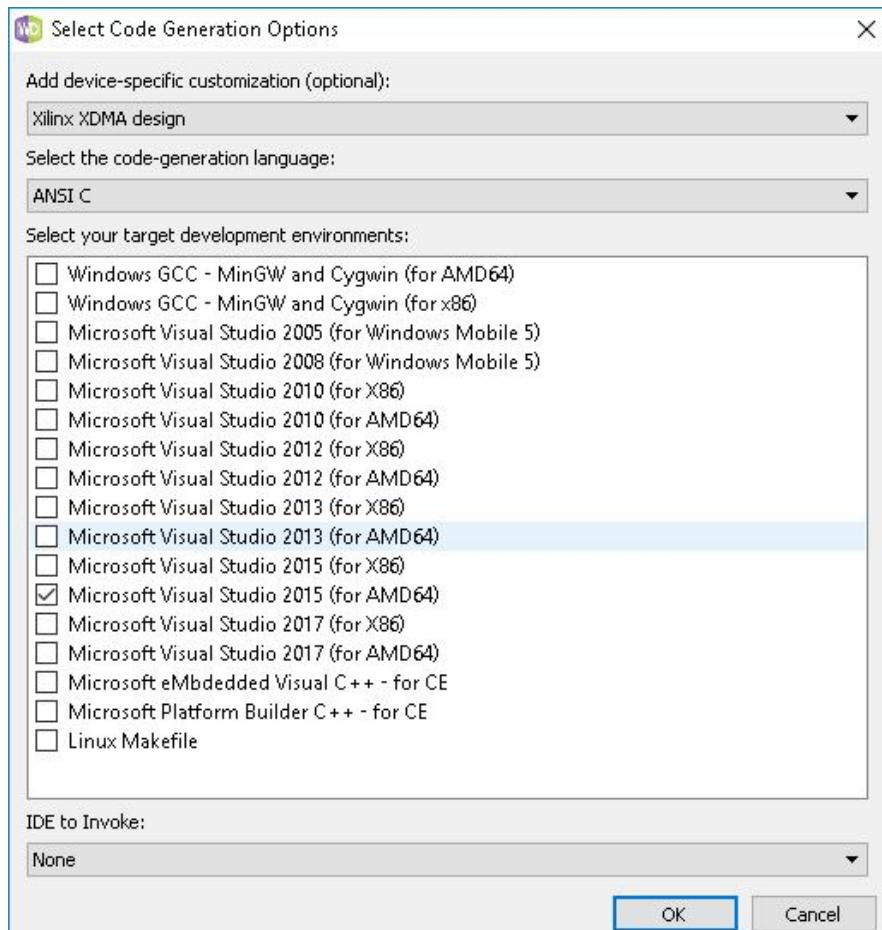
DMA Performance in the xdma_gui Utility



9.3.3. XDMA code generation in DriverWizard

Starting from WinDriver version 12.5, DriverWizard allows generating a user-mode diagnostics program source code that is similar to the supplied `xdma_diag` program, by choosing Xilinx XDMA design from the Add device specific customization (optional) menu.

Code Generation with XDMA device specific customization



9.4. The QDMA sample code

Starting from version 14.4 WinDriver supplies a cross-platform user-mode sample code of a diagnostic utility that demonstrates several features of Xilinx PCI Express cards with QDMA IP (Multi Queue DMA) support.

The sample source code and the pre-compiled sample can be found in the `WinDriver/samples/c/xilinx/qdma` directory.

Before running the diagnostic utility, make sure that DriverWizard is closed, to avoid a "resource overlap" error.

** Attention**

If you have a QDMA supporting device attached, the diagnostic utility will open it automatically on startup. Or you can choose "Find and open a QDMA device" option to try to open a device yourself.

9.4.1. Performing Direct Memory Access (DMA) transaction

After choosing "Direct Memory Access (DMA) transaction" option, the user can either Perform a DMA transaction. The DMA transaction option allows the user to actually read or write data from device.

In order to do this you will need to:

- Add a MM queue.
- Start a MM queue.
- Create a read request or write request.

If you selected Blocking method you can see the transaction details (including transaction rate) when it is finished, otherwise (Non-blocking method), you can see it in the Requests sub-menu.

9.4.2. Changing between physical functions

Several QDMA devices can be opened simultaneously. To switch between physical functions select “Change physical function” in the menu.

9.4.3. Requests info

The following actions may be done on the Requests menu:

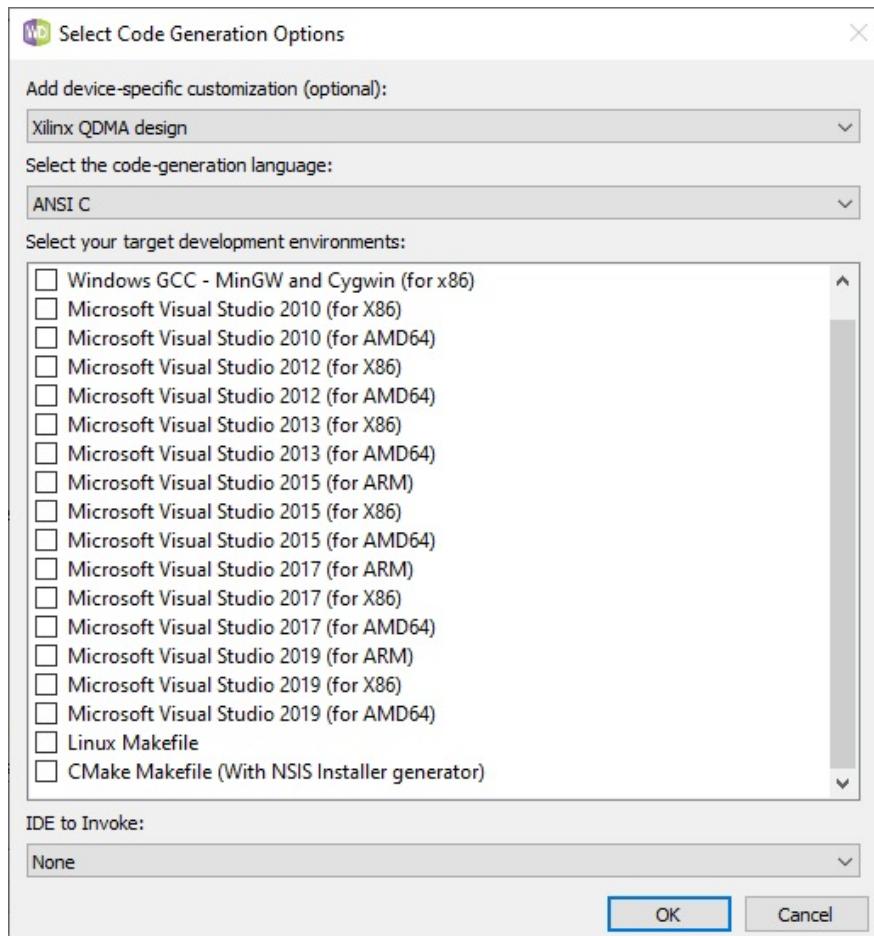
- Show requests list (read and write requests).
- Show buffer content.
- Delete request from list (Including free for the assigned buffer of the request).
- Get status of all queues.

In order to get queues status (Available/Programmed/Started), select “Get queue status”.

9.4.4. QDMA code generation in DriverWizard

Starting from WinDriver version 14.4, DriverWizard allows generating a user-mode diagnostics program source code that is similar to the supplied `qdma_diag` program, by choosing Xilinx QDMA design from the “Add device specific customization” (optional) menu.

Code Generation with QDMA



9.5. The Avalon-MM sample code

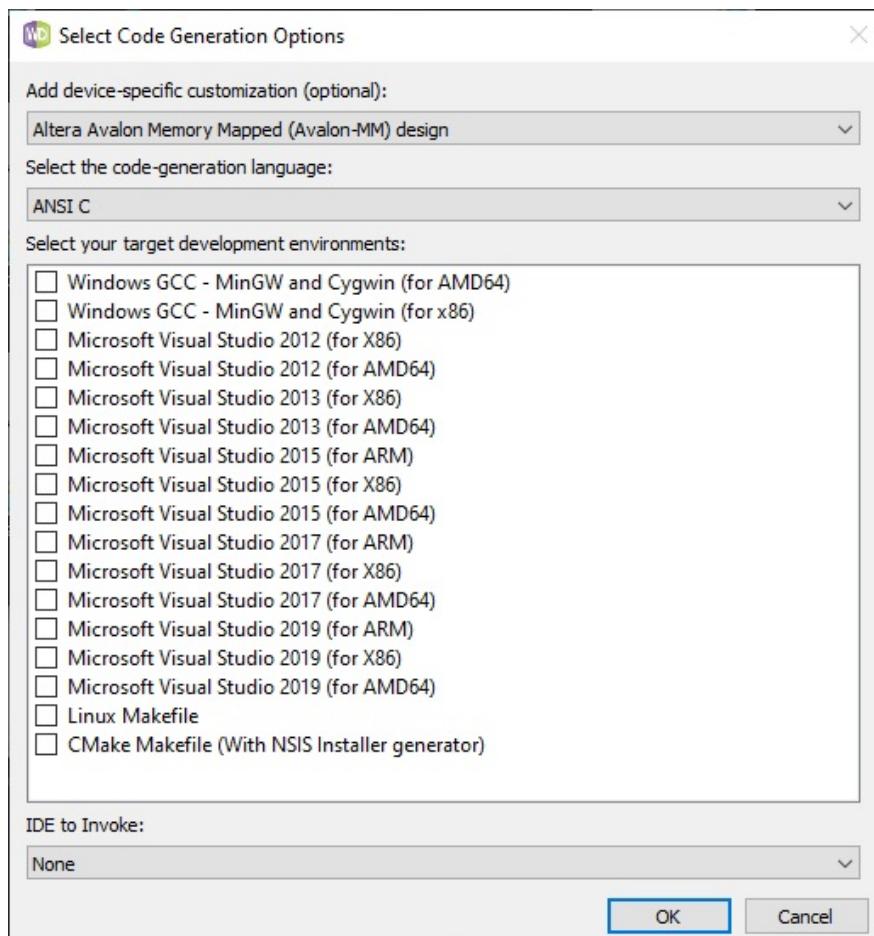
Starting from version 14.7, WinDriver supplies a user-mode sample code of a diagnostic utility that demonstrates several features of Intel Altera PCI Express cards programmed with a Avalon-MM IP.

The sample source code and the pre-compiled sample can be found in the `WinDriver/samples/c/altera/avalonmm` directory.

9.5.1. Avalon-MM code generation in DriverWizard

DriverWizard allows generating a user-mode diagnostics program source code that is similar to the supplied `avalonmm_diag` program, by choosing Altera Avalon Memory Mapped (Avalon-MM) design from the Add device specific customization (optional) menu.

Code Generation with Altera Avalon Memory Mapped (Avalon-MM) device specific customization



Chapter 10

PCI Advanced Features

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, which we discuss in [Chapter 9] ([Enhanced Support for Specific Chipsets](#)), 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.

10.1. Handling Interrupts

WinDriver provides you with API, DriverWizard code generation, and samples, 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](#), 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 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 (if required), 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 provide a general overview of PCI/ISA interrupt handling and explain how to handle interrupts using WinDriver's API. Use this information to understand the sample and generated DriverWizard interrupt code or to write your own interrupt handler.

10.1.1. Interrupt Handling - Overview

PCI and ISA hardware uses interrupts to signal the host.

** Attention**

In order to handle PCI interrupts correctly with WinDriver on Plug-and-Play (PnP) Windows operating systems, you must first install an INF file for the device, which registers it to work with WinDriver's PnP driver - `windrvr<version>.sys` (e.g., `windrvr1511.sys`).

There are two main methods of PCI interrupt handling:

- **Legacy Interrupts:** The traditional interrupt handling, which uses a line-based mechanism. In this method, interrupts are signaled by using one or more external pins that are wired "out-of-band", i.e., separately from the main bus lines. Legacy interrupts are divided into two groups:
 - **Level-sensitive interrupts:** These 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 kernel interrupt handler repeatedly, causing the host platform to hang. To prevent such a situation, the interrupt must be acknowledged (cleared) by the kernel interrupt handler immediately when it is received. Therefore, WinDriver requires you to define an interrupt-status register that will be read/written in order to clear the interrupt. This is a precautionary measurement, because a level sensitive interrupt that is not acknowledged can hang your PC. Legacy PCI interrupts are level sensitive.
 - **Edge-triggered interrupts:** These are interrupts that are generated once, when the physical interrupt signal goes from low to high. Therefore, exactly one interrupt is generated. No special action is required for acknowledging this type of interrupt. ISA/EISA interrupts are edge triggered.

- **Message-Signaled Interrupts (MSI)**: Newer PCI bus technologies, available beginning with v2.2 of the PCI bus and in PCI Express, support Message-Signaled Interrupts (MSI). This method uses "in-band" messages instead of pins, and can target addresses in the host bridge. A PCI function can request up to 32 MSI messages. MSI and MSI-X are edge triggered and do not require acknowledgment in the kernel. Among the advantages of MSIs:

- MSIs can send data along with the interrupt message.
- As opposed to legacy PCI interrupts, MSIs are not shared; i.e., an MSI that is assigned to a device is guaranteed to be unique within the system.

- **Extended Message-Signaled Interrupts (MSI-X)** are available beginning with version 3.0 of the PCI bus. This method provides an enhanced version of the MSI mechanism, which includes the following advantages:

- Supports 2,048 messages instead of 32 messages supported by the standard MSI.
- Supports independent message address and message data for each message.
- Supports per-message masking.
- Enables more flexibility when software allocates fewer vectors than hardware requests. The software can reuse the same MSI-X address and data in multiple MSI-X slots.

The newer PCI buses, which support MSI/MSI-X, maintain software compatibility with the legacy line-based interrupts mechanism by emulating legacy interrupts through in-band mechanisms. These emulated interrupts are treated as legacy interrupts by the host operating system.

WinDriver supports legacy line-based interrupts, both edge triggered and level sensitive, on all supported operating systems.

WinDriver also supports PCI MSI/MSI-X interrupts (when supported by the hardware).

WinDriver provides a single set of APIs for handling both legacy and MSI/MSI-X interrupts, as described in this manual.

10.1.2. WinDriver Interrupt Handling Sequence

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 enables you to implement kernel-mode interrupt routines. To find out how to handle interrupts from the Kernel PlugIn, please refer to [12.5.5. Handling Interrupts in the Kernel PlugIn](#).

To listen to PCI interrupts with the DriverWizard, follow these steps:

- Define the interrupt-status register: In the DriverWizard's Registers tab, select New and define a new register. You should specify the register's name, location (i.e., offset into one of the BARs), size, and access mode (read/write). The interrupt-acknowledgment information is hardware specific. You should therefore review your hardware's specification for the relevant data to set for your specific device.
- Assign the interrupt-status register to your card's interrupt: After defining the interrupt-status register, go back to the Interrupts tab and assign the interrupt that you have defined to the card's interrupt. You should select your interrupt and click on the Edit button to display the Interrupt Information dialog box. Select the register you have defined from the drop-down list in the Access Register box and fill-in the additional information required for acknowledging the interrupt - i.e., read/write mode and the data (if any) to be written to the status register in order to acknowledge and clear the interrupt. You can define several commands for execution upon an interrupt, by simply clicking the More button in the Interrupt Information window.

You should also verify that the interrupt is defined as Level Sensitive and that the Shared box is checked as PCI interrupts should generally be shared. This issue is also explained when clicking the Help button in the Interrupt Information dialog box.

You can now try to listen to the interrupts on your card with the DriverWizard, by clicking the Listen to Interrupts button in the Interrupts tab, and then generating interrupts in the hardware. The interrupts that will be received will be logged in the Log window. To stop listening to the interrupts, click the Stop Listen to Interrupts button in the Interrupts tab.

The interrupt handling sequence using WinDriver is as follows:

- The user calls one of WinDriver's interrupt enable functions - [WDC_IntEnable\(\)](#) or the low-level [InterruptEnable\(\)](#) or [WD_IntEnable\(\)](#) functions, - to enable interrupts on the device. These functions receive an optional array of read/write transfer commands to be executed in the kernel when an interrupt occurs.

Please note:

- When using WinDriver to handle level-sensitive interrupts, you must set up transfer commands for acknowledging the interrupt, as explained in [10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands](#).
- Memory allocated for the transfer commands must remain available until the interrupts are disabled.

When [WDC_IntEnable\(\)](#) or the lower-level [InterruptEnable\(\)](#) function is called, WinDriver spawns a thread for handling incoming interrupts. When using the low-level [WD_IntEnable\(\)](#) function you need to spawn the thread yourself. WinDriver must be registered with the OS as the driver of the device before enabling interrupts. For Plug-and-Play hardware (PCI/PCI Express) on Windows platforms, this association is made by installing an INF file for the device. If the INF file is not installed, the interrupt enable function() will fail with a [WD_NO_DEVICE_OBJECT] ([WD_NO_DEVICE_OBJECT](#)) error .

- The interrupt thread runs an infinite loop that waits for an interrupt to occur.
- When an interrupt occurs, WinDriver executes, in the kernel, any transfer commands that were prepared in advance by the user and passed to WinDriver's interrupt-enable functions (see [10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands](#)).
When the control returns to the user mode, the driver's user-mode interrupt handler routine (as passed to WinDriver when enabling the interrupts with [WDC_IntEnable\(\)](#) or [InterruptEnable\(\)](#)) is called.
- When the user-mode interrupt handler returns, the wait loop continues.
- When the user no longer needs to handle interrupts, or before the user-mode application exits, the relevant WinDriver interrupt disable function should be called - [WDC_IntDisable\(\)](#) or the low-level [InterruptDisable\(\)](#) or [WD_IntDisable\(\)](#) functions, (depending on the function used to enable the interrupts).

The low-level [WD_IntWait\(\)](#) WinDriver function, which is used by the high-level interrupt enable functions to wait on 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 [WD_IntWait\(\)](#) wakes up the interrupt handler thread and returns, as explained above.

Since your interrupt handler runs in the user mode, you may call any OS API from this function, including file-handling and GDI functions.

10.1.3. Registering IRQs for Non-Plug-and-Play Hardware

On Windows, you may need to register an interrupt request (IRQ) with WinDriver before you can assign it to your non-Plug-and-Play device (e.g., your ISA card).

To register an IRQ with WinDriver on Windows, follow these steps:

- Open the Device Manager and select **View | Resources** by type.
- Select a free IRQ from among those listed in the Interrupt request (IRQ) section.
- Register the selected IRQ with WinDriver:
 - Back up the files in the `WinDriver\redist` directory.
 - Edit `windrivr1511.inf`.

Add the following line in the [DriverInstall.NT] section:

`LogConfig=config_irq`

Add a config_irq section (where <IRQ> signifies your selected IRQ number - e.g., 10):

```
[config_irq]
IRQConfig=<IRQ>
```

- Reinstall WinDriver by running the following from a command-line prompt (where "<path to windrvr1511.inf>" is the path to your modified WinDriver INF file):

```
wdreg -inf <path to windrvr1511.inf> install
```

- Verify that that the IRQ was successfully registered with WinDriver: Open the Device Manager and locate the WinDriver device. The device properties should have a Resources tab with the registered IRQ.

This procedure registers the IRQ with the virtual WinDriver device. It is recommended that you rename the `windrvr1511` driver module, to avoid possible conflicts with other instances of WinDriver that may be running on the same machine (see [17.2. Renaming the WinDriver Kernel Driver](#)).

If you rename your driver, replace references to `windrvr1511.inf` in the IRQ registration instructions above with the name of your renamed WinDriver INF file.

10.1.4. Determining the Interrupt Types Supported by the Hardware

When retrieving resources information for a Plug-and-Play device using [WDC_PciGetDeviceInfo\(\)](#) or the low-level [WD_PciGetCardInfo\(\)](#) function, the function returns information regarding the interrupt types supported by the hardware. This information is returned within the `dwOptions` field of the returned interrupt resource (`pDeviceInfo->Card.Item[i].I.Int.dwOptions` for the WDC functions `pPciCard->Card.← Item[i].I.Int.dwOptions` for the low-level functions).

The interrupt options bit-mask can contain a combination of any of the following interrupt type flags:

- [INTERRUPT_MESSAGE_X] ([INTERRUPT_MESSAGE_X](#)): Extended Message-Signaled Interrupts (MSI-X).
- [INTERRUPT_MESSAGE] ([INTERRUPT_MESSAGE](#)): Message-Signaled Interrupts (MSI).
- [INTERRUPT_LEVEL_SENSITIVE] ([INTERRUPT_LEVEL_SENSITIVE](#)): Legacy level-sensitive interrupts.
- [INTERRUPT_LATCHED] ([INTERRUPT_LATCHED](#)): Legacy edge-triggered interrupts. The value of this flag is zero and it is applicable only when no other interrupt flag is set.

The [WDC_GET_INT_OPTIONS\(\)](#) macro returns a WDC device's interrupt options bitmask. You can pass the returned bit-mask to the [WDC_INT_IS_MSI\(\)](#) macro to check whether the bit-mask contains the MSI or MSI-X flags .

The [INTERRUPT_MESSAGE] ([INTERRUPT_MESSAGE](#))

and [INTERRUPT_MESSAGE_X] ([INTERRUPT_MESSAGE_X](#)) flags are applicable only to PCI devices.

The Windows APIs do not distinguish between MSI and MSI-X; therefore, on this OS the WinDriver functions set the [INTERRUPT_MESSAGE] ([INTERRUPT_MESSAGE](#)) flag for both MSI and MSI-X.

10.1.5. Determining the Interrupt Type Enabled for a PCI Card

When attempting to enable interrupts for a PCI card, WinDriver first tries to use MSI-X or MSI, if supported by the card. If this fails, WinDriver attempts to enable legacy level-sensitive interrupts.

WinDriver's interrupt-enable functions return information regarding the interrupt type that was enabled for the card. This information is returned within the `dwEnabledIntType` field of the [WD_INTERRUPT](#) structure that was passed to the function. When using the high-level [WDC_IntEnable\(\)](#) function, the information is stored within the `Int` field of the WDC device structure referred to by the function's `hDev` parameter , and can be retrieved using the [WDC_GET_ENABLED_INT_TYPE\(\)](#) low-level WDC macro.

10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands

When handling interrupts you may find the need to perform high-priority tasks at the kernel mode level immediately when an interrupt occurs. For example, when handling level-sensitive interrupts, such as legacy PCI interrupts (see

[10.1. Handling Interrupts](#)), the interrupt line must be lowered (i.e., the interrupt must be acknowledged) in the kernel, otherwise the operating system will repeatedly call WinDriver's kernel interrupt handler, causing the host platform to hang. Acknowledgment of the interrupt is hardware-specific and typically involves writing or reading from specific runtime registers on the device.

WinDriver's interrupt enable functions receive an optional pointer to an array of `WD_TRANSFER` structures , which can be used to set up read/write transfer command from/to memory or I/O addresses on the device. The `WDC_IntEnable()` function accepts this pointer and the number of commands in the array as direct parameters (`pTransCmds` and `dwNumCmds`). The low-level `InterruptEnable()` and `WD_IntEnable()` functions receive this information within the `Cmd` and `dwCmds` fields of the `WD_INTERRUPT` structure that is passed to them.

When you need to execute performance-critical transfers to/from your device upon receiving an interrupt - e.g., when handling level-sensitive interrupts - you should prepare an array of `WD_TRANSFER` structures that contain the required information regarding the read/write operations to perform in the kernel upon arrival of an interrupt, and pass this array to WinDriver's interrupt enable functions.

As explained in [10.1.2. WinDriver Interrupt Handling Sequence](#), WinDriver's kernel-mode interrupt handler will execute the transfer commands passed to it within the interrupt enable function for each interrupt that it handles, before returning the control to the user mode.

Memory allocated for the transfer commands must remain available until the interrupts are disabled .

10.1.6.1. Interrupt Mask Commands

The interrupt transfer commands array that you pass to WinDriver can also contain an interrupt mask structure, which will be used to verify the source of the interrupt. This is done by setting the transfer structure's `cmdTrans` field, which defines the type of the transfer command, to `CMD_MASK`, and setting the relevant mask in the transfer structure's `Data` field . Interrupt mask commands must be set directly after a read transfer command in the transfer commands array.

When WinDriver's kernel interrupt handler encounters a mask interrupt command, it masks the value that was read from the device in the preceding read transfer command in the array, with the mask set in the interrupt mask command. If the mask is successful, WinDriver will claim control of the interrupt, execute the rest of the transfer commands in the array, and invoke your usermode interrupt handler routine when the control returns to the user mode. However, if the mask fails, WinDriver will reject control of the interrupt, the rest of the interrupt transfer commands will not be executed, and your user-mode interrupt handler routine will not be invoked. Acceptance and rejection of the interrupt is relevant only when handling legacy interrupts; since MSI/MSI-X interrupts are not shared, WinDriver will always accept control of such interrupts.

To correctly handle shared PCI interrupts, you must always include a mask command in your interrupt transfer commands array, and set up this mask to check whether the interrupt handler should claim ownership of the interrupt.

Ownership of the interrupt will be determined according to the result of this mask. If the mask fails, no other transfer commands from the transfer commands array will be executed - including commands that preceded the mask command in the array. If the mask succeeds, WinDriver will proceed to perform any commands that precede the first mask command (and its related read command) in the transfer commands array, and then any commands that follow the mask command in the array.

To gain more flexibility and control over the interrupt handling, you can use WinDriver's Kernel Plug-In feature, which enables you to write your own kernel-mode interrupt handler routines, as explained in [12.5.5. Handling Interrupts in the Kernel PlugIn](#) of the manual.

10.1.6.2. Sample WinDriver Transfer Commands Code

This section provides sample code for setting up interrupt transfer commands using the WinDriver Card (WDC) library API .

The sample code is provided for the following scenario: Assume you have a PCI card that generates level-sensitive interrupts. 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 (`pAddr`), to be `intrMask`. In order to clear and acknowledge the interrupt you need to write 0 to the `INTCSR`.

The code below demonstrates how to define an array of transfer commands that instructs WinDriver's kernel-mode interrupt handler to do the following:

- Read your card's INTCSR register and save its value.
- Mask the read INTCSR value against the given mask (`intrMask`) to verify the source of the interrupt.
- If the mask was successful, write 0 to the INTCSR to acknowledge the interrupt.

** Attention**

All commands in the example are performed in modes of DWORD.

Example:

```
WD_TRANSFER trans[3]; /* Array of 3 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].pPort = pAddr; /* Assume pAddr holds the address of the 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].pPort = pAddr; /* Assume pAddr 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. Memory allocated for the transfer commands must remain available until the interrupts are disabled , as explained above.

The following code demonstrates how to use the `WDC_IntEnable()` function 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 WDC_IntEnable().
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);
```

10.1.7. WinDriver MSI/MSI-X Interrupt Handling

WinDriver supports PCI Message-Signaled Interrupts (MSI) and Extended Message-Signaled Interrupts (MSI-X).

The same APIs are used for handling both legacy and MSI/MSI-X interrupts, including APIs for retrieving the interrupt types supported by your hardware (see [10.1.4. Determining the Interrupt Types Supported by the Hardware](#)) and the interrupt type that was enabled for it (see [10.1.5. Determining the Interrupt Type Enabled for a PCI Card](#)).

When enabling interrupts for a PCI device on an OS that supports MSI/MSIx, WinDriver first tries to enable MSI-X or MSI - if supported by the device - and if this fails, it attempts to enable legacy level-sensitive interrupts.

On Windows, enabling MSI or MSIx interrupts requires that a relevant INF file first be installed for the device, as explained in [10.1.7.1. Windows MSI/MSI-X Device INF Files](#).

On Linux, you can specify the types of PCI interrupts that may be enabled for your device, via the dwOptions parameter of the `WDC_IntEnable()` function or of the lowlevel `InterruptEnable()` function - in which case WinDriver will only attempt to enable interrupts of the specified types (provided they are supported by the device).

WinDriver's kernel-mode interrupt handler sets the interrupt message data in the `dwLastMessage` field of the `WD_INTERRUPT` structure that was passed to the interrupt enable/wait function. If you pass the same interrupt structure as part of the data to your user mode interrupt handler routine, as demonstrated in the sample and generated DriverWizard interrupt code, you will be able to access this information from your interrupt handler. When using a Kernel Plugin driver (see [Understanding the Kernel Plugin](#)), the last message data is passed to your kernel mode `KP_IntAtDpcMSI` handler; it is also passed to `KP_IntAtIrqlMSI` if you have enabled MSI interrupts. You can use the low-level `WDC_GET_ENABLED_INT_LAST_MSG()` macro to retrieve the last message data for a given WDC device.

10.1.7.1. Windows MSI/MSI-X Device INF Files

The information in this section is relevant only when working on Windows.

To successfully handle PCI interrupts with WinDriver on Windows, you must first install an INF file that registers your PCI card to work with WinDriver's kernel driver, as explained in [17.1. Windows INF Files](#). To use MSI/MSI-X on Windows, the card's INF file must contain specific [Install.NT.HW] MSI information, as demonstrated below:

```
[Install.NT.HW]
AddReg = Install.NT.HW.AddReg
[Install.NT.HW.AddReg]
HKR, "Interrupt Management", 0x000000010
HKR, "Interrupt Management\MessageSighaledInterruptProperties", 0x000000010
HKR, "Interrupt Management\MessageSighaledInterruptProperties", MSISupported, 0x10001, 1
```

Therefore, to use MSI/MSI-X on Windows with WinDriver - provided your hardware supports MSI/MSI-X - you need to install an appropriate INF file. When using DriverWizard on Windows to generate an INF file for a PCI device that supports MSI/MSI-X, the INF generation dialogue allows you to select to generate an INF file that supports MSI/MSI-X (see [6.2. DriverWizard Walkthrough](#)).

In addition, the WinDriver sample code for the Xilinx Bus Master DMA (BMD) design, which demonstrates MSI handling, includes a sample MSI INF file for this design - `WinDriver/samples/c/xilinx/bmd_design/xilinx_bmd.inf`.

If your card's INF file does not include MSI/MSI-X information, as detailed above, WinDriver will attempt to handle your card's interrupts using the legacy level-sensitive interrupt handling method, even if your hardware supports MSI/MSI-X.

10.1.8. Sample User-Mode WinDriver Interrupt Handling Code

The sample code below demonstrates how you can use the WDC library's interrupt APIs to implement a simple user-mode interrupt handler.

For a complete interrupt handler source code that uses the WDC interrupt functions, refer, for example, to the `WinDriver/pci_diag` (`WinDriver/samples/c/pci_diag`) and `PLX` (`WinDriver/samples/c/plx`) samples, and to the generated DriverWizard PCI/ISA code. For a sample of MSI interrupt handling, using the same APIs, refer to the Xilinx Bus Master DMA (BMD) design sample (`WinDriver/samples/c/xilinx/bmd_design`), or to the code generated by DriverWizard for PCI hardware that supports MSI/MSI-X.

The following sample code demonstrates interrupt handling for an edge-triggered ISA card. The code does not set up any kernel-mode interrupt transfer commands (see [10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands](#)), which is acceptable in the case of edge-triggered or MSI/MSI-X interrupts (see [10.1.1. Interrupt Handling - Overview](#)). Note that when using WinDriver to handle level-sensitive interrupts from the user mode, you must set up transfer commands for acknowledging the interrupt in the kernel, as explained above and as demonstrated in [10.1.6. Setting Up Kernel-Mode Interrupt Transfer Commands](#).

As mentioned in this Chapter, WinDriver provides a single set of APIs for handling both legacy and MSI/MSI-X interrupts. You can therefore also use the following code to handle MSI/MSI-X PCI interrupts (if supported by your hardware), by simply replacing the use of `WDC_IsaDeviceOpen()` in the sample with `WDC_PciDeviceOpen()`.

```
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_PciDeviceOpen(...);
    ...
    /* 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();  
     * InterruptDisable() calls WD_IntDisable(). */  
}  
...  
WDC_PciDeviceClose(hDev);  
...  
WDC_DriverClose();  
}
```

10.2. Reserving and locking physical memory on Windows and Linux

This chapter explains how to reserve a segment of the physical memory (RAM) for exclusive use, and then access it using WinDriver, on Windows or Linux.

** Attention**

In most cases, there is no need to resort to this method in order to reserve segments of memory for exclusive use. Normally, you can lock a safe Direct Memory Access (DMA) buffer (e.g., using WinDriver's DMA APIs) and then access the buffer from your driver. For more info, see [11.2. Performing Direct Memory Access \(DMA\)](#) or [11.3. Performing Direct Memory Access \(DMA\) transactions](#).

The method described in this document should be used only in rare cases of “memory-intensive” driver projects and only when the required memory block cannot be locked using standard methods, such as allocation of a contiguous DMA buffer. When using this method, take special care not to write to the wrong memory addresses, so as to avoid system crashes, etc. The relevant device must have an INF file installed.

This chapter is organized as follows:

- Reserving the desired amount of RAM: Windows and Linux
- On Windows – Calculating the base address of the reserved memory
- Using WinDriver to access reserved memory

Reserving the desired amount of RAM

** Attention**

Reserving too much space may result in degraded OS performance.

- Windows

Run the command line as an administrator, and use the BCDEdit utility to set the value of `removememory` to the number of MB you wish to reserve. Upon successful completion, BCDEdit will display a success message. To complete reservation, reboot the PC.

```
bcdeedit /set removememory specify_size_in_MB
```

For Example:

```
bcdeedit /set removememory 10
```

- Linux

Run the following command to view a list of the physical memory ranges on your machine
`dmesg | grep BIOS`

This produces entries as in the following sample output usable identifies memory sections that are used by Linux:

```
[ 0.000000] BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
[ 0.000000] BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
[ 0.000000] BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
[ 0.000000] BIOS-e820: 0000000000100000 - 0000000007dce0000 (usable)
[ 0.000000] BIOS-e820: 0000000007dce0000 - 0000000007dce3000 (ACPI NVS)
[ 0.000000] BIOS-e820: 0000000007dce3000 - 0000000007dcf0000 (ACPI data)
[ 0.000000] BIOS-e820: 0000000007dcf0000 - 0000000007dd00000 (reserved)
[ 0.000000] BIOS-e820: 00000000c0000000 - 00000000d0000000 (reserved)
[ 0.000000] BIOS-e820: 00000000fec00000 - 0000000100000000 (reserved)
```

Edit the boot-loader command line to instruct the Linux kernel to boot with less memory in the desired address range. For example, the following line from the sample dmesg output shown above:

```
[ 0.000000] BIOS-e820: 0000000000100000 - 0000000007dce0000 (usable)
```

indicates that there is a ~1.95GB address range, starting at address 0x100000, that is used by Linux. To reserve ~150MB of memory at the end of this range for DMA, on a machine with a GRUB boot loader, add the following to the grub file:

```
GRUB_CMDLINE_LINUX="${GRUB_CMDLINE_LINUX} mem=1800M memmap=1800M@1M"
```

This instructs Linux to boot with ~1,800MB ("mem=1800M") starting at address 0x100000 ("@1M"). Reconfigure GRUB to apply the changes:

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

- Reboot Linux
- Run the command

```
dmesg | grep BIOS\|user
```

to see the available physical memory ranges. The output should include a BIOS-provided physical RAM mappings section with the BIOS-.... lines from the original dmesg output, and a new user-defined RAM mappings section with user: ... lines indicating the actual available physical memory ranges (based on user definitions). The user entry for the memory range you modified in the previous steps should be missing the portion you reserved. For example, for the original BIOS mappings and boot-loader changes examples provided in the previous steps, the new output should look like similar to this:

```
[ 0.000000] BIOS-provided physical RAM map:
[ 0.000000] BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
[ 0.000000] BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
[ 0.000000] BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
[ 0.000000] BIOS-e820: 0000000000100000 - 0000000007dce0000 (usable)
[ 0.000000] BIOS-e820: 0000000007dce0000 - 0000000007dce3000 (ACPI NVS)
[ 0.000000] BIOS-e820: 0000000007dce3000 - 0000000007dcf0000 (ACPI data)
[ 0.000000] BIOS-e820: 0000000007dcf0000 - 0000000007dd00000 (reserved)
[ 0.000000] BIOS-e820: 00000000c0000000 - 00000000d0000000 (reserved)
[ 0.000000] BIOS-e820: 00000000fec00000 - 0000000100000000 (reserved) [ 0.000000] user-defined physical RAM map:
[ 0.000000] user: 0000000000000000 - 000000000009f800 (usable)
[ 0.000000] user: 000000000009f800 - 00000000000a0000 (reserved)
[ 0.000000] user: 00000000000f0000 - 0000000000100000 (reserved)
[ 0.000000] user: 0000000000100000 - 00000000070900000 (usable)
[ 0.000000] user: 0000000007dce0000 - 0000000007dce3000 (ACPI NVS)
[ 0.000000] user: 0000000007dce3000 - 0000000007dcf0000 (ACPI data)
[ 0.000000] user: 0000000007dcf0000 - 0000000007dd00000 (reserved)
[ 0.000000] user: 00000000c0000000 - 00000000d0000000 (reserved)
```

Comparing the following line from the BIOS-provided mapping section:

```
[ 0.000000] BIOS-e820: 0000000000100000 - 0000000007dce0000 (usable)
```

with the following line from the user-defined mapping section:

```
[ 0.000000] user: 0000000000100000 - 00000000070900000 (usable)
```

shows that the original ~1.95GB memory range - 0x100000 – 0x7dce0000 was reduced to a ~1.75GB range - 0x100000–0x70900000. The memory in address range 0x70900000 – 0x7dce0000 is no longer available because it has been reserved in the previous steps, allowing you to use this range for DMA.

Calculating the base address

To acquire the base address of the reserved memory segment on Windows, you must first determine the physical memory mapping on your PC and retrieve the base address and length (in bytes) of the highest address space used by the operating system. Then add the length of this address space to its base address to receive the base address of your reserved memory segment:

Reserved memory base address = Highest OS physical memory base address + length of the highest OS memory base address

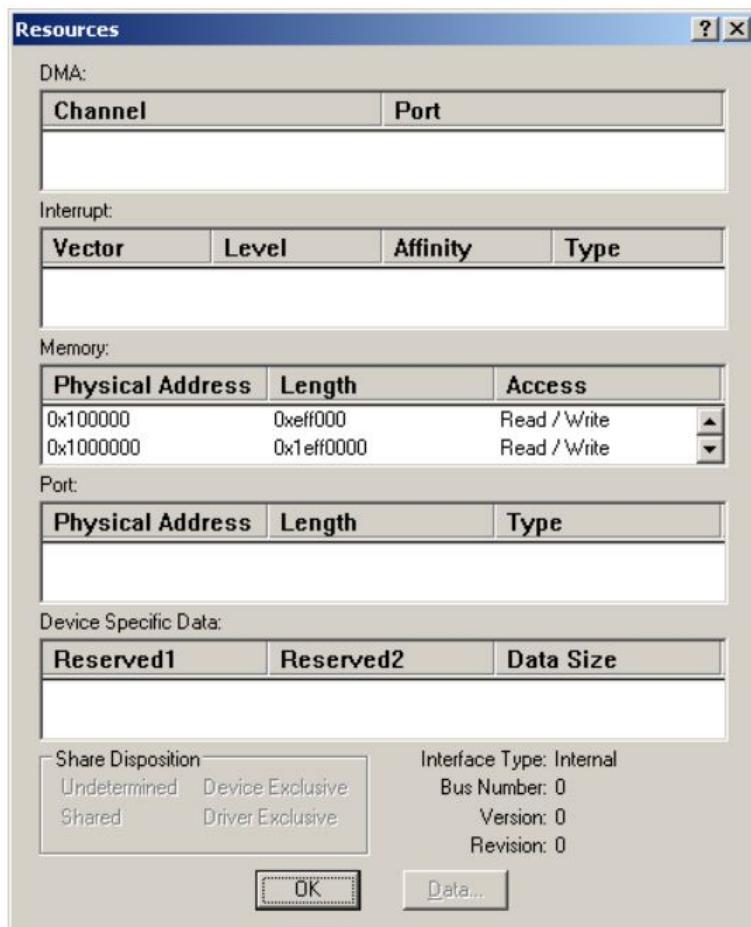
To verify the size of your reserved memory block, compare the length of the highest OS address space, before and after modifying boot configuration to reserve the memory. This can be done as follows:

- Open the registry (Start → Run → regedit.exe)
- Navigate to the registry key: HKEY_LOCAL_MACHINE\HARDWARE\RESOURCERMAP\System Resources\Physical Memory\.Translated

This key is of type REG_RESOURCE_LIST and holds information regarding the physical memory mapping on your PC. To view a parsed version of the mapped addresses, double-click on the Translated key, select the relevant resource from the Resource Lists dialog, and double-click on the resource (or select Display...) in order to display the resources dialog, which contains a list of all memory address ranges for the selected resource. The base address for your reserved physical memory block is calculated by locating the highest base address in the list and adding to it the length of the relevant address space.

For example, for the following Resources dialog, the highest base address is 0x1000000 and the length of the address space that begins at this address is 0x1eff0000, so the base address of the reserved memory is 0x1fff0000.
reserved memory base address = 0x1000000 + 0x1eff0000 = 0x1fff0000

RAM Reserved



Using WinDriver with reserved memory

Once you acquire the physical base address of the memory segment that you reserved, you can easily access it using WinDriver, as you would any memory on an ISA card. You can use WinDriver's DriverWizard to test the access to the memory you reserved: use the wizard to create a new ISA project, define the new memory item according to the information you acquired in the previous step(s), then proceed to access the memory with the wizard. You can also use DriverWizard to generate a sample diagnostics application that demonstrates how to lock and access the reserved memory region using WinDriver's API. The following code segment demonstrates how you can define and lock the physical memory using WinDriver's WDC API. The handle returned by the sample LockReservedMemory() function can be used to access the memory using WinDriver's WDC memory access APIs, defined in the WinDriver/include/wdc_lib.h header file.

** Attention**

There may be differences between the API in your version of WinDriver and that used in the following example (such as differences in field names and/or types).

```

/* LockReservedMemory: Returns a WDC handle for accessing
   the reserved memory block.

Parameters:
  pReservedRegionBase: The physical base address of
    the reserved memory region (as calculated in the
    previous step)
  qwReservedRegionLength: The length (in bytes) of the
    reserved memory region, i.e.,:
    <size_in_MB> (as configured in Step 1) * 0x100000

Note:
  The function uses the high-level WDC APIs.
  You can implement similar code using the low-level
  WD_CardRegister() API - see the WinDriver User's
  Manual for details.

*/
WDC_DEVICE_HANDLE LockReservedMemory(PHYS_ADDR pReservedRegionBase,
                                      UINT64 qwReservedRegionLength)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev = NULL;
    WD_CARD deviceInfo;
    /* Set the reserved memory's resources information */
    BZERO(deviceInfo);
    SetMemoryItem(&deviceInfo, pReservedRegionBase,
                  qwReservedRegionLength);
    /* Get a handle to the reserved memory block */
    dwStatus = WDC_IsaDeviceOpen(&hDev, &deviceInfo, NULL,
                                 NULL, NULL, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf( "Failed opening a WDC device handle.
                 Error 0x%lx - %s\n", dwStatus,
                 Stat2Str(dwStatus));
        return NULL;
    }
    /* Return the handle to the reserved memory */
    return hDev;
}
/* SetMemoryItem: Initializes a WDC device information
   structure for a specified memory region.

Parameters:
  pDeviceInfo: Pointer to a resources information structure
  pPhysAddr:      The memory region's physical base address
  qwBytes:        The memory region's length, in bytes

*/
void SetMemoryItem(WD_CARD *pDeviceInfo, PHYS_ADDR pPhysAddr, UINT64 qwBytes)
{
    WD_ITEMS *pItem = pDeviceInfo->Item;
    pDeviceInfo->dwItems = 2;
    /* 1st item: Bus */
    pItem[0].item = ITEM_BUS;
    pItem[0].I.Bus.dwBusType = WD_BUS_ISA;
    /* 2nd item: Memory */
    pItem[1].item = ITEM_MEMORY;
    /* Lock the memory for exclusive use */
    pItem[1].fNotSharable = 1;
    /* Set the reserved memory's base address */
    pItem[1].I.Mem.pPhysicalAddr = pPhysAddr;
    /* Set the reserved memory's size */
    pItem[1].I.Mem.qwBytes = qwBytes;
    /* Set the reserved memory's address space */
    pItem[1].I.Mem.dwBar = 0;
    /* Map physical memory as cached (applicable only to RAM). */
    pItem[1].I.Mem.dwOptions = WD_ITEM_MEM_ALLOW_CACHE;
}

```

10.3. Buffer sharing between multiple processes

This section describes how to share a contiguous DMA buffer or a kernel buffer between multiple processes.

The buffer sharing mechanism can be used to improve the work of multiple processes by allowing the developer avoiding unnecessary buffer copying. Coupled with the WinDriver IPC mechanism it might also be used as a way for two processes to convey large messages/data. Currently WinDriver supports sharing DMA contiguous buffer (See [WDC_DMAContigBufLock\(\)](#)) and Kernel buffer (See [WDS_SharedBufferAlloc\(\)](#)).

In order to share a buffer the developer must first pass its global handle to the other process(es). See Macros [WDS_SharedBufferGetGlobalHandle\(\)](#) and [WDC_DMAGetGlobalHandle\(\)](#) for getting a buffer's global handle.

Than, the handle should be passed to the other process(es). E.g. by WinDriver IPC calls: [WDS_IpcUidUnicast\(\)](#), [WDS_IpcSubGroupMulticast\(\)](#) or [WDS_IpcMulticast\(\)](#).

After a process gets the global buffer handle, it should use [WDC_DMABufGet\(\)](#) or [WDS_SharedBufferGet\(\)](#) to retrieve the buffer. WinDriver will re-map the buffer to the new process virtual memory (I.e. user-space) and will fill all the necessary information as though the buffer was allocated from the calling process.

Once the process no longer needs the shared buffer it should use the regular unlock/free calls - [WDC_DMABufUnlock\(\)](#) appropriately.

WinDriver manages all system resources, hence the call to [WDC_DMABufUnlock\(\)](#) will not remove system resources until all processes no longer use the buffer.

10.4. Single Root I/O Virtualization (SR-IOV)

10.4.1. Introduction

Single Root I/O Virtualization (SR-IOV) is a PCIe extended capability which makes one physical device appear as multiple virtual devices. The physical device is referred to as Physical Function (PF) while the virtual devices are referred to as Virtual Functions (VF).

Allocation of VF can be dynamically controlled by the PF via registers encapsulated in the capability. By default, SR-IOV is disabled and the PF behaves as a regular PCIe device. Once it is turned on, each VF's PCI configuration space can be accessed by its own domain, bus, slot and function number (Routing ID). Each VF also has a PCI memory space, which is used to map its register set. A VF device driver operates on the register set so it can be functional and appear as a real PCI device.

At the moment, WinDriver only supports SR-IOV in Linux.

Important notes:

- VFs have to be the same type of device as the PF.
- SR-IOV requires hardware support, as the number of VFs that can be presented depends upon the device.
- The PCI SIG SR-IOV specification indicates that each device can have up to 256 VFs, but practical limits are usually lower, as each VF requires actual hardware resources.
- VFs can't be used to configure the device.

10.4.2. Using SR-IOV with WinDriver

The following steps were required to operate the Intel 82599ES 10-Gigabit SFI/SFP+ Network Adaptor. Other cards might require different adjustments.

Make sure that prior to installing WinDriver you have run `./configure --enable-sriov-support` from the `WinDriver/redist` directory.

The `pci_diag` application may be run from the `WinDriver/samples/c/pci_diag` directory, in order to show relevant SR-IOV options.

It is possible that upon boot it will be required to add `pci=assign-busses` to the boot arguments in the kernel command line. This tells the kernel to always assign all PCI bus numbers, overriding whatever the firmware may have done.

Otherwise, assigning VF to a device may result in "bus number is out of range" error. This argument can be added by editing the file `/etc/default/grub/`, and adding it to the `GRUB_CMDLINE_LINUX_DEFAULT` variable. Then run `update-grub` to make this change permanent. Under Ubuntu, a temporary change of this variable can be done by pressing 'e' when the GRUB menu, and appending the argument to the 'linux' line. WinDriver provides you with the API for enabling, disabling and getting the number of assigned virtual functions - see the description of [WDC_PciSriovEnable\(\)](#), [WDC_PciSriovDisable\(\)](#), and [WDC_PciSriovGetNumVFs\(\)](#) respectively.

10.5. Using WinDriver's IPC APIs

This section describes how to use WinDriver to perform Inter-Process Communication.

WinDriver IPC allows several WinDriver-based user applications to send and receive user defined messages.

The `pci_diag` sample found in `WinDriver/util` (source code found in `WinDriver/samples/c/pci->_diag`) exemplifies usage of this ability. Using IPC with WinDriver requires having a license that includes IPC support. IPC APIs will fully work during the Evaluation period.

10.5.1 IPC Overview

You should define a group ID that will be used by all your applications, and optionally you can also define several sub-group IDs to differentiate between different types of applications. For example, define one sub-group ID for operational applications, and another sub-group ID for monitoring or debug applications. This will enable you to send messages only to one type of application.

An IPC message can be sent in three ways:

- Multicast - The message will be sent to all processes that were registered with the same group ID.
- Sub-group Multicast - The message will be sent to all processes that were registered with the same sub-group ID.
- Unicast (By WinDriver IPC unique ID) - The message will be sent to one specific process.

In order to start working with WinDriver IPC each user application must first register by calling `WDS_IpcRegister()`. It must provide a group ID and a sub-group ID and can optionally provides callback function to receive incoming messages. WinDriver IPC unique ID is received either in the result of `WDS_IpcScanProcs()` or in the message received by the callback function. Multicast messages may be sent by calling `WDS_IpcMulticast()` or `WDS_IpcSubGroupMulticast()`, while unicast messages may be sent by WinDriver IPC unique ID by calling `WDS_IpcUidUnicast()`.

Query of current registered processes can be done using `WDS_IpcScanProcs()`.

10.5.2. Shared Interrupts via IPC

A method of acknowledging interrupts from more than one process is by using the Shared Interrupts via IPC API provided by WinDriver.

This mechanism allows creating a special IPC "process" that sends IPC messages to other processes that are registered to WinDriver IPC.

10.5.2.1. Enabling Shared Interrupts:

A recommended method would be: In each process that you'd like to enable the Shared Interrupts in:

- Register the process to IPC using `WDS_IpcRegister()`.
- Write your own callback function that is relevant to what you wish to occur when a Shared Interrupt occurs. Send a pointer to that function as an argument to `WDS_SharedIntEnable()`. You can write a different callback for each process to achieve a different handling of the interrupt in different contexts.

10.5.2.2. Disabling Shared Interrupts:

You can either disable Shared Interrupts locally (for the current process only) using `WDS_SharedIntDisableLocal()` or disable Shared Interrupts for all processes by using `WDS_SharedIntDisableGlobal()`.

10.6. FAQ

10.6.1. What is the significance of marking a resource as 'shared' with WinDriver, and how can I verify the 'shared' status of a specific resource on my card?

The "shared" status determines whether the resource is locked for exclusive use. When a memory, I/O, or interrupt item is defined as non-sharable, the device registration function - [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#), or the low-level [WD_CardRegister\(\)](#) function - checks whether the resource is already locked for exclusive use. If it is, the registration fails; otherwise, the resource is locked exclusively, and any subsequent attempt to lock the resource - either from within the same application, or from another WinDriver application - will fail, until the resource is released using [WDC_PciDeviceClose\(\)](#) / [WDC_IsaDeviceClose\(\)](#), or the low-level [WD_CardUnregister\(\)](#) function.

** Attention**

Device registers cannot be defined as shared.

PCI resources, and specifically interrupts, should generally be defined as sharable.

For memory and I/O, the share status defined with WinDriver affects only WinDriver-based applications. However, for interrupts there is further significance to the share status, since it also determines whether other drivers (not necessarily WinDriver drivers) can lock the same interrupt. For an explanation of how to read the value of the PCI interrupt status register, please see futher FAQ questions.

Use one of the following methods to check whether a resource is defined as sharable:

- From your code, print the value of the `fNotSharable` flag for the memory/I/O/interrupt item that you wish to check - `deviceInfo.Card.Item[i].fNotSharable` when using the WDC API, or `cardReg.Card.Item[i].fNotSharable` when using the low-level WinDriver API. A value of 1 indicates that the resources is not sharable. A value of 0 indicates that the resource is sharable.
- If you have used the DriverWizard to generate your code, you can run the DriverWizard, open the `xxx.wdp` DriverWizard project file, which served as the basis for your application, select the **Memory**, **I/O**, or **Interrupt** tab (respectively) and click on **Edit**. Then look to see if the **Shared** box in the Resource Information window is checked, to signify that the resource is defined as shared for this device. Note that this test is only relevant when using the original `xxx.wdp` project file for your application code, and assuming your code does not modify the "shared" status - since any changes made in the `xxx.wdp` file, via the DriverWizard, will only affect subsequent code generation, and will not affect the existing application code. Therefore, checking the "shared" values from the code is more accurate.

As a general guideline, avoid locking the same resource twice. It is always recommend to release a previously locked resource before trying to lock it again (either from the same application, or from different applications). If you need to access the resources from several applications, you can share the handle received from a single call to the device registration function - [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#), or the low-level [WD_CardRegister\(\)](#) function.

10.6.2. Can I access the same device, simultaneously, from several WinDriver applications?

It is possible to create several WinDriver-based driver processes that will access the same card, although we do not recommend this, and it should not generally be required.

Should you decide to implement such a design, consider the synchronization issues carefully. To bypass the synchronization problems, we recommend you use only one point of access to your hardware. Use a single process to directly access your hardware, while the other processes should access the hardware only via this process. The advantage of this design is that only one point requires synchronization.

Please note, however, that we will not be able to provide technical support specifically related to the implementation of such designs and accompanying issues (such as synchronization problems that might occur). You should therefore carefully consider if this is indeed the desired design in your case.

For USB, note that while multiple calls to [WDU_Init\(\)](#) for the same device may succeed, after the first attach callback accepts control of the device no other attach notifications will be received until [WDU_Uninit\(\)](#) is called for the relevant [WDU_Init\(\)](#) call. Using a single process to perform a single [WDU_Init\(\)](#) call with a single attach callback function, as suggested above, will eliminate problems resulting from multiple [WDU_Init\(\)](#) calls.

You may consider accessing the device from one application, and communicate with other apps using the IPC API.

10.6.3. How can I read the value of the PCI interrupt status register from my WinDriver ISR, in order to determine, for example, which card generated the interrupt when the IRQ is shared between several devices?

When handling the interrupts from the user mode you must acknowledge (clear) the interrupt on the card whether your card generated the interrupt or not (in case of shared IRQs). However, this does not necessarily present a problem.

To determine which card generated the interrupt, and activate your ISR only in case the interrupt was generated by your card, you can set up the transfer commands of the [WD_INTERRUPT](#) structure, which is passed to [WDC_IntEnable\(\)](#) / [InterruptEnable\(\)](#), or to the lower level [WD_IntEnable\(\)](#) function - `int.Cmds` - to include a command to read from the interrupt register before clearing the interrupt. The register will then be read in the kernel, immediately when an interrupt is received and before it is cleared, and you will be able to access the read value from your user mode interrupt handler routine when it is activated.

** Attention**

In order to save the read value you must set the `INTERRUPT_CMD_COPY` flag in the `dwOptions` field of the [WD_INTERRUPT](#) structure (`int.dwOptions |= INTERRUPT_CMD_COPY`).

When using WinDriver's Kernel PlugIn feature to handle the interrupts directly in the kernel, you can read the value of the interrupt status register on your card directly when an interrupt is received, from within your `[KP_IntAtirql()]` ([KP_PCI_IntAtirql](#)) routine, and proceed to acknowledge (clear) the interrupt and execute your ISR only if the value of the status register indicates that the interrupt was indeed generated by your card. For more information on how to handle shared PCI interrupts from the Kernel PlugIn, please refer to the Kernel PlugIn chapter of this manual.

10.6.4. I need to be able to count the number of interrupts occurring and possibly call a routine every time an interrupt occurs. Is this possible with WinDriver?

Yes. You can use the DriverWizard to generate a diagnostics code for your device, which includes a skeletal interrupt handling mechanism that also monitors the number of interrupts received. You may need to modify the code slightly to suit your hardware's specification with regard to interrupt handling.

You can also refer to the `int_io` sample, which is available under the `WinDriver/samples/c/int_io/` directory, for an example of handling ISA interrupts with WinDriver.

Both the sample and the generated code will install an interrupt and spawn a thread that waits on it. The number of interrupts received is returned in the `dwCounter` field of the [WD_INTERRUPT](#) structure.

10.6.5. Does WinDriver poll the interrupt (Busy Wait)?

No. When calling [WD_IntWait\(\)](#) (which is called from the high-level [WDC_IntEnable\(\)](#) and [InterruptEnable\(\)](#) APIs) the calling thread is put to sleep. CPU cycles are not wasted on waiting. When an interrupt occurs the thread is awoken and you can perform your interrupt handling.

10.6.6. Can I write to disk files during an interrupt routine?

Yes. WinDriver enables you to implement your interrupt service routine (ISR) in the user mode, thereby allowing you to perform all library function calls from within your ISR. Just remember to keep it short, so you don't miss the next interrupt.

If you select to implement your ISR in the kernel, using WinDriver's Kernel PlugIn feature, you will not be able to use the standard user mode file I/O functions. You can either leave the file I/O handling in the user-mode ISR, and implement your interrupt code so that the user-mode interrupt routine is executed only once every several interrupts. You can refer to the generated DriverWizard Kernel PlugIn code and/or to the generic WinDriver Kernel PlugIn sample - WinDriver/samples/c/pci_diag/kp_pci/kp_pci.c - for an example of monitoring the interrupt count); or replace the user-mode file I/O code with calls to OS-specific functions, which can be called at the kernel level (for example, the WDK ZwCreateFile(), ZwWriteFile() and ZwReadFile() functions). Please note, however, that this will diminish your driver's cross-platform portability.

Chapter 11

Improving PCI Performance

11.1. Improving PCI Performance 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:

- Implement DMA I/O mechanism for your transfers ([11.2. Performing Direct Memory Access \(DMA\)](#) or [11.3. Performing Direct Memory Access \(DMA\) transactions](#)).
- Create a Kernel PlugIn driver ([Understanding the Kernel PlugIn](#)) and move the performance-critical portions of your code to the Kernel PlugIn.

In general (this is true for most cases) I/O transfers from and to your PCI device could be of the following types: (This could vary depending on the device)

Transfer type	Speed	Hardness of implementation
Blocking	Slowest	Easy, use WDC_ReadAddrBlock() / WriteAddrBlock()
Non Blocking	Slower	Easy, use WDC_ReadAddrXXX / WDC_WriteAddrXXX
DMA with interrupts method	Fast	Harder to implement, requires using WDC_DMASGBufLock() / WDC_DMAContigBufLock() and a firm understanding of the device's specifics
DMA with polling method	Fastest	Harder to implement, requires using WDC_DMASGBufLock() / WDC_DMAContigBufLock() and a firm understanding of the device's specifics

WinDriver's enhanced support's advantage is that it saves the developer time on getting acquainted with the popular DMA IPs available on the market today, and allows them to utilize DMA without having to spend time on learning how to implement DMA for their device, or at least reducing their learning curve. For more info see [Enhanced Support for Specific Chipsets](#).

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

11.1.1. Performance Improvement Checklist

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

ISA Card - accessing an I/O-mapped range on the card

When transferring a large amount of data, use block (string) transfers and/or group several data transfer function calls into a single multi-transfer function call, as explained in [11.1.3.2. Block Transfers and Grouping Multiple Transfers \(Burst Transfer\)](#) below.

If this does not solve the problem, handle the I/O at kernel mode by writing a Kernel PlugIn driver, as explained in [Understanding the Kernel PlugIn](#) and [Creating a Kernel PlugIn Driver](#).

PCI Card - accessing an I/O-mapped range on the card

Avoid using I/O ranges in your hardware design. Use Memory mapped ranges instead as they are accessed significantly faster.

Accessing a memory-mapped range on the card

Try to access memory directly instead of using function calls, as explained in [11.2.1. Implementing Scatter/Gather DMA](#) below. When transferring large amounts of data, consider also the solution to problem #1 above.

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.

Interrupt latency - missing interrupts, receiving interrupts too late

Handle the interrupts in the kernel mode by writing a Kernel PlugIn driver, as explained in [Understanding the Kernel PlugIn](#) and [Creating a Kernel PlugIn Driver](#).

PCI target access vs. master access

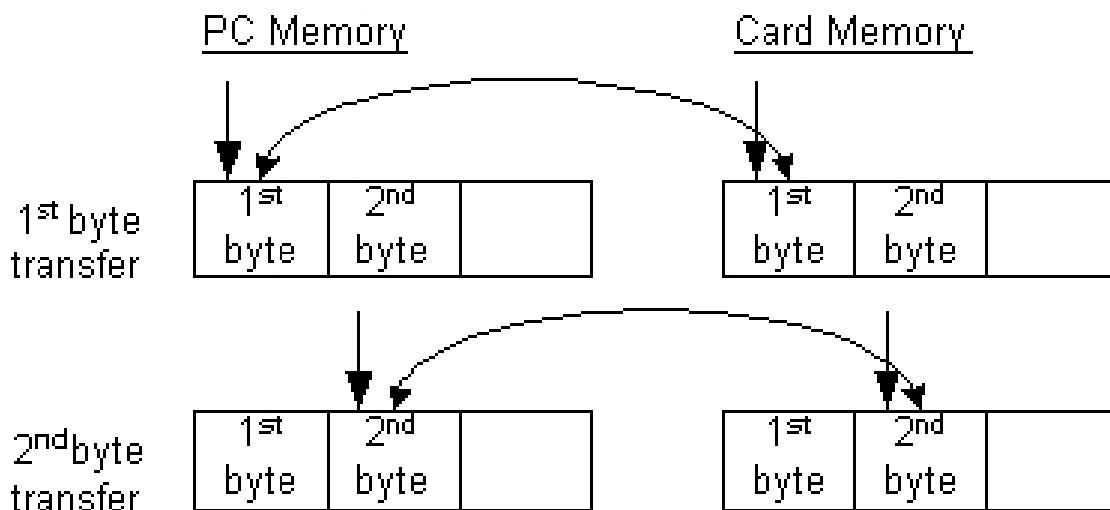
PCI target access is usually slower than PCI master access (bus-master DMA). For large data transfers, bus-master DMA access is preferable. Read [11.2. Performing Direct Memory Access \(DMA\)](#) to learn how to use WinDriver to implement bus master DMA.

11.1.2 PCI Transfers Overview

There are two PCI transfer options: reading/writing to FIFO (string transfers), or reading/writing to different memory blocks.

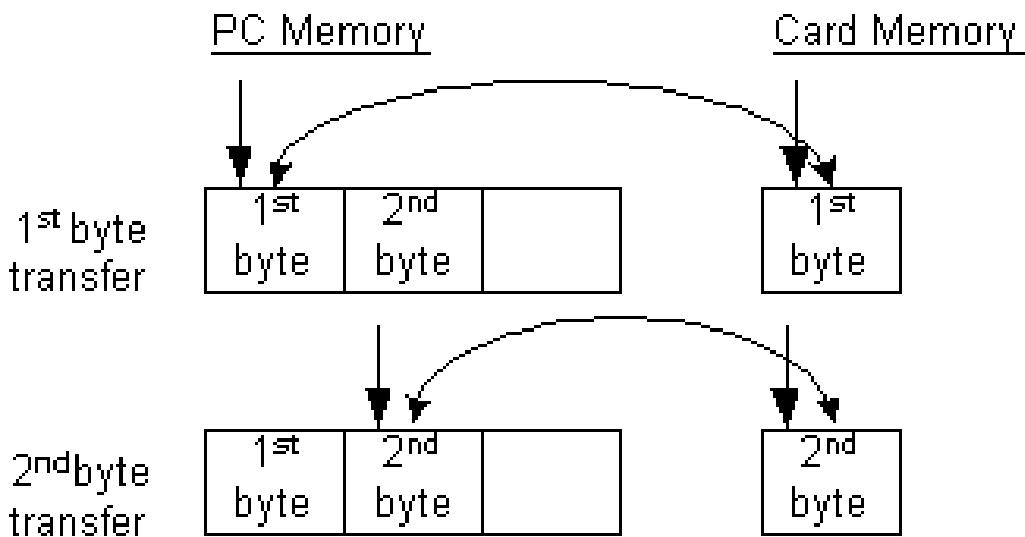
In the case of reading/writing to memory blocks, the data is transferred to/from a memory address in the PC from/to a memory address in the card, and then both the card's memory address and the PC's memory address are incremented for the next transfer. This way, the data is transferred between the address in the PC and the same relative address in the card.

PCI Transfer



In the case of reading/writing to FIFO, the data is transferred to/from a memory address in the PC from/to a single address in the card, and then only the PC's memory address is incremented for the next transfer. This way the data is transferred between incremented memory addresses in the PC and the same FIFO in the card's memory.

PCI String Transfer



The `WD_TRANSFER` structure includes an automatic increment flag, called [fAutoinc] (`WD_TRANSFER_fAutoinc`). When defined as TRUE, the I/O or memory address is incremented for transfer to/from FIFO (string transfers), when defined as FALSE, all data is transferred to the same port/address.

For more information on PCI transfers, please refer to the description of WinDriver's `WD_Transfer()` function, which executes a read/write instruction to an I/O port or to a memory address.

11.1.3. 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, because WinDriver enables you to access memory-mapped regions directly from the user mode, without the need for a function call, as explained in [11.1.3.1. Using Direct Access to Memory-Mapped Regions](#).

In addition, the WinDriver APIs enable you to improve the performance of your I/O and memory data transfers by using block (string) transfers and by grouping several data transfers into a single function call, as explained in [11.1.3.2. Block Transfers and Grouping Multiple Transfers \(Burst Transfer\)](#).

11.1.3.1. Using Direct Access to Memory-Mapped Regions

When registering a PCI card, using `WDC_PciDeviceOpen()` / `WDC_IsaDeviceOpen()` or the low-level `WD_CardRegister()` function, WinDriver returns both user-mode and kernel-mode mappings of the card's physical memory regions. These addresses can then be used to access the memory regions on the card directly, either from the user mode or from the kernel mode (respectively), thus eliminating the context switches between the user and kernel modes and the function calls overhead for accessing the memory.

The `WDC_MEM_DIRECT_ADDR` macro provides the relevant direct memory access base address - user-mode mapping when called from the user-mode / kernel-mode mapping when called from a Kernel PlugIn driver (see [Understanding the Kernel PlugIn](#)) - for a given memory address region on the card. You can then pass the mapped base address to the `WDC_ReadMem8()` / `WDC_ReadMem16()` / `WDC_ReadMem32()` / `WDC_ReadMem64()` and `WDC_WriteMem8()` / `WDC_WriteMem16()` / `WDC_WriteMem32()` / `WDC_WriteMem64()` macros, along with the desired offset within the selected memory region, to directly access a specific memory address on the card, either from the user mode or in the kernel. In addition, all the `WDC_ReadAddr8()` / `WDC_ReadAddr16()` / `WDC_ReadAddr32()` / `WDC_ReadAddr64()` and `WDC_WriteAddr8()` / `WDC_WriteAddr16()` / `WDC_WriteAddr32()` / `WDC_WriteAddr64()` functions - with the exception of `WDC_ReadAddrBlock()` and `WDC_WriteAddrBlock()` - access memory addresses directly, using the correct mapping, based on the calling context (user mode/kernel mode).

When using the low-level `WD_xxx()` APIs, the user-mode and kernel-mode mappings of the card's physical memory regions are returned by `WD_CardRegister()` within the `pTransAddr` and `pUserDirectAddr` fields of the `pCard->Reg->Card.Item[i]` card resource item structures. The `pTransAddr` result should be used as a base address

in calls to [WD_Transfer\(\)](#) or [WD_MultiTransfer\(\)](#) or when accessing memory directly from a Kernel Plugin driver ([Understanding the Kernel Plugin](#)).

To access the memory directly from your user-mode process, use pUserDirectAddr as a regular pointer. Whatever the method you select to access the memory on your card, 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 a 2-byte boundary */
UINT32 buf[len]; /* for DWORD transfers - aligned on a 4-byte boundary */
UINT64 buf[len]; /* for QWORD transfers - aligned on a 8-byte boundary */
```

11.1.3.2. Block Transfers and Grouping Multiple Transfers (Burst Transfer)

The main methods for transferring large amounts of data between PCI-device memory addresses and a host machine's random-access memory (RAM) are block transfers (which may or may not result in PCI burst transfers), and bus-master DMA.

Block transfers are easier to implement, but DMA is much more effective and reliable for transferring large amounts of data, as explained in [11.2. Performing Direct Memory Access \(DMA\)](#).

You can use the WinDriver [WDC_ReadAddrBlock\(\)](#) and [WDC_WriteAddrBlock\(\)](#) functions (or the low-level [WD_Transfer\(\)](#) function with a string command) to perform block (string) transfers - i.e., transfer blocks of data from the device memory (read) or to the device memory (write). You can use also [WDC_MultiTransfer\(\)](#) (or the low-level [WD_MultiTransfer\(\)](#) function) to group multiple block transfers into a single function call. This is more efficient than performing multiple single transfers. The WinDriver block-transfer functions use assembler string instructions (such as REP MOVSD, or a 64-bit MMX instruction for 64-bit transfers) to move a block of memory between PCI-mapped memory on the device and the host's RAM. From a software perspective, this is the most that can be done to attempt to initiate PCI burst transfers.

The hardware uses PCI burst mode to perform burst transfers - i.e., transfer the data in "bursts" of block reads/writes, resulting in a small performance improvement compared to the alternative of single WORD transfers. Some host controllers implement burst transfers by grouping access to successive PCI addresses into PCI bursts.

The host-side software has no way to control whether a target PCI transfer is issued as a burst transfer. The most the host can do is initiate transfers using assembler string instructions - as done by the WinDriver block-transfer APIs - but there's no guarantee that this will translate into burst transfers, as this is entirely up to the hardware. Most PCI host controllers support PCI burst mode for write transfers. It is generally less common to find similar burst-mode support for PCI readtransfers.

To sum it up, to transfer large amounts of data to/from memory addresses or I/O addresses (which by definition cannot be accessed directly, as opposed to memory addresses), use the following methods to improve performance by reducing the function calls overhead and context switches between the user and kernel modes:

- Perform block (string) transfers using [WDC_ReadAddrBlock\(\)](#) / [WDC_WriteAddrBlock\(\)](#) , or the low-level [WD_Transfer\(\)](#) function.
- Group several transfers into a single function call, using [WDC_MultiTransfer\(\)](#) or the low-level [WD_MultiTransfer\(\)](#) function.

11.1.3.3. Performing 64-Bit Data Transfers

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

WinDriver supports 64-bit PCI data transfers on the supported 64-bit platforms, as well as on Windows and Linux 32-bit x86 platforms. If your PCI hardware (card and bus) is 64-bit, the ability to perform 64-bit data transfers on 32-bit platforms will enable you to utilize your hardware's broader bandwidth, even if your host operating system is only 32-bit.

However, note that:

- The ability to perform actual 64-bit transfers requires that such transfers be supported by the hardware - including the CPU, the PCI card, the PCI host controller, and the PCI bridge - and it can be affected by any of these components or their specific combination.
- The conventional wisdom among hardware engineers is that performing two 32-bit DWORD transfers is more efficient than performing a single 64-bit QWORD transfer; the reason is that the 64-bit transfer requires an additional CPU cycle to negotiate a 64-bit transfer mode, and this cycle can be used, instead, to perform a second 32-bit transfer. Therefore, performing 64-bit transfers is generally more advisable if you wish to transfer more than 64 bits of data in a single burst.

This innovative technology makes possible data transfer rates previously unattainable on 32-bit platforms. Drivers developed using WinDriver will attain significantly better performance results than drivers written with the WDK 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 will achieve far better performance than 32-bit data transfer normally allows.

You can perform 64-bit data transfers using any of the following methods:

- Call [WDC_ReadAddr64\(\)](#) or [WDC_WriteAddr64\(\)](#) .
- Call [WDC_ReadAddrBlock\(\)](#) with an access mode of `WDC_SIZE_64` .
- Call [WDC_MultiTransfer\(\)](#) or the low-level [WD_Transfer\(\)](#) or [WD_MultiTransfer\(\)](#) functions with QWORD read/write transfer commands (see the documentation of these functions for details).

You can also perform 64-bit transfers to/from the PCI configuration space using [WDC_PciReadCfg64\(\)](#) and [WDC_PciReadCfgBySlot64\(\)](#) / [WDC_PciWriteCfgBySlot64\(\)](#) .

The best way to improve the performance of large PCI memory data transfers is by using bus-master direct memory access (DMA), and not by performing block transfers (which as explained above, may or may not result in PCI burst transfers).

Most PCI architectures today provide DMA capability, which enables data to be transferred directly between memory-mapped addresses on the PCI device and the host's RAM, freeing the CPU from involvement in the data transfer and thus improving the host's performance.

DMA data-buffer sizes are limited only by the size of the host's RAM and the available memory.

11.2. 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 and PCIe - 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\(\)](#), and [WDC_DMABufUnlock\(\)](#).

The following sections include code samples that demonstrate how to use WinDriver to implement Scatter/Gather DMA and contiguous-buffer DMA, and an explanation on how to preallocate contiguous DMA buffers on Windows.

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.

11.2.1. Implementing Scatter/Gather DMA

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

11.2.1.1. C Example

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

```
BOOL DMARoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMABufSize,
UINT32 u32LocalAddr, BOOL fPolling, BOOL fToDev)
{
    PVOID pBuf;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;
    /* Allocate a user-mode buffer for Scatter/Gather DMA */
    pBuf = malloc(dwDMABufSize);
    if (!pBuf)
        return FALSE;
    /* Lock the DMA buffer and program the DMA controller */
    if (!DMAOpen(hDev, pBuf, u32LocalAddr, dwDMABufSize, fToDev, &pDma))
        goto Exit;
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
            goto Exit; /* Failed enabling DMA interrupts */
    }
    /* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
    WDC_DMASyncCpu(pDma);
    /* Start DMA - write to the device to initiate the DMA transfer */
    MyDMAStart(hDev, pDma);
    /* Wait for the DMA transfer to complete */
    MyDMAWaitForCompletion(hDev, pDma, fPolling);
    /* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
    WDC_DMASyncIo(pDma);
    fRet = TRUE;
Exit:
    DMAClose(hDev, pDma, fPolling);
    free(pBuf);
    return fRet;
}
/* DMAOpen: Locks a Scatter/Gather DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID pBuf, UINT32 u32LocalAddr, DWORD dwDMABufSize, BOOL fToDev,
            WD_DMA **ppDma)
{
    DWORD dwStatus, i;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;
    /* 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%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }
    /* Program the device's DMA registers for each physical page */
    MyDMAProgram((*ppDma)->Page, (*ppDma)->dwPages, fToDev, u32LocalAddr);
    return TRUE;
```

```

}

/* DMAClose: Unlocks a previously locked Scatter/Gather DMA buffer */
void DMAClose(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);
    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);
}

```

11.2.1.2. C# Example

A more detailed example, which is specific to the enhanced support for PLX chipsets can be found in the `WinDriver/samples/c/plx/dotnet/lib/` library and `WinDriver/samples/c/plx/dotnet/diag/diagnostics library`.

```

bool DMARoutine(PCI_Device dev, DWORD dwDMABufSize, UINT32 u32LocalAddr, bool fPolling, bool fToDev,
                IntHandler MyDmaIntHandler)
{
    bool fRet = false;
    IntPtr pBuf = IntPtr.Zero;
    Log log = new Log(new Log.TRACE_LOG(TraceLog),
                      new Log.ERR_LOG(ErrLog));
    WD_DMA wdDma = new WD_DMA();
    DmaBuffer dmaBuf = new DmaBufferSG(dev, log);
    /* Allocate user mode buffer for Scatter/Gather DMA */
    pBuf = Marshal.AllocHGlobal((int)dwDMABufSize);
    if (pBuf == IntPtr.Zero)
    {
        fRet = false;
        goto Exit;
    }
    /* Lock the DMA buffer and program the DMA controller */
    if (!DMAOpen(ref dmaBuf, pBuf, u32LocalAddr, dwDMABufSize, fToDev, ref wdDma))
    {
        fRet = false;
        goto Exit;
    }
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(dev, MyDmaIntHandler, ref wdDma))
        {
            fRet = false;
            goto Exit; /* Failed enabling DMA interrupts */
        }
    }
    /* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
    wdc_lib_decl.WDC_DMASyncCpu(dmaBuf.pWdDma);
    /* Start DMA - write to the device to initiate the DMA transfer */
    MyDmaStart(dev, ref wdDma);
    /* Wait for the DMA transfer to complete */
    MyDMAWaitForCompletion(dev, ref wdDma, fPolling);
    /* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
    wdc_lib_decl.WDC_DMASyncIo(dmaBuf.pWdDma);
    fRet = true;
Exit:
    DMAClose(dev, dmaBuf, fPolling);
    if (pBuf != IntPtr.Zero)
        Marshal.FreeHGlobal(pBuf);
    return fRet;
}
/* OpenDMA: Locks a Scatter/Gather DMA buffer */
public bool DMAOpen(ref DmaBuffer dmaBuf, IntPtr pBuf, UINT32 u32LocalAddr, DWORD bufSize, bool fToDev, ref
                    WD_DMA wdDma)
{
    IntPtr pDma = dmaBuf.pWdDma;
    WDC_DEVICE_HANDLE hDev = dmaBuf.DeviceHandle;
    DWORD dwOptions = fToDev ? (DWORD)WD_DMA_OPTIONS.DMA_TO_DEVICE :
                           (DWORD)WD_DMA_OPTIONS.DMA_FROM_DEVICE;
    wdDma = MarshalDMA(pDma);
    /* Lock a Scatter/Gather DMA buffer */
    if (wdc_lib_decl.WDC_DMASGBufLock(hDev, pBuf, dwOptions, bufSize, ref pDma) !=
        (DWORD)wdc_err.WD_STATUS_SUCCESS)
        return false;

    /* Program the device's DMA registers for each physical page */
    MyDMAProgram(fToDev, wdDma.Page[0], wdDma.dwPages, u32LocalAddr);
    return true;
}
/* DMAClose: Unlocks a previously locked Scatter/Gather DMA buffer */
public void DMAClose(PCI_Device dev, DmaBuffer dmaBuf, bool fPolling)
{

```

```

/* Disable DMA interrupts (if not polling) */
if (!fPolling)
    MyDMAInterruptDisable(dev);
/* Unlock and free the DMA buffer */
wdc_lib_decl.WDC_DMABufUnlock(dmaBuf.pWdDma);
}

```

Notice the difference between ref [WD_DMA](#) wdDma and IntPtr pWdDma. The former is used by C# user functions, while the latter is used by WDC API C functions, that can only take a pointer, not a reference.

11.2.1.3. 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:

- MyDMAProgram(): Program the device's DMA registers. Refer the device's data sheet for the details.
- MyDMAStart(): Write to the device to initiate DMA transfers.
- MyDMAInterruptEnable() and MyDMAInterruptDisable(): Use [WDC_IntEnable\(\)](#) / [WDC_IntDisable\(\)](#) (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 [11.3. Performing Direct Memory Access \(DMA\) transactions](#) for details regarding interrupt handling with WinDriver.)
- MyDMAWaitForCompletion(): Poll the device for completion or wait for "DMA DONE" interrupt.
- MyDmaIntHandler: The device's interrupt handler. IntHandler is a pointer to a function prototype of your choice.

When using the basic WD_xxx API to allocate a Scatter/Gather DMA buffer that is larger than 1MB, you need to set the [DMA_LARGE_BUFFER] ([DMA_LARGE_BUFFER](#)) flag in the call to [WD_DMALock\(\)](#) and allocate memory for the additional memory pages.

However, when using [WDC_DMASGBufLock\(\)](#) to allocate the DMA buffer, you do not need any special implementation for allocating large buffers, since the function handles this for you.

11.2.2. Implementing Contiguous-Buffer DMA

Following is a sample routine that uses WinDriver's WDC API to allocate a contiguous DMA buffer and perform bus-master DMA transfers.

For more detailed, hardware-specific, contiguous DMA examples, refer to the following enhanced-support chipset sample library files:

- PLX - WinDriver/samples/c/plx/lib/plx_lib.c and WinDriver/samples/c/plx/diag_lib/plx_diag_lib.c
(which utilizes the plx_lib.c DMA API)
- Xilinx Bus Master DMA (BMD) design - WinDriver/samples/c/xilinx/bmd_design/bmd_↔ lib.c

11.2.2.1. C Example

```

BOOL DMARoutine(WDC\_DEVICE\_HANDLE hDev, DWORD dwDMABufSize,
    UINT32 u32LocalAddr, BOOL fPolling, BOOL fToDev)
{
    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, dwDMABufSize, fToDev, &pDma))
        goto Exit;
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)

```

```

{
    if (!MyDMAInterruptEnable(hDev, MyDmaIntHandler, pDma))
        goto Exit; /* Failed enabling DMA interrupts */
}
/* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
WDC_DMASyncCpu(pDma);
/* Start DMA - write to the device to initiate the DMA transfer */
MyDMAStart(hDev, pDma);
/* Wait for the DMA transfer to complete */
MyDMAWaitForCompletion(hDev, pDma, fPolling);
/* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
WDC_DMASyncIo(pDma);
fRet = TRUE;
Exit:
    DMAClose(hDev, pDma, fPolling);
    return fRet;
}
/* DMAOpen: Allocates and locks a contiguous DMA buffer */
BOOL DMAOpen(WDC_DEVICE_HANDLE hDev, PVOID *ppBuf, UINT32 u32LocalAddr,
             DWORD dwDMABufSize, BOOL fToDev, WD_DMA **ppDma)
{
    DWORD dwStatus;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;
    /* Allocate and lock a contiguous DMA buffer */
    dwStatus = WDC_DMAContigBufLock(hDev, ppBuf, dwOptions, dwDMABufSize, ppDma);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed locking a contiguous DMA buffer. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return FALSE;
    }
    /* Program the device's DMA registers for the physical DMA page */
    MyDMAProgram((*ppDma)->Page, (*ppDma)->dwPages, fToDev, u32LocalAddr);
    return TRUE;
}
/* DMAClose: Frees a previously allocated contiguous DMA buffer */
void DMAClose(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);
    /* Unlock and free the DMA buffer */
    WDC_DMABufUnlock(pDma);
}

```

11.2.2. C# Example

```

bool DMARoutine(PCI_Device dev, DWORD dwDMABufSize, UINT32 u32LocalAddr,
                bool fPolling, bool fToDev, IntHandler MyDmaIntHandler)
{
    bool fRet = false;
    IntPtr pBuf = IntPtr.Zero;
    Log log = new Log(new Log.TRACE_LOG(TraceLog),
                      new Log.ERR_LOG(ErrLog));
    WD_DMA wdDma = new WD_DMA();
    DmaBuffer dmaBuf = new DmaBufferContig(dev, log);
    /* Lock the DMA buffer and program the DMA controller */
    if (!DMAOpen(ref dmaBuf, pBuf, u32LocalAddr, dwDMABufSize,
                fToDev, ref wdDma))
    {
        fRet = false;
        goto Exit;
    }
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        if (!MyDMAInterruptEnable(dev, MyDmaIntHandler, ref wdDma))
        {
            fRet = false;
            goto Exit; /* Failed enabling DMA interrupts */
        }
    }
    /* Flush the CPU caches (see documentation of WDC_DMASyncCpu()) */
    wdc_lib_decl.WDC_DMASyncCpu(dmaBuf.pWdDma);
    /* Start DMA - write to the device to initiate the DMA transfer */
    MyDMAStart(dev, ref wdDma);
    /* Wait for the DMA transfer to complete */
    MyDMAWaitForCompletion(dev, ref wdDma, fPolling);
    /* Flush the I/O caches (see documentation of WDC_DMASyncIo()) */
    wdc_lib_decl.WDC_DMASyncIo(dmaBuf.pWdDma);
    fRet = true;
Exit:
    DMAClose(dev, dmaBuf, fPolling);
    return fRet;
}

```

```

/* OpenDMA: Locks a Contiguous DMA buffer */
public bool DMAOpen(ref DmaBuffer dmaBuf, IntPtr pBuf, uint u32LocalAddr,
    DWORD bufSize, bool fToDev, ref WD_DMA wdDma)
{
    IntPtr pDma = dmaBuf.pWdDma;
    WDC_DEVICE_HANDLE hDev = dmaBuf.DeviceHandle;
    DWORD dwOptions = fToDev ? (DWORD)WD_DMA_OPTIONS.DMA_TO_DEVICE :
        (DWORD)WD_DMA_OPTIONS.DMA_FROM_DEVICE;
    wdDma = MarshalDMA(pDma);
    /* Lock a Scatter/Gather DMA buffer */
    if (wdc_lib_decl.WDC_DMAContigBufLock(hDev, ref pBuf, dwOptions,
        bufSize, ref pDma) != (DWORD)wdc_err.WD_STATUS_SUCCESS)
        return false;
    /* Program the device's DMA registers for each physical page */
    MyDMAProgram(fToDev, wdDma.Page[0], wdDma.dwPages, u32LocalAddr);
    return true;
}
/* DMAClose: Unlocks a previously locked Scatter/Gather DMA buffer */
public void DMAClose(PCI_Device dev, DmaBuffer dmaBuf, bool fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(dev);
    /* Unlock and free the DMA buffer */
    wdc_lib_decl.WDC_DMABufUnlock(dmaBuf.pWdDma);
}

```

11.2.2.3. 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:

- MyDMAProgram(): Program the device's DMA registers. Refer the device's data sheet for the details.
- MyDMAStart(): Write to the device to initiate DMA transfers.
- MyDMAInterruptEnable() and MyDMAInterruptDisable(): Use [WDC_IntEnable\(\)](#) / [WDC_IntDisable\(\)](#) (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 [11.3. Performing Direct Memory Access \(DMA\) transactions](#) for details regarding interrupt handling with WinDriver.)
- MyDMAWaitForCompletion(): Poll the device for completion or wait for "DMA DONE" interrupt.
- MyDmaIntHandler: The device's interrupt handler. IntHandler is a pointer to a function prototype of your choice.

11.2.2.4. Preallocating Contiguous DMA Buffers on Windows

WinDriver doesn't limit the size of the DMA buffer that can be allocated using its DMA APIs. However, the success of the DMA allocation is dependent on the amount of available system resources at the time of the allocation. Therefore, the earlier you try to allocate the buffer, the better your chances of succeeding.

WinDriver for Windows allows you to configure your device INF file to preallocate contiguous DMA buffers at boot time, thus increasing the odds that the allocation(s) will succeed. You may preallocate a maximum of 512 buffers: - 256 host-to-device buffers and/or 256 device-to-host buffers.

There are 2 ways to preallocate contiguous DMA buffers on Windows: directly from the DriverWizard, or manually via editing the INF file.

Directly from DriverWizard:

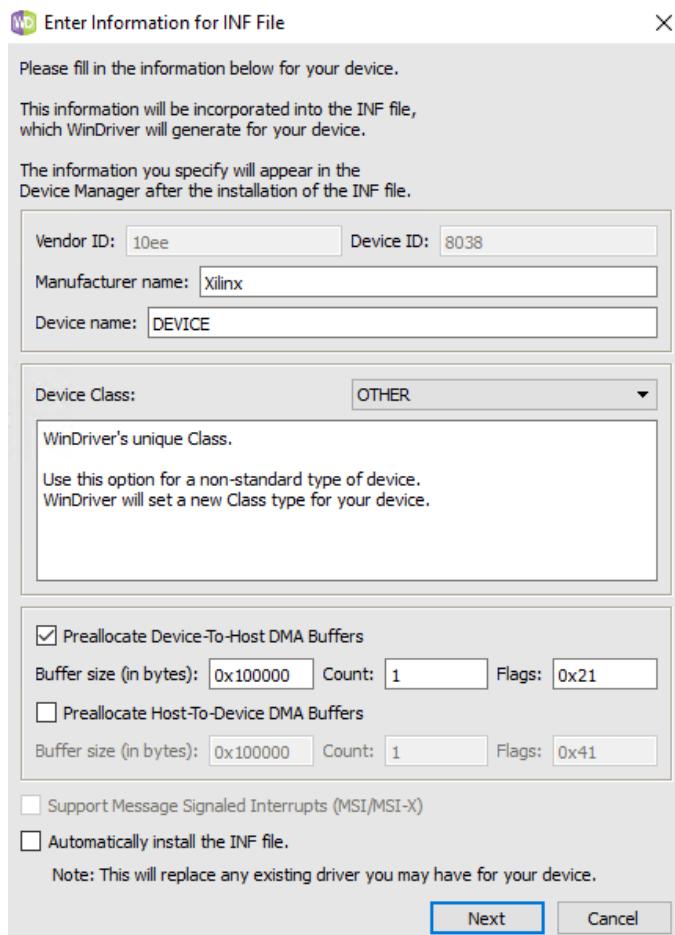
- In DriverWizard, start a new project, select a device from the list and click **Generate .INF file**.
- Check Preallocate Host-To-Device DMA Buffers and/or Preallocate Device-To-Host DMA Buffers to enable the text boxes under each checkbox.
- Adjust the Size, Count and Flags parameters as desired. The supported WinDriver DMA flags are documented in the [WD_DMA_OPTIONS] ([WD_DMA_OPTIONS](#)) enum.

- Click **Next**, and you will then be prompted to choose a filename for your .INF file. After choosing a filename, the INF file will be created and ready to use, with your desired parameters.

** Attention**

The Size and Flags fields must be hexadecimal numbers, formatted with the "0x" prefix, as shown below.

DriverWizard INF File Information



Manually by editing an existing INF file:

- Add the required configuration under the `UpdateRegistryDevice` registry key in your device INF file, as shown below.

The examples are for configuring preallocation of eight DMA buffers but you may, of-course, select to preallocate just one buffer (or none at all). To preallocate unidirectional buffers, add these lines:

```
; Host-to-device DMA buffer:
HKR,, "DmaToDeviceCount", 0x00010001, 0x04 ; Number of preallocated
; DMA_TO_DEVICE buffers
HKR,, "DmaToDeviceBytes", 0x00010001, 0x100000 ; Buffer size, in bytes
HKR,, "DmaToDeviceOptions", 0x00010001, 0x41 ; DMA flags (0x40=DMA_TO_DEVICE
; + 0x1=DMA_KERNEL_BUFFER_ALLOC

; Device-to-host DMA buffer:
HKR,, "DmaFromDeviceCount", 0x00010001, 0x04 ; Number of preallocated
; DMA_FROM_DEVICE buffers
HKR,, "DmaFromDeviceBytes", 0x00010001, 0x100000 ; Buffer size, in bytes
HKR,, "DmaFromDeviceOptions", 0x00010001, 0x21 ; DMA flags (0x20=DMA_FROM_DEVICE
; + 0x1=DMA_KERNEL_BUFFER_ALLOC)
```

- Edit the buffer sizes and add flags to the options masks in the INF file, as needed. Note, however, that the direction flags and the [DMA_KERNEL_BUFFER_ALLOC] (DMA_KERNEL_BUFFER_ALLOC) flag must be set as shown in Step 1.

The supported WinDriver DMA flags for the dwOptions field of the [WD_DMA](#) struct are documented in [WD_<- DMA_OPTIONS] ([WD_DMA_OPTIONS](#)).

DmaFromDeviceCount and DmaFromDeviceCount values are supported from version 12.4.0. If those values aren't set value of 1 will be assumed.

The Wizard-generated and relevant sample WinDriver device INF files already contain the unidirectional buffers configuration lines, so you only need to remove the comment indicator ; at the start of each line. The examples are for configuring preallocation of 8 DMA buffers (4 for each direction), but you may, of-course, select to preallocate just one buffer (or none at all, by leaving the above code commented out).

In your code, the first n calls (if you configured the INF file to preallocate n DMA buffers) to the contiguous-DMA-lock function - [WDC_DMAContigBufLock\(\)](#) - should set parameter values that match the buffer configurations in the INF file:

** Attention**

For a device-to-host buffer, the DMA-options mask parameter (dwOptions / pDma->dwOptions) should contain the same DMA flags set in the DmaFromDeviceOptions registry key value, and the buffer-size parameter (dwDMABufSize / pDma->dwBytes) should be set to the value of the DmaFromDeviceBytes registry key value. For a host-to-device buffer, the DMA-options mask parameter (dwOptions / pDma->dwOptions) should contain the same flags set in the DmaToDeviceOptions registry key value, and the buffer-size parameter (dwDMABufSize / pDma->dwBytes) should be set to the value of the DmaToDeviceBytes registry key value.

In calls to [WDC_DMAContigBufLock\(\)](#), the DMA configuration information is provided via dedicated function parameters - dwDMABufSize and dwOptions. In calls to [WD_DMALock\(\)](#), the information is provided within the fields of the [WD_DMA](#) struct pointed to by the pDma parameter - pDma->dwBytes and pDma->dwOptions.

When using [WDC_DMAContigBufLock\(\)](#) you don't need to explicitly set the [DMA_KERNEL_BUFFER_<- ALLOC] ([DMA_KERNEL_BUFFER_ALLOC](#)) flag (which must be set in the INF-file configuration) because the function sets this flag automatically.

When using the low-level WinDriver [WD_DMALock\(\)](#) function, the DMA options are set in the function's pDma->dwOptions parameter - which must also include the [DMA_KERNEL_BUFFER_ALLOC] ([DMA_KERNEL_BUFFER_ALLOC](#)) flag - and the buffer size is set in the pDma->dwBytes parameter.

If the buffer preallocation fails due to insufficient resources, you may need to increase the size of the non-paged pool (from which the memory is allocated).

11.3. Performing Direct Memory Access (DMA) transactions

This section describes how to use WinDriver to implement bus-master Direct Memory Access (DMA) transactions 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 and PCIe - 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.

To understand the use of WinDriver's DMA transaction API, you must be familiar with the following concepts:

DMA transaction

A DMA transaction is a complete I/O operation, such as a single read or write request from an application.

DMA transfer

A DMA transfer is a single hardware operation that transfers data from computer memory to a device or from the device to computer memory.

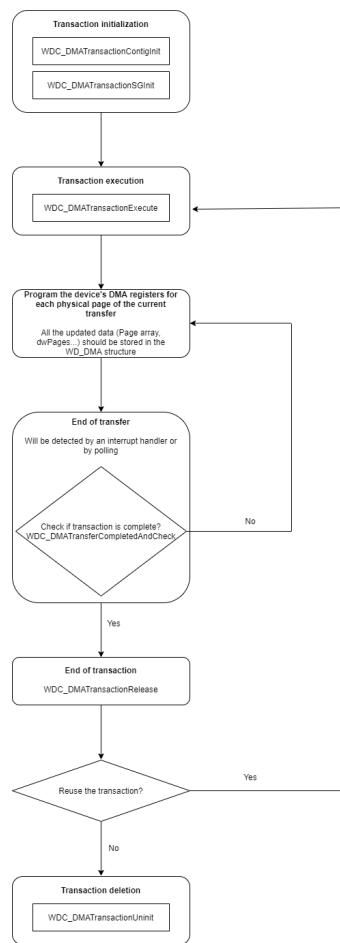
A single DMA transaction always consists of at least one DMA transfer, but a transaction can consist of many transfers.

A DMA transaction buffer can be allocated in two ways:

- **Contiguous buffer** - A contiguous block of memory is allocated.
- **Scatter/Gather (SG)** - 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 an API for implementing both Contiguous-Buffer DMA transactions and Scatter/Gather DMA transactions (if supported by the hardware) - see the description of [WDC_DMATransactionContigInit\(\)](#), [WDC_DMATransactionSGInit\(\)](#), [WDC_DMATransferCompletedAndCheck\(\)](#), [WDC_DMATransactionRelease\(\)](#) and [WDC_DMATransactionUninit\(\)](#).

DMA transaction diagram



The following sections include code samples that demonstrate how to use WinDriver to implement a Scatter/Gather DMA transaction (see [11.2.1. Implementing Scatter/Gather DMA](#)) and a Contiguous-Buffer DMA transaction (see [11.2.2. Implementing Contiguous-Buffer DMA](#)).

The sample routines demonstrate using either an interrupt mechanism or a polling mechanism to determine DMA completion.

11.3.1. Implementing Scatter/Gather DMA transactions

Following is a sample routine that uses WinDriver's WDC API to initialize a Scatter/Gather DMA transaction buffer and perform bus-master DMA transfers.

For more detailed, hardware-specific, Scatter Gather DMA transaction examples, refer to the following enhanced-support chipset ([Enhanced Support for Specific Chipsets](#)) sample library files:

- PLX - WinDriver/samples/c/plx/lib/plx_lib.c, WinDriver/samples/c/plx/diag←_lib/plx_diag_lib.c (which utilizes the plx_lib.c DMA API) and WinDriver/samples/c/plx/9656/p9656←_diag.c
- XDMA design - WinDriver/samples/c/xilinx/xdma/xdma_lib.c

11.3.1.1. C Example

```
#define MAX_TRANSFER_SIZE 0xFFFF /* According to the device limits */
typedef struct {
    UINT32 u32PADDR; /* PCI address */
    UINT32 u32LADR; /* Local address */
    UINT32 u32SIZ; /* Transfer size */
    UINT32 u32DPR; /* Next descriptor pointer */
} MY_DMA_TRANSFER_ELEMENT; /* DMA transfer element */
BOOL DMATransactionRoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMABufSize,
                           UINT32 u32LocalAddr, BOOL fPolling, BOOL fToDev)
{
    PVOID pBuf;
    WD_DMA *pDma = NULL;
    BOOL fRet = FALSE;
    /* Allocate a user-mode buffer for a Scatter/Gather DMA transaction */
    pBuf = malloc(dwDMABufSize);
    if (!pBuf)
        return FALSE;
    /* Initialize the DMA transaction */
    if (!DMATransactionInit(hDev, pBuf, u32LocalAddr, dwDMABufSize, fToDev, &pDma,
                           fPolling))
    {
        goto Exit;
    }
    if (!DMATransactionExecute(hDev, pDma, fPolling))
    {
        goto Exit;
    }
    /* Wait for the DMA transaction to complete */
    MyDMAWaitForCompletion(hDev, pDma, fPolling);
    if (!DMATransactionRelease(hDev, pDma))
    {
        goto Exit;
    }
    /* The DMA transaction can be reused as many times as you like
     * Notice: after each call to DMATransactionExecute
     * (WDC_DMATransactionExecute) there must be a call to
     * DMATransactionRelease (WDC_DMATransactionRelease) */
    fRet = TRUE;
Exit:
    DMAClose(hDev, pDma, fPolling);
    free(pBuf);
    return fRet;
}
/* DMATransactionInit: Initializes a Scatter/Gather DMA transaction buffer */
BOOL DMATransactionInit(WDC_DEVICE_HANDLE hDev, PVOID pBuf, UINT32 u32LocalAddr,
                       DWORD dwDMABufSize, BOOL fToDev, WD_DMA **ppDma, BOOL fPolling)
{
    DWORD dwStatus, dwNumCmds, i;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;
    WD_TRANSFER *pTrans;
    WDC_INTERRUPT_PARAMS interruptsParams;
    BOOL fRet = FALSE;
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        pTrans = GetMyTransCmds(&dwNumCmds);
        interruptsParams.pTransCmds = pTrans;
        interruptsParams.dwNumCmds = dwNumCmds;
        interruptsParams.dwOptions = INTERRUPT_CMD_COPY;
        interruptsParams.funcIntHandler = MyDmaIntHandler;
        interruptsParams.pData = hDev;
        interruptsParams.fUseKP = FALSE;
    }
    /* Initialize a Scatter/Gather DMA transaction buffer */
    dwStatus = WDC_DMATransactionSGInit(hDev, pBuf, dwOptions, dwDMABufSize,
                                         ppDma, &interruptsParams, MAX_TRANSFER_SIZE,
                                         sizeof(MY_DMA_TRANSFER_ELEMENT));
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize a Scatter/Gather DMA transaction buffer. Error "
               "0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    fRet = TRUE;
Exit:
    return fRet;
}
```

```

}

/* DMATransactionExecute: Executes the DMA transaction */
BOOL DMATransactionExecute(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    BOOL fRet = FALSE;
    DWORD dwPage;
    UINT64 u64Offset;
    dwStatus = WDC_DMATransactionExecute(pDma, DMATransactionProgram, hDev);
    if (dwStatus != WD_STATUS_SUCCESS)
    {
        printf("Failed to execute DMA transaction. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    fRet = TRUE;
Exit:
    return fRet;
}
/* DMATransactionProgram: Programs the device's DMA registers and starts the
 * DMA */
void DMATransactionProgram(WDC_DEVICE_HANDLE hDev)
{
    PDEV_CTX pDevCtx = (PDEV_CTX)WDC_GetDevContext(hDev);
    WD_DMA *pDma = pDevCtx->pDma;
    BOOL fToDev = pDevCtx->fToDev;
    DWORD dwPage;
    UINT64 u64Offset;
    /* The value of u64Offset is the required offset on the device */
    u64Offset = pDevCtx->u32LocalAddr + pDma->dwBytesTransferred;
    for (dwPage = 0; dwPage < pDma->dwPages; dwPage++)
    {
        /* Use pDma->Page array, u64Offset, fToDev, ...
         * to program the device's DMA registers for each physical page */
        u64Offset += pDma->Page[dwPage].dwBytes;
    }
    /* Enable interrupts on hardware */
    MyDMAInterruptHardwareEnable(hDev);
    /* Start DMA - write to the device to start the DMA transfer */
    MyDMAStart(hDev, pDma);
}
/* MyDmaIntHandler: Interrupt handler */
void DLLCALLCONV MyDmaIntHandler(PVOID pData)
{
    PDEV_CTX pDevCtx = (PDEV_CTX)WDC_GetDevContext(hDev);
    WD_DMA *pDma = pDevCtx->pDma;
    DWORD dwStatus;
    dwStatus = WDC_DMATransferCompletedAndCheck(pDma, TRUE);
    if (dwStatus == WD_STATUS_SUCCESS)
    {
        /* DMA transaction completed */
    }
    else if (dwStatus != WD_MORE_PROCESSING_REQUIRED)
    {
        /* DMA transfer failed */
    }
    else /* dwStatus == WD_MORE_PROCESSING_REQUIRED */
    {
        /* DMA transfer completed (but the transaction is not completed) */
        /* If the fRunCallback parameter given to
         * WDC_DMATransferCompletedAndCheck() is TRUE, then the
         * DMATransactionProgram function is automatically called
         * synchronously by WDC_DMATransferCompletedAndCheck */
    }
}
/* DMATransactionRelease: Releases the DMA transaction */
BOOL DMATransactionRelease(WD_DMA *pDma)
{
    DWORD dwStatus = WDC_DMATransactionRelease(pDma);
    return (dwStatus == WD_STATUS_SUCCESS) ? TRUE : FALSE;
}
/* DMAClose: Deletes a previously initiated Scatter/Gather DMA transaction */
void DMAClose(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);
    WDC_DMATransactionUninit(pDma);
}

```

11.3.1.2. C# Example

A detailed example, which is specific to the enhanced support for PLX chipsets can be found in the Win↔Driver/samples/c/plx/dotnet/lib/ library and WinDriver/samples/c/plx/dotnet/diag/

diagnostics library.

11.3.1.3. What Should You Implement?

In the code sample above, it is up to you to implement the following MyDMAxxx() routines and MY_DMA_TRANSFER_ELEMENT structure according to your device's specification and your needs:

- MY_DMA_TRANSFER_ELEMENT: A DMA transfer element structure according to your device's specification.
- MyDMAStart(): Write to the device to initiate DMA transfers.
- MyDMAInterruptHardwareEnable(): Write/read the relevant register(s) on the device in order to physically enable the hardware DMA interrupts (see [Section 10.4](#) for details regarding interrupt handling with WinDriver).
- MyDMAInterruptDisable(): Use [WDC_IntDisable\(\)](#) to disable the software interrupts and write/read the relevant register(s) on the device in order to physically disable the hardware DMA interrupts (see [Section 11.4](#) for details regarding interrupt handling with WinDriver).
- MyDMAWaitForCompletion(): Poll the device for completion or wait for a "DMA transfer done" interrupt.
- MyDmaIntHandler: The device's interrupt handler. IntHandler is a pointer to a function prototype of your choice.

11.3.2. Implementing Contiguous-Buffer DMA transactions

Following is a sample routine that uses WinDriver's WDC API to initialize a Contiguous- Buffer DMA transaction and perform bus-master DMA transfers.

For more detailed, hardware-specific, contiguous DMA examples, refer to the following enhanced-support chipset sample library files:

- PLX - WinDriver/samples/c/plx/lib/plx_lib.c and WinDriver/samples/c/plx/diag-lib/plx_diag_lib.c (which utilizes the plx_lib.c DMA API)
- XDMA design - WinDriver/samples/c/xilinx/xdma/xdma_lib.c

11.3.2.1. C Example

```
#define DMA_ALIGNMENT_REQUIREMENT 0xFF /* According to the device requirements */
BOOLEAN DMATransactionRoutine(WDC_DEVICE_HANDLE hDev, DWORD dwDMABufSize,
UINT32 u32LocalAddr, BOOLEAN fPolling, BOOLEAN fToDev)
{
    PVOID pBuf = NULL;
    WD_DMA *pDma = NULL;
    BOOLEAN fRet = FALSE;
    /* Initialize the DMA transaction */
    if (!DMATransactionInit(hDev, &pBuf, u32LocalAddr, dwDMABufSize, fToDev,
        &pDma, fPolling))
    {
        goto Exit;
    }
    if (!DMATransactionExecute(hDev, pDma, fPolling))
    {
        goto Exit;
    }
    /* Wait for the DMA transaction to complete */
    MyDMAWaitForCompletion(hDev, pDma, fPolling);
    if (!DMATransactionRelease(hDev, pDma))
    {
        goto Exit;
    }
    /* The DMA transaction can be reused as many times as you like
     * Notice: after each call to DMATransactionExecute
     * (WDC_DMATransactionExecute) there must be a call to
     * DMATransactionRelease (WDC_DMATransactionRelease) */
    fRet = TRUE;
Exit:
```

```
DMAClose(hDev, pDma, fPolling);
    free(pBuf);
    return fRet;
}
/* DMATransactionInit: Initializes a Contiguous-Buffer DMA transaction buffer */
BOOL DMATransactionInit(WDC_DEVICE_HANDLE hDev, PVOID *ppBuf, UINT32 u32LocalAddr,
    DWORD dwDMABufSize, BOOL fToDev, WD_DMA **pDma, BOOL fPolling)
{
    DWORD dwStatus, dwNumCmds, i;
    DWORD dwOptions = fToDev ? DMA_TO_DEVICE : DMA_FROM_DEVICE;
    WD_TRANSFER *pTrans;
    WDC_INTERRUPT_PARAMS interruptsParams;
    BOOL fRet = FALSE;
    /* Enable DMA interrupts (if not polling) */
    if (!fPolling)
    {
        pTrans = GetMyTransCmds(&dwNumCmds);
        interruptsParams.pTransCmds = pTrans;
        interruptsParams.dwNumCmds = dwNumCmds;
        interruptsParams.dwOptions = INTERRUPT_CMD_COPY;
        interruptsParams.funcIntHandler = MyDmaIntHandler;
        interruptsParams.pData = hDev;
        interruptsParams.fUseKP = FALSE;
    }
    /* Initialize a Contiguous-Buffer DMA transaction buffer */
    dwStatus = WDC_DMATransactionConfigInit(hDev, ppBuf, dwOptions, dwDMABufSize,
        ppDma, &interruptsParams, DMA_ALIGNMENT_REQUIREMENT);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize a Contiguous-Buffer DMA transaction buffer. Error "
            "0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    fRet = TRUE;
    Exit:
    return fRet;
}
/* DMATransactionExecute: Executes the DMA transaction */
BOOL DMATransactionExecute(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    BOOL fRet = FALSE;
    DWORD dwPage;
    UINT64 u64Offset;
    dwStatus = WDC_DMATransactionExecute(pDma, DMATransactionProgram, hDev);
    if (dwStatus != WD_STATUS_SUCCESS)
    {
        printf("Failed to execute DMA transaction. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    fRet = TRUE;
    Exit:
    return fRet;
}
/* DMATransactionProgram: Programs the device's DMA registers and starts the
 * DMA */
void DMATransactionProgram(WDC_DEVICE_HANDLE hDev)
{
    PDEV_CTX pDevCtx = (PDEV_CTX)WDC_GetDevContext(hDev);
    WD_DMA *pDma = pDevCtx->pDma;
    BOOL fToDev = pDevCtx->fToDev;
    UINT64 u64Offset;
    /* The value of u64Offset is the required offset on the device */
    u64Offset = pDevCtx->u32LocalAddr + pDma->dwBytesTransferred;
    /* Use pDma->Page[0], u64Offset, fToDev,... */
    /* to program the device's DMA registers */
    /* Enable interrupts on hardware */
    MyDMAInterruptHardwareEnable(hDev);
    /* Start DMA - write to the device to start the DMA transfer */
    MyDMAStart(hDev, pDma);
}
/* MyDmaIntHandler: Interrupt handler */
void DLLCALLCONV MyDmaIntHandler(PVOID pData)
{
    PDEV_CTX pDevCtx = (PDEV_CTX)WDC_GetDevContext(hDev);
    WD_DMA *pDma = pDevCtx->pDma;
    DWORD dwStatus;
    dwStatus = WDC_DMATransferCompletedAndCheck(pDma, TRUE);
    if (dwStatus == WD_STATUS_SUCCESS)
    {
        /* DMA transaction completed */
    }
    else if (dwStatus == WD_MORE_PROCESSING_REQUIRED)
    {
        /* DMA transfer completed (but the transaction is not completed) */
        /* If the fRunCallback parameter given to
         * WDC_DMATransferCompletedAndCheck() is TRUE, then the
```

```
* DMATransactionProgram function is automatically called
 * synchronously by WDC_DMATransferCompletedAndCheck */
}

else
{
    /* DMA transfer failed */
}

}

/* DMATransactionRelease: Releases the DMA transaction */
BOOL DMATransactionRelease(WD_DMA *pDma)
{
    DWORD dwStatus = WDC_DMATransactionRelease(pDma);
    return (dwStatus == WD_STATUS_SUCCESS) ? TRUE : FALSE;
}

/* DMAClose: Deletes a previously initiated Contiguous-Buffer DMA transaction */
void DMAClose(WDC_DEVICE_HANDLE hDev, WD_DMA *pDma, BOOL fPolling)
{
    /* Disable DMA interrupts (if not polling) */
    if (!fPolling)
        MyDMAInterruptDisable(hDev);
    WDC_DMATransactionUninit(pDma);
}
```

11.3.2.2. C# Example

A detailed example, which is specific to the enhanced support for PLX chipsets can be found in the `WinDriver/samples/c/plx/dotnet/lib/` library and `WinDriver/samples/c/plx/dotnet/diag/` diagnostics library.

11.3.2.3 What Should You Implement?

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

- `MY_DMA_TRANSFER_ELEMENT`: DMA transfer element structure according to your device's specification.
- `MyDMAStart()`: Write to the device to initiate DMA transfers.
- `MyDMAInterruptHardwareEnable()`: Write/read the relevant register(s) on the device in order to physically enable the hardware DMA interrupts (see [Section 11.4](#) for details regarding interrupt handling with WinDriver).
- `MyDMAInterruptDisable()`: Use `WDC_IntDisable()` to disable the software interrupts and write/read the relevant register(s) on the device in order to physically disable the hardware DMA interrupts (see [Section 11.4](#) for details regarding interrupt handling with WinDriver).
- `MyDMAWaitForCompletion()`: Poll the device for completion or wait for a "DMA transfer done" interrupt.
- `MyDmaIntHandler`: The device's interrupt handler. `IntHandler` is a pointer to a function prototype of your choice.

11.4. DMA between PCI devices and NVIDIA GPUs with GPUDirect (Linux only)

11.4.1. What is GPUDirect for RDMA?

GPUDirect for RDMA is a feature available on selected NVIDIA GPUs that allows performing Direct Memory Access (DMA) between GPUs and PCI Express devices. Check out NVIDIA's website to make sure your GPU does support GPUDirect for this purpose.

NVIDIA GPUDirect RDMA is currently supported only on Linux. We strive to add support for further GPUs and OSes in the future, please contact WinDriver@jungo.com for further inquiries and suggestions.

11.4.2. System Requirements

The following system requirements:

- NVIDIA GPU that supports GPUDirect.
- PCIe device controlled by WinDriver.

11.4.3. Software Prerequisites

The following software prerequisites:

- Installed NVIDIA kernel drivers for your GPU.
- Installed CUDA version that supports GPUDirect (we have tested with version 10) along with the NVCC compiler.

11.4.4. WinDriver installation

Unpack WinDriver from the tar file.

```
cd WinDriver/redist
```

Make sure your WinDriver kernel module is linked with NVIDIA's kernel module to allow GPUDIRECT.

```
./configure --with-gpudirect-source=<>YOUR_NVIDIA_KERNEL_SOURCE_DIRECTORY>/kernel  
sudo make && sudo make install
```

11.4.5. Moving DMA from CPU to GPU

We strongly recommend making sure you have already implemented and tested a “regular” DMA routine between your device and the computer RAM before moving on to implementing a GPUDirect DMA routine.

See previous sections for more info ([11.2. Performing Direct Memory Access \(DMA\)](#) or [11.3. Performing Direct Memory Access \(DMA\) transactions](#)).

Assuming you have already implemented a DMA routine in your WinDriver-based code, perform the following steps to perform DMA to the GPU memory instead of the main memory.

11.4.6. Modify Compilation

If compiling using Make/CMake, follow this instructions and see a detailed example below:

- Change your makefile to compile your app with the CUDA compiler (nvcc) instead of gcc.
- Change your makefile to link your app with the CUDA compiler (nvcc) instead of ld.
- Add -lcuda to your linker flags(LFLAGS) in order to link with the CUDA shared libraries.
- Remove -fno-pie and -m\$ (USER_BITS) from your linker flags.

11.4.7. Modify your code

Add to your code.

```
#include <cuda.h>  
#include <cuda_runtime.h>
```

Instead of using regular `malloc()` to allocate the memory for the pBuf parameter in the function `WDC_DMASGBufLock()`, use `cudaMalloc()`.

Make sure that the dwOptions parameter of `WDC_DMASGBufLock()` contains the [DMA_GPUDIRECT] ([DMA_GPUDIRECT](#)) flag.

Add the following code to enable synchronous memory operations with your DMA buffer (`pDma->pBuf` in this example)

```
int flag = 1;
if (CUDA_SUCCESS != cuPointerSetAttribute(&flag,
    CU_POINTER_ATTRIBUTE_SYNC_MEMOPS, (CUdeviceptr)pDma->pBuf))
{
    printf("cuDeviceGet failed\n");
    return;
}
```

Use CUDA functions to access the GPU memory such as: `cudaMemcpy()`, `cudaFree()` etc. Using regular memory management functions on pointers to GPU memory might lead to crashes.

** Attention**

Calling `WDC_DMASGBufLock()` with [DMA_GPUDIRECT] (`DMA_GPUDIRECT`) flag with a buffer allocated with regular memory buffer (not allocated using `cudaMalloc()`) will result in an Internal System Error (Status 0x20000007).

11.4.8. CMake Example

If using CMake to compile your code – use the following recipe as a guide.

```
cmake_minimum_required(VERSION 3.0)
set(WD_BASEDIR ~/WinDriver) #change according to your installation path
project(my_wd_gpudirect_project C)
include(${WD_BASEDIR}/include/wd.cmake)
include_directories(
    ${WD_BASEDIR}
    ${WD_BASEDIR}/include
)
set(SRCS my_wd_gpudirect_project.c)
add_executable(my_wd_gpudirect_project ${SRCS} ${SAMPLE_SHARED_SRCS})
#link with and libwdapi1511 and libcuda
target_link_libraries(my_wd_gpudirect_project wdapi${WD_VERSION} cuda)
set_target_properties(my_wd_gpudirect_project PROPERTIES
    RUNTIME_OUTPUT_DIRECTORY "${ARCH}/"
)
#remove definitions to allow compilation with nvcc
remove_definitions("-Wno-unused-result -Wno-write-strings")
set(CMAKE_SHARED_LIBRARY_LINK_C_FLAGS "")
#change compiler to nvcc
set(CMAKE_C_COMPILER /usr/local/cuda-10.2/bin/nvcc)
#add GPUDIRECT definition to compilation
target_compile_definitions(my_wd_gpudirect_project PRIVATE GPUDIRECT)
```

11.5. FAQ

11.5.1. How do I perform system or slave DMA using WinDriver?

WinDriver supports the implementation of interrupt service routines and locking down DMA buffers into memory, giving you the physical addresses and lengths of the kernel DMA buffer.

Assuming you want to implement slave DMA for an ISA card, you will need to write the code to program the DMA controller yourself. There is no specific API to program the system DMAC on PCs, but you can use the generic WinDriver API for direct hardware access and DMA allocation from your application

(see specifically `WDC_DMAContigBufLock()`, `WDC_DMASGBufLock()`,

and the `WDC_PciReadCfg8()` / `WDC_PciReadCfg16()` / `WDC_PciReadCfg32()` / `WDC_PciReadCfg64()`

and `WDC_PciWriteCfg8()` / `WDC_PciWriteCfg16()` / `WDC_PciWriteCfg32()` / `WDC_PciWriteCfg64()` or the lower-level `WD_DMALock()` and `WD_Transfer()` APIs).

11.5.2. I have locked a memory buffer for DMA on Windows. Now, when I access this memory directly, using the user-mode pointer, it seems to be 5 times slower than accessing a “regular” memory buffer, allocated with malloc(). Why?

“Regular” memory (stack, heap, etc.) is cached by the operating system. When using WinDriver DMA APIs, the data is non-cached, in order to make it DMA-safe. Therefore, the memory access is slower. Note that this is the correct behavior for DMA.

When performing Contiguous Buffer DMA, you can set the DMA_ALLOW_CACHE flag in the dwOptions parameter of [WDC_DMAContigBufLock\(\)](#), or directly in the dwOptions field of the [WD_DMA](#) structure that is passed to [WD_DMALock\(\)](#) (when using the low-level WinDriver API), in order to allocate a cached DMA buffer. When working on Windows x86 and x86_64 platforms, it is recommended to always set this flag.

If you have allocated the memory in the user mode and then passed its address to [WDC_DMASGBufLock\(\)](#) or to the low-level [WD_DMALock\(\)](#) function in order to lock a Scatter/Gather DMA buffer, then calling [WD_DMAUnlock\(\)](#) will unlock the memory buffer and it will now function like other “regular” memory in terms of access speed.

11.5.3. My attempt to allocate and lock a 1GB DMA buffer with WinDriver on Windows fails. Is this a limitation of the operating system?

WinDriver does not impose any inherent limitation on the size of the DMA buffer that can be allocated using its DMA APIs. However, the success of the DMA allocation is dependent of the amount of available system resources at the time of the attempted allocation. Therefore, the earlier your try to allocate the buffer, the better your chances of succeeding.

For contiguous-buffer DMA allocation, there must be enough contiguous physical memory for the allocation. On Windows, WinDriver allows preallocation of DMA buffers to evade this, so please refer to the relevant chapter of the manual for more information.

When allocating a Scatter/Gather DMA buffer that is larger than 1MB, using the low-level WinDriver API, be sure to set the [DMA_LARGE_BUFFER] ([DMA_LARGE_BUFFER](#)) flag in the dwOptions field of the [WD_DMA\(\)](#) structure that is passed to [WD_DMALock\(\)](#). (When using the high-level [WDC_DMASGBufLock\(\)](#) function, this flag is already handled internally by the function.)

The DMA buffer allocated by WinDriver uses page-locked memory, to ensure a safe DMA operation. This memory is allocated from Windows' non-paged kernel pool of memory. The size of this pool is fixed at boot time by a Registry setting. You can increase the allocated memory by increasing the value of the NonPagedPoolSize Registry entry, found under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management`.

Sometimes, there is enough contiguous memory, but there are not enough page table entries to map the memory. Even after increasing the value of the relevant Registry entries, the memory allocation might still fail, specifically when trying to allocate a very large buffer (such as 1GB). The solution in this case is to try decreasing the size of the buffer you are trying to lock, until you succeed.

** Attention**

Please note that the WinDriver DMA allocation APIs (([WDC_DMAContigBufLock\(\)](#) / [WDC_DMASGBufLock\(\)](#) / [WD_DMALock\(\)](#)) also map the physical memory of the allocated buffer into virtual user mode space. Therefore, there must also be enough free virtual memory to enable the mapping of the entire buffer into the user space.

11.5.4. How do I perform PCI DMA Writes from system memory to my card, using WinDriver?

See [11.2. Performing Direct Memory Access \(DMA\)](#) and [11.3. Performing Direct Memory Access \(DMA\) transactions](#).

11.5.5. How do I perform Direct Block transfers from one PCI card to another?

Locate and register both cards (using [WDC_PciScanDevices\(\)](#), [WDC_PciGetDeviceInfo\(\)](#) and [WDC_PciScanDevices\(\)](#)). At least one of the cards must be PCI DMA Master Capable.

Program it with the physical address of the Slave card. Obtaining this address is possible by using `pciCard ← Card.Item[i].I.Mem.pPhysicalAddr`, set by [WDC_PciGetDeviceInfo\(\)](#) for the slave card.

Refer to the DMA section of this manual for more info regarding programming DMA using WinDriver.

Chapter 12

Understanding the Kernel PlugIn

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

12.1. 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 Kernel PlugIn is available for Windows and Linux, and it is an integral part of the WinDriver PCI/ISA toolkit that does not require additional licensing.

Writing a Kernel PlugIn driver provides the following advantages over a standard OS kernel mode driver:

- All the driver code is written and debugged in the user mode.
- The code segments that are moved to the 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.

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.

12.2. Expected Performance

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.

12.3. Overview of the Development Process

Using the WinDriver Kernel PlugIn, you normally first develop and debug the driver in the user mode, using 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.

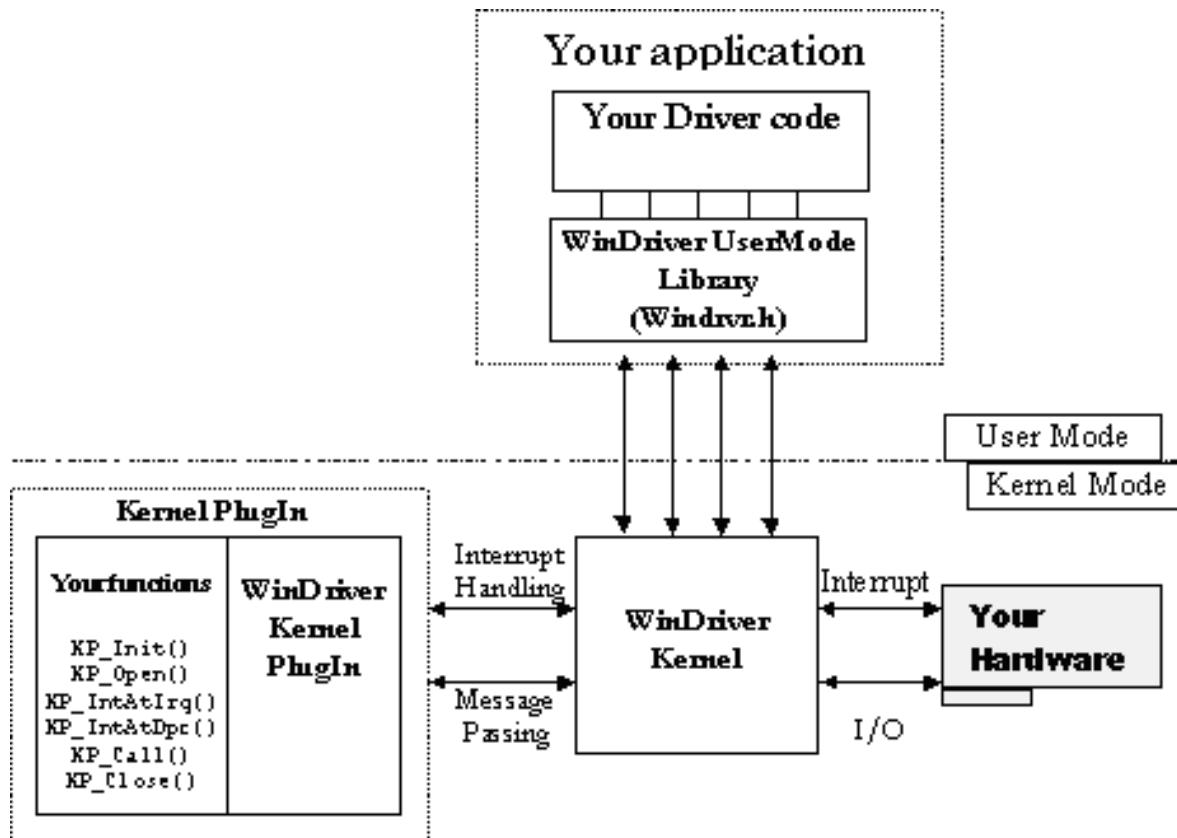
12.4. The Kernel PlugIn Architecture

12.4.1. Architecture Overview

A driver written in user mode uses WinDriver's API (WDC_xxx and/or WD_xxx) 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.

Kernel Plugin Architecture



12.4.2. 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 consist 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 that you rewrite the interrupt acknowledgment and handling code in the Kernel Plugin to utilize the flexibility offered by the Kernel Plugin (see [12.5.5. Handling Interrupts in the Kernel Plugin](#)).

- **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.

12.4.3. 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 - windrvr1511.sys/.o/.ko, depending on the operating system.
- Kernel Plugin driver (<Kernel Plugin driver name>/ .sys/.o/.ko/.kext), 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.

12.4.4. Kernel Plugin Event Sequence

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

12.4.4.1. Opening a Handle from the User Mode to a Kernel Plugin Driver

Event

Windows loads your Kernel Plugin driver.

This takes place at boot time, by dynamic loading, or as instructed by the registry.

Callback

Your KP_Init Kernel Plugin routine is called.

KP_Init informs WinDriver of the name(s) of your KP_Open routine(s). WinDriver calls the relevant open routine when there is a user-mode request to open a handle to your Kernel Plugin driver.

Event

Your user-mode driver application requests a handle to your Kernel Plugin driver, by calling one of the following functions:

- [WDC_KernelPluginOpen\(\)](#)
- [WDC_PciDeviceOpen\(\)](#)/[WDC_IsaDeviceOpen\(\)](#) (PCI / ISA) with the name of the Kernel Plugin driver.
- [WD_KernelPluginOpen\(\)](#) - when using the low-level WinDriver API.

Callback

The relevant KP_Open Kernel Plugin callback routine is called.

The KP_Open callback 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.

12.4.4.2. Handling User-Mode Requests from the Kernel Plugin

Event

Your application calls [WDC_CallKerPlug\(\)](#), or the low-level [WD_KernelPluginCall\(\)](#) function.

Your application calls [WDC_CallKerPlug\(\)](#) / [WD_KernelPluginCall\(\)](#) 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 KP_Call Kernel Plugin routine is called. KP_Call executes code according to the message passed to it from the user mode.

12.4.4.3. Interrupt Handling - Enable/Disable and High Interrupt Request Level Processing

Event

Your application calls [WDC_IntEnable\(\)](#) with the fUseKP parameter set to TRUE (after having opened a handle to the Kernel Plugin), or calls the low-level [InterruptEnable\(\)](#) or [WD_IntEnable\(\)](#) functions with a handle to a Kernel Plugin driver (set in the hKernelPlugin field of the [WD_INTERRUPT](#) structure passed to the function).

Callback

Your KP_IntEnable Kernel Plugin routine is called. This function should contain any initialization required for your Kernel Plugin interrupt handling.

Event

Your hardware creates an interrupt.

Callback

Your high-IRQL Kernel Plugin interrupt handler routine - KP_IntAtIrql (legacy interrupts) or KP_IntAtIrqlMSI (MSI/MSI-X) - is called.

KP_IntAtIrql and KP_IntAtIrqlMSI run at a high priority, and therefore should perform only the basic interrupt handling, such as lowering the HW interrupt signal of level-sensitive interrupts to acknowledge the interrupt. If more interrupt processing is required, KP_IntAtIrql (legacy interrupts) or KP_IntAtIrqlMSI (MSI/MSI-X) can return TRUE in order to defer additional processing to the relevant deferred processing interrupt handler - KP_IntAtDpc or KP_IntAtDpcMSI .

Event

Your application calls [WDC_IntDisable\(\)](#) , or the low-level [InterruptDisable\(\)](#) or [WD_IntDisable\(\)](#) functions, when the interrupts were previously enabled in the Kernel Plugin (see the description of the interrupt enable event above).

Callback

Your KP_IntDisable Kernel Plugin routine is called. This function should free any memory that was allocated by the KP_IntEnable callback .

12.4.4.4. Interrupt Handling - Deferred Procedure Calls Event/Callback Notes

Event

The Kernel Plugin high-IRQL interrupt handler - KP_IntAtIrql or KP_IntAtIrqlMSI - returns TRUE. This informs WinDriver that additional interrupt processing is required as a Deferred Procedure Call (DPC) in the kernel.

Callback

Your Kernel Plugin DPC interrupt handler - KP_IntAtDpc (legacy interrupts) or KP_IntAtDpcMSI (MSI/MSI-X) - is called. Processes the rest of the interrupt code, but at a lower priority than the high-IRQL interrupt handler.

Event

The DPC interrupt handler - KP_IntAtDpc or KP_IntAtDpcMSI - returns a value greater than 0. This informs WinDriver that additional usermode interrupt processing is required.

Callback

[WD_IntWait\(\)](#) returns. Your user-mode interrupt handler routine is executed.

12.4.4.5. Plug-and-Play and Power Management Events

Event

Your application registers to receive Plug-and-Play and power management notifications 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), or calls the low-level [EventRegister\(\)](#) 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).

Event

A Plug-and-Play or power management event (to which the application registered to listen) occurs.

Callback

Your KP_Event Kernel Plugin routine is called. KP_Event receives information about the event that occurred and can proceed to handle it as needed.

Event

KP_Event returns TRUE. This informs WinDriver that the event also requires user-mode handling.

Callback

`WD_IntWait()` returns. Your user-mode event handler routine is executed.

12.5. 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 the user mode. Then, if you encounter performance problems or require greater flexibility, port portions of your code to a Kernel Plugin driver.

12.5.1. Minimal Requirements for Creating a Kernel Plugin Driver

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

- On Windows:

The Windows Driver Kit (WDK), including its C build tools. The WDK is available as part of a Microsoft Development Network (MSDN) subscription, or from Microsoft Connect. For more information, refer to Microsoft's Windows Driver Kit (WDK) page.

- On Linux:

GCC, gmake or make.

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.

12.5.2. Kernel Plugin Implementation

12.5.2.1. 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 [12.4.4. Kernel Plugin Event Sequence](#) for details. For example, `KP_Init` is the callback function that is called when the driver is loaded.

The name of your driver is given in `KP_Init`. The Kernel Plugin driver's implementation of this callback must be named `KP_Init`. The names of the other Kernel Plugin callback functions (which are passed to `KP_Init`) are left to the discretion of the driver developer.

It is the convention of this reference guide to refer to these callbacks using the format `KP_<Functionality>` - for example, `KP_Open`.

When generating Kernel Plugin code with the DriverWizard, the names of the callback functions (apart from `KP_Init`) conform to the following format: `KP_<Driver Name>_<Functionality>`. For example, if you named your project `MyDevice`, the name of your Kernel Plugin `KP_Call` callback will be `KP_MyDevice_Call`.

12.5.2.2. Write Your `KP_Init` Function

Your `KP_Init` function should be of the following prototype: `BOOL __cdecl KP_Init (KP_INIT *kpInit);`

This function is called once, when the driver is loaded. The function should fill the received `KP_INIT` structure with the name of the Kernel Plugin driver, the name of the WinDriver Kernel Plugin driver library, and the driver's `KP_Open` callback(s) (see example in `WinDriver/samples/c/pci_diag/kp_pci/kp_pci.c`).

The name that you select for your Kernel PlugIn driver - by setting it in the cDriverName field of the **KP_INIT** structure that is passed to KP_Init - 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 used when opening a handle to the Kernel PlugIn driver in the user mode (see [13.5. Open a Handle to the Kernel PlugIn](#)) - in the pKPOpenData parameter of the **WDC_KernelPlugInOpen()** or **WDC_PciDeviceOpen()**/**WDC_IsaDeviceOpen()** (PCI / ISA) functions, or in the pcDriverName field of the pKernelPlugIn parameter passed to the low-level **WD_KernelPlugInOpen()** function - 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 is to define the 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/c/pci_diag/kp_pci/kp_pci.c`):

```
/* KP_Init is called when the Kernel PlugIn driver is loaded. This function sets the name of the Kernel PlugIn driver
and the driver's open callback function(s). */
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)
    {
        /* Rebuild your Kernel PlugIn driver project with the compatible
        version of the WinDriver Kernel PlugIn library (kp_nt<version>.lib)
        and windrvr.h and wd_kp.h files */
        return FALSE;
    }
    kpInit->funcOpen = KP_PCI_Open;
    kpInit->funcOpen_32_64 = KP_PCI_VIRT_Open_32_64;
    strcpy (kpInit->cDriverName, KP_PCI_DRIVER_NAME);
    return TRUE;
}
```

Note that the driver name in the sample is set using a preprocessor definition. This definition is found in the `WinDriver/samples/c/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"
```

12.5.2.3. Write Your KP_Open Function(s)

You can implement either one or two KP_Open functions, depending on your target configuration. The KP_Open function(s) 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 opening a handle to the Kernel PlugIn driver from the user mode - i.e., when **WD_KernelPlugInOpen()** is called, either directly or via **WDC_KernelPlugInOpen()** / ISA), as explained in [13.5. Open a Handle to the Kernel PlugIn](#).

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	Description
KP_Close	Called when the WD_KernelPlugInClose() function is called from the user mode - either directly, or via one of the high-level WDC_PciDeviceClose() / WDC_IsaDeviceClose() functions (PCI / ISA) when called for a device that contains an open Kernel PlugIn handle (see 13.5. Open a Handle to the Kernel PlugIn).
KP_Call	Called when the user-mode application calls the WDC_CallKerPlug() function or the low-level WD_KernelPlugInCall() function, which is called by the wrapper WDC_CallKerPlug() function. This function implements a Kernel PlugIn message handler.

Callback	Description
KP_IntEnable	Called when the user-mode application enables Kernel Plugin interrupts, by calling WDC_IntEnable() with the fUseKP parameter set to TRUE (after having opened a Kernel Plugin handle), or by calling the low-level InterruptEnable() or WD_IntEnable() 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	Called when the user-mode application calls WDC_IntDisable() , or the low-level InterruptDisable() or WD_IntDisable() 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 callback.
KP_IntAtIrql	Called when WinDriver receives a legacy interrupt, provided the received interrupt was enabled with a handle to the Kernel Plugin. This is the function that will handle your legacy interrupt in the kernel mode. The function runs at high interrupt request level. Additional deferred processing of the interrupt can be performed in KP_IntAtDpc and also in the user mode (see below).
KP_IntAtDpc	Called if the KP_IntAtIrql callback has requested deferred handling of a legacy 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 usermode interrupt handler routine will be invoked (if at all).
KP_IntAtIrqlMSI	Called when WinDriver receives an MSI or MSI-X, provided MSI/MSI-X was enabled for the received interrupt with a handle to the Kernel Plugin. This is the function that will handle your MSI/MSI-X in the kernel mode. The function runs at high interrupt request level. Additional deferred processing of the interrupt can be performed in KP_IntAtDpcMSI and also in the user mode (see below).
KP_IntAtDpcMSI	Called if the KP_IntAtIrqlMSI callback has requested deferred handling of an MSI/MSI-X interrupt by returning TRUE. This function should include lower-priority kernel-mode MSI↔MSI-X handler code. The return value of this function determines the amount of times that the application's usermode interrupt handler routine will be invoked (if at all).
KP_Event	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() with the fUseKP parameter set to TRUE (after having opened a Kernel Plugin handle), or by calling the low-level EventRegister() 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).

In addition to defining the Kernel Plugin callback functions, you can implement code to perform any required initialization for the Kernel Plugin in your KP_Open callback(s). In the sample KP_PCI driver and in the generated DriverWizard Kernel Plugin driver, for example, the Kernel Plugin open callbacks also call the shared library's initialization function and allocate 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/c/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 the rest of the Kernel Plugin callback functions. */
BOOL __cdecl KP_PCI_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData,
PVOID *ppDrvContext)
{
    PCI_DEV_ADDR_DESC *pDevAddrDesc;
    WDC_ADDR_DESC *pAddrDesc;
    DWORD dwSize;
    DWORD dwStatus;
    /* 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_GetLastErr());
        return FALSE;
    }
    KP_PCI_Trace("KP_PCI_Open entered. PCI library initialized.\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->funcIntAtIrqlMSI = KP_PCI_IntAtIrqlMSI;
kpOpenCall->funcIntAtDpcMSI = KP_PCI_IntAtDpcMSI;
kpOpenCall->funcEvent = KP_PCI_Event;

/* Create a copy of device information in the driver context */
dwSize = sizeof(PCI_DEV_ADDR_DESC);
pDevAddrDesc = malloc(dwSize);
if (!pDevAddrDesc)
    goto malloc_error;

COPY_FROM_USER(pDevAddrDesc, pOpenData, dwSize);
dwSize = sizeof(WDC_ADDR_DESC) * pDevAddrDesc->dwNumAddrSpaces;
pAddrDesc = malloc(dwSize);
if (!pAddrDesc)
    goto malloc_error;

COPY_FROM_USER(pAddrDesc, pDevAddrDesc->pAddrDesc, dwSize);
pDevAddrDesc->pAddrDesc = pAddrDesc;
*ppDrvContext = pDevAddrDesc;
KP_PCI_Trace("KP_PCI_Open: Kernel PlugIn driver opened successfully\n");
return TRUE;
malloc_error:
    KP_PCI_Err("KP_PCI_Open: Failed allocating %ld bytes\n", dwSize);
    if (pDevAddrDesc)
        free(pDevAddrDesc);
    PCI_LibUninit();
    return FALSE;
}

```

The KP_PCI sample also defines a similar KP_PCI_Open_32_64 callback, for use when opening a handle to a 64-bit Kernel PlugIn from a 32-bit application.

12.5.2.4. 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).

12.5.3. Sample/Generated Kernel Plugin Driver Code Overview

You can use DriverWizard to generate a skeletal Kernel Plugin driver for your device, and use the generated code as the basis for your Kernel Plugin driver development (recommended); alternatively, you can use one of the Kernel Plugin WinDriver samples as the basis for your Kernel Plugin development.

The Kernel Plugin documentation in this manual focuses on the generated DriverWizard code, and the generic PCI Kernel Plugin sample - KP_PCI, located in theWinDriver/samples/c/pci_diag/kp_pci directory.

If you are using the a PCI Express card with the Xilinx Bus Master DMA (BMD) design, you can also use the KP↔BMD Kernel Plugin sample as the basis for your development; the WinDriver/samples/c/xilinx/bmd↔_design directory contains all the relevant sample files - see the Xilinx BMD Kernel Plugin directory structure note at the end of 12.5.4.1. [pci_diag and kp_pci Sample Directories](#).

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

Both the KP_PCI sample and the wizard-generated code demonstrate communication between a user-mode application (pci_diag / xxx_diag - where xxx is the name you selected for your generated driver project) and a Kernel Plugin driver (kp_pci.sys/.o/.ko/.kext / kp_xxx.sys/.o/.ko/.kext - depending on the OS).

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 that 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 and interrupt handlers, including deferred processing interrupt handling, which is used to notify the user-mode application of the arrival of every fifth incoming interrupt.

The KP_PCI sample's KP_IntAtIrql() functions demonstrate legacy level-sensitive PCI interrupt handling. 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 acknowledgement is hardware-specific. The sample KP_IntAtIrqlMSI() and KP_IntAtDpcMSI() functions demonstrate handling of Message-Signaled Interrupts (MSI) and Extended Message-Signaled Interrupts (MSI-X) (see detailed information in [10.4. Single Root I/O Virtualization \(SR-IOV\)](#)).

The generated DriverWizard code will include sample interrupt handler code for the selected device (PCI/ISA). The generated KP_IntAtIrql function will include code to implement any interrupt transfer commands defined in the wizard (by assigning registers read/write commands to the card's interrupt in the Interrupt tab).

For legacy PCI interrupts, which need to be acknowledged in the kernel when the interrupt is received (see [10.4. Single Root I/O Virtualization \(SR-IOV\)](#)), 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.

It is also recommended that you prepare such transfer commands when handling interrupts for hardware that supports MSI/MSI-X, in case enabling of MSI/MSI-X fails and the interrupt handling defaults to using level-sensitive interrupts (if supported by the hardware).

** Attention**

Memory allocated for the transfer commands must remain available until the interrupts are disabled .

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

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 you will need to modify or remove the hardware-specific transfer commands in the sample's KP_IntAtIrql function, as explained above.

12.5.4. Kernel PlugIn Sample/Generated Code Directory Structure

12.5.4.1. pci_diag and kp_pci Sample Directories

The KP_PCI Kernel PlugIn sample code is implemented in the [kp_pci.c](#) file. This sample driver is part of the Win↔Driver PCI diagnostics sample - pci_diag - which contains, in addition to the KP_PCI driver, a user-mode application that communicates with the driver (pci_diag) and a shared library that includes APIs 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/samples/c/pci_diag directory:

- kp_pci - Contains the KP_PCI Kernel PlugIn driver files:
 - [kp_pci.c](#): The source code of the KP_PCI driver.
 - Project and/or make files and related files for building the Kernel PlugIn driver.

The Windows project/make files are located in subdirectories for the target development environment (`msdev-<version>/win_gcc`) under x86 (32-bit) and amd64 (64-bit) directories. The Linux makefile is generated using a configure script, located directly under the kp_pci directory.

- A pre-compiled version of the KP_PCI Kernel PlugIn driver for the target OS:

Windows x86 32-bit: `WINNT.i386\kp_pci.sys` - a 32-bit version of the driver. Windows x64: `WINNT.x86->_64\kp_pci.sys` - a 64-bit version of the driver. Linux: There is no pre-compiled version of the driver for Linux, since Linux kernel modules must be compiled with the header files from the kernel version installed on the target - see [15.3. Linux Driver Distribution](#)).

- `pci_lib.c`: Implementation of a library for accessing PCI devices using WinDriver's WDC API. The library's API is used both by the user-mode application (`pci_diag.c`) and by the Kernel PlugIn driver (`kp_pci.c`).
- `pci_lib.h`: Header file, which provides the interface 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 Kernel PlugIn driver. If successful, the sample demonstrates how to interact with a Kernel PlugIn driver, as detailed in [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#). 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` (Windows): A sample WinDriver PCI INF file for Windows. NOTE: To use this file, change the vendor and device IDs in the file to comply with those of your specific device.

To use Message-Signaled Interrupts (MSI) or Extended Message-Signaled Interrupts (MSI-X) on Windows (for PCI cards that support MSI/MSI-X) you will need to modify or replace the sample INF file so that your INF file includes specific MSI information; otherwise WinDriver will attempt to use legacy level-sensitive interrupt handling for your card, as explained in [10.1.7.1. Windows MSI/MSI-X Device INF Files](#).

- Project and/or make files for building the `pci_diag` user-mode application.

The Windows project/make files are located in subdirectories for the target development environment (`msdev_<version>/win_gcc`) under `x86` (32-bit) and `amd64` (64-bit) directories. The `msdev_<version>` MS Visual Studio directories also include solution files for building both the Kernel PlugIn driver and user-mode application projects. The Linux makefile is located under a `LINUX` subdirectory.

- A pre-compiled version of the user-mode application (`pci_diag`) for your target operating system:

Windows: `WIN32\pci_diag.exe` Linux: `LINUX/pci_diag`

- `files.txt`: A list of the sample `pci_diag` files.
- `readme.pdf`: An overview of the sample Kernel PlugIn driver and user-mode application and instructions for building and testing the code.

12.5.4.2. Xilinx BMD Kernel PlugIn Directory Structure

The structure of the sample directory for PCI Express cards with the Xilinx Bus Master DMA (BMD) design - `WinDriver/samples/c/xilinx/bmd_design` - is similar to that of the generic PCI sample's `pci_diag` directory, except for the following issues: the `bmd_diag` user-mode application files are located under a `diag` subdirectory, and the `kp` subdirectory, which contains the Kernel PlugIn driver's (`KP_BMD`) source files, currently has make files only for Windows.

12.5.4.3. 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 wizard-generated code will utilize the resources information detected and/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 [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#), when using the driver to handle legacy PCI interrupts, it is highly recommended that you define the registers that need to be read/written in order to acknowledge the interrupt, and set up the relevant read/write commands from/to these registers in DriverWizard, before generating the code, thus enabling the generated interrupt handler code to utilize the hardware-specific information that you defined. It is also recommended that you prepare such transfer commands when handling interrupts for hardware that supports MSI/MSI-X, in case enabling of MSI/MSI-X fails and the interrupt handling defaults to using level-sensitive interrupts (if supported by the hardware).

**** Attention****

Memory allocated for the transfer commands must remain available until the interrupts are disabled .

Following is an outline of the generated DriverWizard files when selecting to generate Kernel PlugIn code (where `xxx` signifies the name that you selected for the driver when generating the code).

The outline below relates to the generated C code, but on Windows you can also generate similar C# code, which includes a C Kernel PlugIn driver (since kernel-mode drivers cannot be implemented in C#), a .NET C# library, and a C# user-mode application that communicates with the Kernel PlugIn driver.

- `kernmode` - 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.

The Windows project/make files are located in subdirectories for the target development environment (`msdev->_<version>/win_gcc`) under x86 (32-bit) and amd64 (64-bit) directories. The Linux makefile is generated using a configure script, located in a linux subdirectory.

- `xxx_lib.c`: Implementation of a library for accessing your device using WinDriver's WDC API. 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, which provides the interface for the `xxx_lib` library.
- `xxx_diag.c`: Implementation of a sample diagnostics user-mode console (CUI) application, which demonstrates communication with your device using the `xxx_lib` and WDC libraries.

The application also demonstrates how to communicate with a Kernel PlugIn driver from a user-mode WinDriver application. By default, the application attempts to open your device with a handle to the KP_XXX Kernel PlugIn driver. If successful, the application demonstrates how to interact with a Kernel PlugIn driver, as detailed in [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#). If the application fails to open a handle to the Kernel PlugIn driver, all communication with the device is performed from the user mode.

- Project and/or make files for building the `xxx_diag` user-mode application.

The Windows project/make files are located in subdirectories for the target development environment (`msdev->_<version>/win_gcc`) under x86 (32-bit) and amd64 (64-bit) directories. The `msdev_<version>` MS Visual Studio directories also include solution files for building both the Kernel PlugIn driver and user-mode application projects. The Linux makefile is located in a linux subdirectory.

- `xxx_files.txt`: A list of the generated files and instructions for building the code.
- `xxx.inf` (Windows): A WinDriver INF file for your device. This file is required only when creating a Windows driver for a Plug-and-Play device, such as PCI.

12.5.5. 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 (see [12.5.5.2. Interrupt Handling in the Kernel \(Using the Kernel PlugIn\)](#)).

If Kernel PlugIn interrupts were enabled, when WinDriver receives a hardware interrupt, it calls the Kernel PlugIn driver's high-IRQL handler - `KP_IntAtIrql` (legacy interrupts) or `KP_IntAtIrqlMSI` (MSI/MSI-X). If the high-IRQL handler returns TRUE, the relevant deferred Kernel PlugIn interrupt handler - `KP_IntAtDpc` (legacy interrupts) or `KP_IntAtDpcMSI` (MSI/MSI-X) - will be called after the high-IRQL handler completes its processing and returns. The return value of the DPC function determines how many times (if at all) the user-mode interrupt handler routine will be executed.

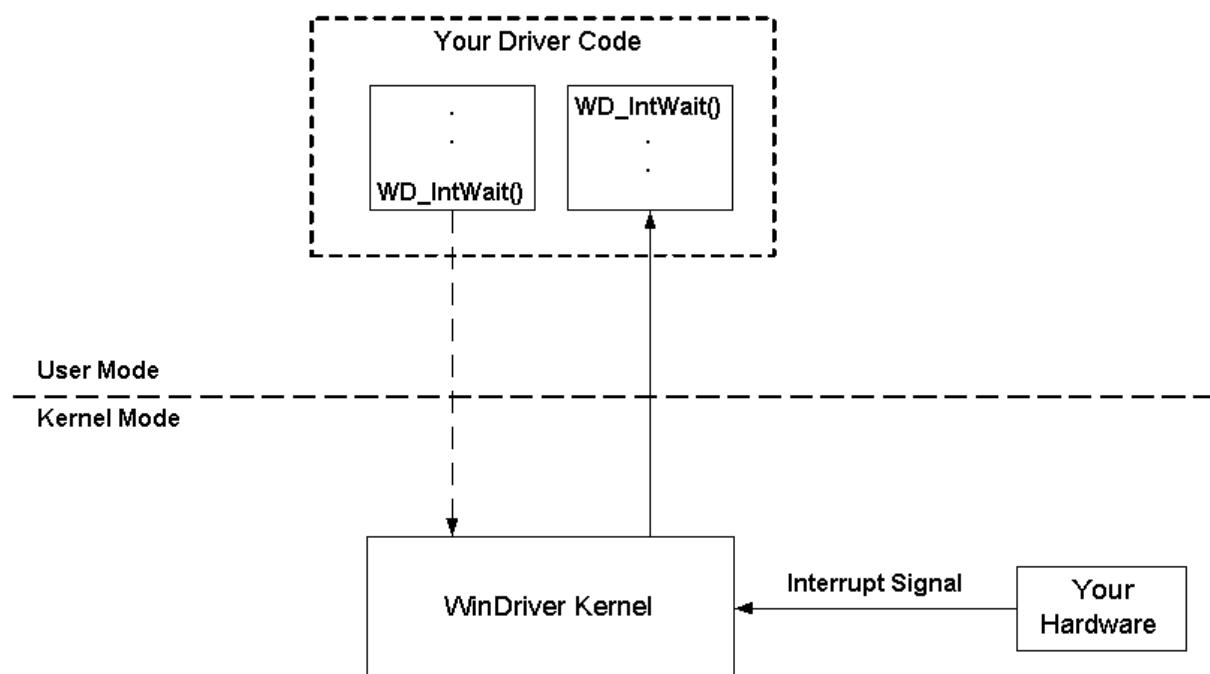
In the KP_PCI sample, for example, the Kernel PlugIn interrupt handler code counts five interrupts, and notifies the user mode on every fifth interrupt; thus `WD_IntWait()` will return on only one out of every five incoming interrupts in the user mode.

The high-IRQL handler - KP_IntAtIrql or KP_IntAtIrqlMSI - returns TRUE every five interrupts to activate the DPC handler - KP_IntAtDpc or KP_IntAtDpcMSI - and the DPC function returns the number of accumulated DPC calls from the high-IRQL handler. As a result, the user-mode interrupt handler will be executed once for every 5 interrupts.

12.5.5.1. Interrupt Handling in the User Mode (Without the 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()` or the low-level `InterruptEnable()` or `WD_IntEnable()` functions) - see Figure below.

Interrupt Handling Without Kernel Plugin

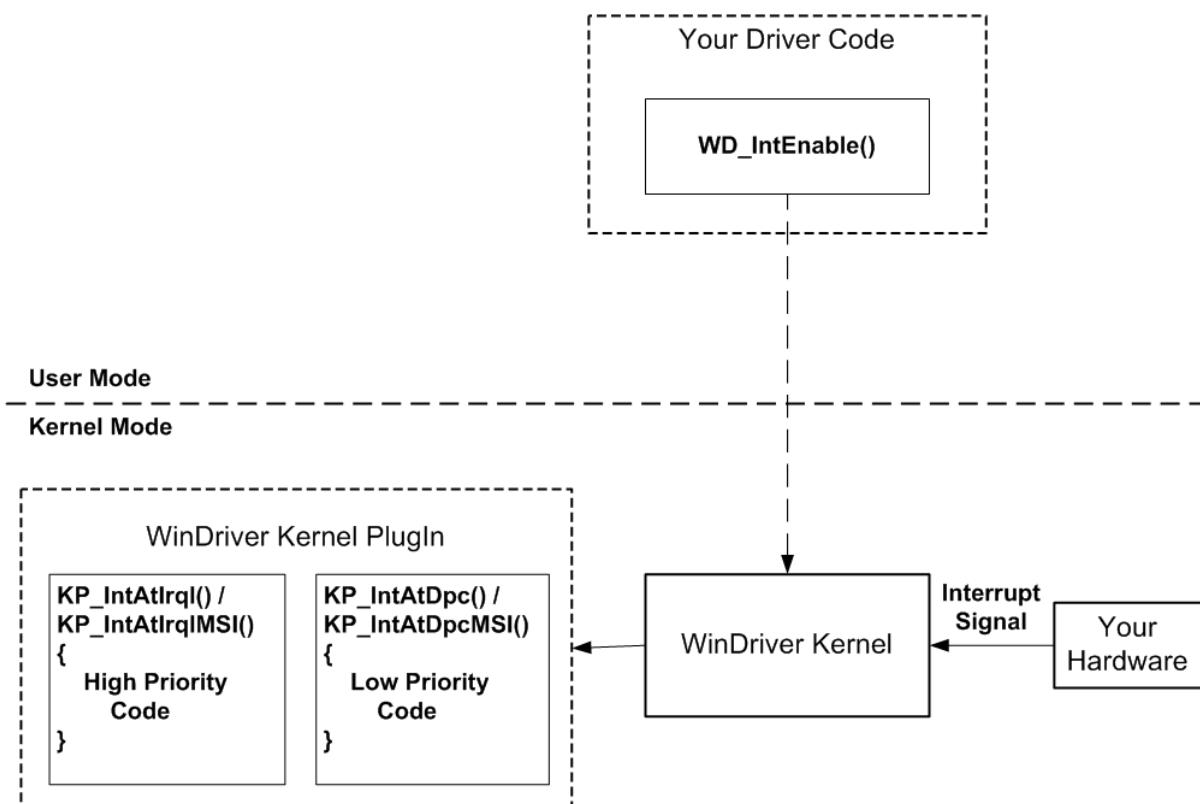


12.5.5.2. Interrupt Handling in the Kernel (Using the Kernel Plugin)

To have the interrupts handled by the Kernel Plugin, the user-mode application should open a handle to a Kernel Plugin driver (as explained in [13.5. Open a Handle to the Kernel Plugin](#)), and then call `WDC_IntEnable()` with the `fUseKP` parameter set to TRUE.

If you are not using the `WDC_xxx` API, your application should pass a handle to the Kernel Plugin driver to the `WD_IntEnable()` function or the wrapper `InterruptEnable()` function (which calls `WD_IntEnable()` and `WD_IntWait()`). 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.

Interrupt Handling With the Kernel Plugin



When calling [WDC_IntEnable\(\)](#) / [InterruptEnable\(\)](#) / [WD_IntEnable\(\)](#) to enable interrupts in the Kernel Plugin, your Kernel Plugin's `KP_IntEnable` callback function 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 the relevant high-IRQL handler, based on the type of interrupt that was enabled - `KP_IntAtIrql` (legacy interrupts) or `KP_IntAtIrqlMSI` (MSI/MSI-X) - will be called for each incoming interrupt. The code in the high-IRQL handler 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.

Code running at high IRQL is limited in the following ways:

- It may only access non-pageable memory.
- It may only call the following functions (or wrapper functions that call these functions):
- `WDC_xxx()` read/write address or configuration space functions.
- [WDC_MultiTransfer\(\)](#) , or the low-level [WD_Transfer\(\)](#), [WD_MultiTransfer\(\)](#), or [WD_DebugAdd\(\)](#) functions.
- Specific kernel OS functions (such as WDK 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 the high-IRQL handler (`KP_IntAtIrql` or `KP_IntAtIrqlMSI`) 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 the DPC function (`KP_IntAtDpc` or `KP_IntAtDpcMSI`), which runs at a deferred interrupt level and does not face the same limitations as the high-IRQL handlers. The DPC function is called after its matching high-IRQL function returns, provided the high-IRQL handler returns TRUE.

You can also leave some additional interrupt handling to the user mode. The return value of your DPC function - `KP_IntAtDpc()` - 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.

12.5.6. 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\(\)](#) or the low-level [WD_KernelPlugInCall\(\)](#) function.

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` 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 .

12.6. FAQ

12.6.1. Why does my `WD_KernelPlugInOpen()` call fail?

There are several reasons why the call to [WD_KernelPlugInOpen\(\)](#) might fail:

- You did not set up the parameters correctly in the call to [WD_KernelPlugInOpen\(\)](#). Please check this carefully.
- You did not set up the driver name correctly in all locations in the code. Please verify that you are using the correct Kernel PlugIn driver name in all the relevant locations in the code. Look specifically at the Kernel PlugIn `KP_Init()` implementation and at `kernelPlugIn.pcDriverName` in the user mode project. Please indicate the driver name in capital letters and without the file extension (*.sys/.ko).
- You did not install the driver properly. Make sure you copied the driver file that was created to the correct location (e.g.,`windir%\system32\drivers` - to install a SYS Kernel PlugIn driver on Windows) and that you installed the Kernel PlugIn kernel module correctly, as explained in the this Manual.

For Windows, for example, make sure to use the correct `wdreg/wdreg_gui/wdreg_frontend` syntax. For SYS drivers:

```
wdreg -name <your KP driver name> install
```

- For registered users. You did not call [WD_License\(\)](#) before calling [WD_KernelPlugInOpen\(\)](#). If you are using a registered version of WinDriver, remember to call [WD_License\(\)](#) (or a registration function, which calls [WD_License\(\)](#)) before calling [WD_KernelPlugInOpen\(\)](#).

** Attention**

[WD_KernelPlugInOpen\(\)](#) is also called from the high-level [WDC_KernelPlugInOpen\(\)](#) function, and from the [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) functions - when they are called with the name of a Kernel PlugIn driver.

12.6.2. When handling my interrupts entirely in the Kernel PlugIn, can I erase the interrupt handler in the user mode?

Yes, you can erase the user mode interrupt handler routine.

You can also implement some of the interrupt handling in the Kernel PlugIn and some of it in the user mode. The return value of `KP_IntAtDpc()` (which is called when the high-priority `KP_IntAtIrql()` routine returns TRUE) determines the number of times that the user mode interrupt handler routine will be executed (if at all).

12.6.3. How can I print debug statements from the Kernel Plugin that I can view using a kernel debugger, such as WinDbg?

You can use WinDriver's [WD_DebugAdd\(\)](#) function to print debug messages from your Kernel Plugin or user-mode code to the Debug Monitor utility, and then view the messages in the Debug Monitor log. [WD_DebugAdd\(\)](#) can be called from within any user-mode or Kernel Plugin function, including [KP_IntAtIrq\(\)](#).

You can also select to send the debug information from WinDriver's Debug Monitor to a kernel debugger, as explained in [Debugging Drivers](#).

In addition, you can add calls in your Kernel Plugin code to OS kernel functions that print directly to the kernel debugger - such as `KdPrint()` on Windows, or `printf()` on Linux.

12.6.4. My PC hangs while closing my application. The code fails in [WD_IntDisable\(\)](#). Why is this happening? I am using the Kernel Plugin to handle interrupts.

This might happen if you are enabling the interrupt from your Kernel Plugin interrupt routines, and simultaneously disabling it from the user mode (using [WD_IntDisable\(\)](#) or [InterruptEnable\(\)](#) / [WDC_IntEnable\(\)](#) - which call [WD_IntDisable\(\)](#)). Since the interrupt is active (having been enabled from the Kernel Plugin), the interrupt cannot be disabled and the PC hangs waiting for [WD_IntDisable\(\)](#) to return.

A possible solution, is to call [WD_IntDisable\(\)](#) / [InterruptEnable\(\)](#) / [WDC_IntEnable\(\)](#) as an atomic operation, so that it will disable the interrupts successfully before the Kernel Plugin interrupt routine enables the interrupt.

Chapter 13

Creating a Kernel PlugIn Driver

The easiest way to write a Kernel PlugIn driver is to use DriverWizard to generate the Kernel PlugIn code for your hardware (see [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#) and [12.5.4.3. The Generated DriverWizard Kernel PlugIn Directory](#)).

Alternatively, you can use one of the WinDriver Kernel PlugIn samples as the basis for your Kernel PlugIn development. You can also develop your code "from scratch", if you wish.

As indicated in [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#), the Kernel PlugIn documentation in this manual focuses on the generated DriverWizard code, and the generic PCI Kernel PlugIn sample - KP↔_PCI, located in the `WinDriver/samples/c/pci_diag/kp_pci` directory. If you are using the a PCI Express card with the Xilinx Bus Master DMA (BMD) design, you can also use the `KP_BMD` Kernel Plug↔In sample as the basis for your development; the `WinDriver/samples/c/xilinx/bmd_design` directory contains all the relevant sample files - see the Xilinx BMD Kernel PlugIn directory structure note at the end of ([12.5.4.1. pci_diag and kp_pci Sample Directories](#)).

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

13.1. Determine Whether a Kernel PlugIn is Needed

The Kernel PlugIn should be used only after your driver code has been written and debugged in the 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 [Improving PCI Performance](#), 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 in regard to the interrupt handling).

13.2. What programming languages can be used with a Kernel PlugIn?

The Kernel PlugIn is a kernel module, and therefore must be written in the C language. The user application that calls the Kernel PlugIn can be of any langauge supported by WinDriver: C, C#.NET, Visual Basic.NET, Python, Java.

When generating code in the DriverWizard in languages other than C with Kernel PlugIn, the Kernel PlugIn code will still be generated in C.

13.3. Prepare the User-Mode Source Code

Please follow this step:

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

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 the low-level [WD_CardRegister\(\)](#) function - 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\(\)](#)).

When using the API of the WDC library 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.

13.4. Create a New Kernel PlugIn Project

As indicated above, you can use DriverWizard to generate a new Kernel PlugIn project (and a corresponding user-mode project) for your device (recommended), or use one of the WinDriver Kernel PlugIn samples as the basis for your development.

** Attention**

To successfully build a Kernel PlugIn project using MS Visual Studio, the path to the project directory must not contain any spaces.

If you select to start your development with the KP_PCI sample, follow these steps:

- Make a copy of the `WinDriver/samples/c/pci_diag/kp_pci` directory. For example, to create a new Kernel PlugIn project called KP_MyDrv, copy `WinDriver/samples/c/pci_diag/kp_pci` to `WinDriver/samples/c/mydrv`.
- 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).

The names of the KP_PCI_xxx() functions in the `kp_pci.c` files do not have to be changed, but the code will be clearer if you use your selected driver name in the function names.

- Change all occurrences of "KP_PCI" in file names to "kp_mydrv".
- 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/c/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 usermode 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.

- 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).
- To use the `pci_diag` user-mode application, copy `WinDriver/samples/c/pci_diag/pci← diag.c` and the relevant `pci_diag` project, 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 usermode application name.

To use the 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 sample and generated Kernel PlugIn code and its structure, see [12.5.3. Sample/Generated Kernel PlugIn Driver Code Overview](#) and [12.5.4. Kernel PlugIn Sample/Generated Code Directory Structure](#) (respectively).

13.5. Open a Handle to the Kernel PlugIn

To open a handle to a Kernel PlugIn driver, `WD_KernelPlugIn()` needs to be called from the user mode. This low-level function is called from `WDC_KernelPlugInOpen()` - when it is called with the name of a Kernel PlugIn driver.

When using the high-level WDC API you should use the following method to open a Kernel PlugIn handle:

- First open a regular device handle - by calling the relevant `WDC_PciDeviceOpen()` / `WDC_IsaDeviceOpen()` function without the name of a Kernel PlugIn driver. Then call `WDC_KernelPlugInOpen()`, passing to it the handle to the opened device. `WDC_KernelPlugInOpen()` opens a handle to the Kernel PlugIn driver, and stores it within the `kerPlug` field of the provided device structure .
- Open a handle to the device, using the relevant `WDC_PciDeviceOpen()` / `WDC_IsaDeviceOpen()` function, and pass the name of a Kernel PlugIn driver within the function's `pcKPDriverName` parameter. The device handle returned by the function will also contain (within the `kerPlug` field) a Kernel PlugIn handle opened by the function.

** Attention**

This method cannot be used to open a handle to a 64-bit Kernel PlugIn driver from a 32-bit application, or to open a Kernel PlugIn handle from a .NET application.

To ensure that your code works correctly in all the supported configurations, use the first method described above.

The generated DriverWizard and the sample `pci_diag` shared library (`xxx_lib.c` / `pci_lib.c`) demonstrate how to open a handle to the Kernel PlugIn - see the generated/sample `XXX_DeviceOpen()` / `PCI↔DeviceOpen()` library function (which is called from the generated/sample `xxx_diag/pci_diag` user-mode application).

The handle to the Kernel PlugIn driver is closed when the `WD_KernelPlugInClose()` function is called from the user mode.

When using the low-level WinDriver API, this function is called directly from the user-mode application.

When using the high-level WDC API , the function is called automatically when calling `WDC_PciDeviceClose()` / `WDC_IsaDeviceClose()` with a device handle that contains an open Kernel PlugIn handle. The function is also called by WinDriver as part of the application cleanup, for any identified open Kernel PlugIn handles.

13.6. Set Interrupt Handling in the Kernel PlugIn

Kindly follow these instructions:

- When calling `WDC_IntEnable()` (after having opened a handle to the Kernel PlugIn driver, as explained in [13.5. Open a Handle to the Kernel PlugIn](#)), 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 , in order to enable interrupts in the Kernel PlugIn call `WD_IntEnable()` or `InterruptEnable()` (which calls `WD_IntEnable()`), and pass the handle to the Kernel PlugIn driver that you received from `WD_KernelPlugInOpen()` (within the `hKernelPlugIn` field of the `WD_KERNEL_PLUGIN` structure that was passed to the function).

- When calling `WDC_IntEnable()` / `InterruptEnable()` / `WD_IntEnable()`, to enable interrupts in the Kernel PlugIn, WinDriver will activate your Kernel PlugIn's `KP_IntEnable` callback function.

You can implement this function to set the interrupt context that will be passed to the high-IRQL and DPC Kernel PlugIn interrupt handler routines, 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.

- 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 relevant high-IRQL handler - KP_IntAtIrql (legacy interrupts) or KP_IntAtIrqMSI (MSI/MSI-X) - which runs at high interrupt request level.

Deferred processing of the interrupt can be moved to the relevant DPC handler - KP_IntAtDpc - which will be executed once the high-IRQL handler 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, or toggle specific register bits). For a detailed explanation on how to handle interrupts in the kernel using a Kernel PlugIn, refer to [12.5.5. Handling Interrupts in the Kernel PlugIn](#) of the manual.

13.7. Set I/O Handling in the Kernel PlugIn

Perform the next steps:

- Move your I/O handling code (if needed) from the user mode to the Kernel PlugIn message handler - KP_Call()
- To activate the kernel code that performs the I/O handling from the user mode, call [WDC_CallKerPlug\(\)](#) or the low-level [WD_KernelPlugInCall\(\)](#) function 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.
- 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 driver are defined in the `pci_lib.h/xxx_lib.h` shared library header file.

13.8. Compile Your Kernel PlugIn Driver

The Kernel PlugIn is not backward compatible. Therefore, when switching to a different version of WinDriver, you need to rebuild your Kernel PlugIn driver using the new version.

13.8.1. Windows Kernel PlugIn Driver Compilation

The sample `WinDriver\samples\c\pci_diag\kp_pci` Kernel PlugIn directory and the generated DriverWizard Kernel PlugIn `<project_dir>\kermode` directory (where `<project_dir>` is the directory in which you selected to save the generated driver project) contain the following Kernel PlugIn project files (where `xxx` is the driver name - `pci` for the sample / the name you selected when generating the code with the wizard):

- x86 - 32-bit project files:
 - `msdev_<version>\kp_xxx.vcxproj` - 32-bit MS Visual Studio project file (where `<version>` signifies the IDE version - e.g., "2019")
 - `win_gcc/makefile` - 32-bit Windows GCC (MinGW/Cygwin) makefile
- amd64 - 64-bit project files:
 - `msdev_<version>\kp_xxx.vcxproj` - 64-bit MS Visual Studio project file (where `<version>` signifies the IDE version - e.g., "2019")
 - `win_gcc/makefile` - 64-bit Windows GCC (MinGW/Cygwin) makefile

The sample `WinDriver\samples\c\pci_diag` directory and the generated `<project_dir>` directory contain the following project files for the user-mode application that drives the respective Kernel PlugIn driver (where `xxx` is the driver name - `pci` for the sample / the name you selected when generating the code with the wizard):

- x86 - 32-bit project files:
 - msdev_<version>\xxx_diag.vcxproj - 32-bit MS Visual Studio project file (where <version> signifies the IDE version - e.g., "2019")
 - win_gcc/makefile - 32-bit Windows GCC (MinGW/Cygwin) makefile
- amd64 - 64-bit project files:
 - msdev_<version>\xxx_diag.vcxproj - 64-bit MS Visual Studio project file (where <version> signifies the IDE version - e.g., "2019")
 - win_gcc/makefile - 64-bit Windows GCC (MinGW/Cygwin) makefile

The msdev_<version> MS Visual Studio directories listed above also contain xxx_diag.sln solution files that include both the Kernel PlugIn and user-mode projects.

If you used DriverWizard to generate your code and you selected to generate a dynamic link library (DLL), the generated <project_dir> directory will also have a libproj DLL project directory. This directory has x86 (32-bit) and/or amd64 (64-bit) directories that contain msdev_<version> directories for your selected IDEs, and each IDE directory has an xxx_libapi.vcproj project file for building the DLL. The DLL is used from the wizard-generated user-mode diagnostics project (xxx_diag.vcxproj).

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

- Verify that the Windows Driver Kit (WDK) is installed.
- Build the Kernel PlugIn SYS driver (kp_pci.sys - sample / kp_xxx.sys - wizard generated code):
 - Using MS Visual Studio - Start Microsoft Visual Studio, and do the following:

From your driver project directory, open the Visual Studio Kernel PlugIn solution file - <project_dir>\msdev_<version>\xxx_diag.sln, where <project_dir> is your driver project directory (pci_diag for the sample code / the directory in which you selected to save the generated DriverWizard code), msdev_<version> is your target Visual Studio directory (e.g., msdev_2019), and xxx is the driver name (pci for the sample / the name you selected when generating the code with the wizard).

When using DriverWizard to generate code for MS Visual Studio, you can use the IDE to Invoke option to have the wizard automatically open the generated solution file in your selected IDE, after generating the code files.

1. To successfully build a Kernel PlugIn project using MS Visual Studio, the path to the project directory must not contain any spaces.
 - Set the Kernel PlugIn project (kp_pci.vcxproj / kp_xxx.vcxproj) as the active project.
 - Select the active configuration for your target platform: From the Build menu, choose Configuration Manager..., and select the desired configuration.
2. To build the driver for multiple operating systems, select the lowest OS version that the driver must support from the Project Properties -> Driver Settings -> Target OS Version, select Windows 8 as it is the lowest version currently supported by WinDriver.
 - Build your driver: Build the project from the Build menu or using the relevant shortcut key (e.g. CTRL+SHIFT+B in Visual Studio).
3. Build the user-mode application that drives the Kernel PlugIn driver (pci_diag.exe - sample / xxx_diag.exe - wizard-generated code):
 - Using MS Visual Studio -
4. Set the user-mode project (pci_diag.vcxproj - sample / xxx_diag.vcxproj - wizard generated code) as the active project. Then build the application: Build the project from the Build menu or using the relevant shortcut key (e.g., CTRL+SHIFT+B in Visual Studio 2017).
 - Using Windows GCC -

Do the following from your selected Windows GCC development environment (MinGW/Cygwin):

Change directory to your target Windows GCC application directory - <project_dir>/<CPU>/win_gcc, where <project_dir> is your driver project directory (pci_diag for the sample code / the directory in which you selected to save the generated DriverWizard code), and <CPU> is the target CPU architecture (x86 for x86 platforms, and amd64 for x64 platforms).

For example:

- When building a 32-bit version of the sample pci_diag application, which drives the sample KP_PCI driver
 -

```
$ cd WinDriver/samples/c/pci_diag/x86/win_gcc
```

- When building a 64-bit wizard-generated user-mode application that drivers a wizard-generated Kernel Plugin driver -

```
$ cd <project_dir>/amd64/win_gcc
```

- where <project_dir> signifies the path to your generated DriverWizard project directory (for example, ~/WinDriver/wizard/my_projects/my_kp).
 - Build the application using the make command.

13.8.2. Linux Kernel Plugin Driver Compilation

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

- Open a shell terminal.
- Change directory to your Kernel Plugin directory.

For example:

- When building the sample KP_PCI driver -

```
$ cd WinDriver/samples/c/pci_diag/kp_pci
```

- When building a wizard-generated Kernel Plugin driver -

```
$ cd <project_dir>/kermode/linux/
```

- where <project_dir> signifies the path to your generated DriverWizard project directory (for example, ~/WinDriver/wizard/my_projects/my_kp).
 - Generate the makefile using the configure script:

```
$ ./configure
```

If you have renamed the WinDriver kernel module, be sure to uncomment the following line in your Kernel Plugin configuration script (by removing the pound sign - "#"), before executing the script, in order to build the driver with the -DWD_DRIVER_NAME_CHANGE flag (see [17.2.2. Linux Driver Renaming](#)):
ADDITIONAL_FLAGS="-DWD_DRIVER_NAME_CHANGE"

The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the --with-kernel-source=<path> option, where <path> is the full path to the kernel source directory - e.g., /usr/src/linux. If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use kbuild to compile the kernel modules.
.configure --help

- Build the Kernel Plugin module using the make command.

This command creates a new LINUX.<kernel version>.<CPU> directory, which contains the created kp_xxx_module.o/.ko driver.

- Change directory 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\
```

- Compile the sample diagnostics program using the `make` command.

13.8.3. Porting a Kernel PlugIn project developed prior to version 10.3.0, to support working with a 32-bit user-mode application and a 64-bit Kernel PlugIn driver

Beginning with version 10.3.0 of WinDriver, you can communicate with a 64-bit Kernel PlugIn driver from a 32-bit user-mode application. To support such a configuration, you need to keep in mind the following points:

- The application must open a handle to the Kernel PlugIn driver using [WDC_KernelPlugInOpen\(\)](#) (or the low-level [WD_KernelPlugInOpen\(\)](#) function, if you are not using the WDC library).

The handle cannot be opened using one of the WDC device-open functions, e.g. [WDC_PciDeviceOpen\(\)](#), by calling it with the name of a Kernel PlugIn driver. In all Kernel PlugIn configurations, except for a 64-bit driver that interacts with a 32-bit application, either of these methods can be used to open a handle to the Kernel PlugIn from the application.

- The Kernel PlugIn driver must implement a [KP_Open](#) callback that correctly translates the 32-bit data received from the user mode, into 64-bit kernel data.

The Kernel PlugIn driver's [KP_Init\(\)](#) function must set this callback in the `funcOpen_32_64` field of the [KP_INIT](#) structure that it initializes. This field was added in version 10.3.0 of WinDriver.

The callback set in the [KP_INIT](#) `funcOpen_32_64` field, is called by WinDriver when a 32-bit application opens a handle to a Kernel PlugIn driver. This callback will hereby be referred to as the `funcOpen_32_64` callback. The callback set in the [KP_INIT](#) `funcOpen` field is the standard [KP_Open](#) callback, which is called by WinDriver whenever an application opens a Kernel PlugIn handle, except for the specific configuration in which the `funcOpen_32_64` callback is used. This callback will hereby be referred to as the `funcOpen` callback. When writing your Kernel PlugIn driver, you can select whether to provide both of these callback types or just one of them - depending on how you plan to build the driver and how you expect it to be used. For example, if you are not planning to use the driver on a 64-bit operating system, you do not need to implement and set `funcOpen_32_64`.

In version 10.3.0 and above of WinDriver, the generated DriverWizard Kernel PlugIn projects, and the sample PCI Kernel PlugIn project (`kp_pci`), are written so as to support all types of user-mode and Kernel PlugIn configurations, including using a 32-bit application with a 64-bit Kernel PlugIn.

To modify code developed with earlier versions of WinDriver to afford the same support, follow these steps:

In the user-mode code, do the following:

- Separate the steps of opening a handle to the device and opening a handle to the Kernel PlugIn. In version 10.2.1 and below of WinDriver, the sample and generated code used the [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) functions (e.g., [WDC_PciDeviceOpen\(\)](#)) to open both of these handles. As explained above, this method cannot be used if you also wish to work with a 32-bit application and a 64-bit Kernel PlugIn driver. If your code uses this method, modify the call to [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) so that you pass NULL for the last two parameters - `pckPDriverName` and `pKPOpenData`.
- Use [WDC_KernelPlugInOpen\(\)](#) to open the handle to the Kernel PlugIn driver.

For the function's `hDev` parameter, pass the same handle whose address was passed as the `phDev` parameter of the device-open function.

** Attention**

You are passing hDev - WDC_DEVICE_HANDLE - and not &hDev - WDC_DEVICE_→HANDLE* - like in the first parameter of [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#).

For the pcKPDriverName parameter, pass the name of your Kernel PlugIn driver (as previously done for the equivalent device-open function parameter, if you used it to also open the Kernel PlugIn handle).

For the pKPOpenData parameter, pass the data you wish to pass down from the application to the Kernel PlugIn (similar to the device-open function parameter of the same name).

In the older WinDriver sample and generated code, the device handle was passed as the open-data. However, to reduce the work that would need to be done in the Kernel PlugIn to convert the 32-bit data to 64-bit, the newer code now passes only the necessary information to the Kernel PlugIn, we recommend that you do the same:

- Define the following types, to be used both by the user-mode and Kernel PlugIn code (in the sample and generated code, these types are defined in the xxx_lib.h file):

```
/* Device address description struct */
typedef struct {
    DWORD dwNumAddrSpaces; /* Total number of device address spaces */
    WDC_ADDR_DESC *pAddrDesc; /* Array of device address spaces information */
} PCI_DEV_ADDR_DESC;
```

- Allocate a variable of type PCI_DEV_ADDR_DESC and set its fields according to the information returned by the previous call to the [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) function. In the sample and generated code, this is done in the xxx_lib.c file as shown below:

```
PCI_DEV_ADDR_DESC devAddrDesc;
PWDC_DEVICE pDev;
...
dwStatus = WDC_PciDeviceOpen(&hDev, pDeviceInfo, pDevCtx, NULL, NULL, NULL);
...
pDev = hDev;
devAddrDesc.dwNumAddrSpaces = pDev->dwNumAddrSpaces;
devAddrDesc.pAddrDesc = pDev->pAddrDesc;
WDC_KernelPlugInOpen(hDev, KP_PCI_DRIVER_NAME, &devAddrDesc);
```

In the Kernel PlugIn driver code, do the following:

- Modify your funcOpen callback.

If you have changed the type of the open-data that is passed from the user mode to the driver, you need to adjust the implementation of your open-callback accordingly. If your existing code was based on the sample or generated Kernel PlugIn code in earlier versions of WinDriver, and you have selected to pass the address of a PCI_DEV_→ADDR_DESC struct as your Kernel PlugIn open-data modify your funcOpen callback as follows:

- Replace the PWDC_DEVICE hDev; definition with PCI_DEV_ADDR_DESC *pDevAddrDesc; and replace instances of WDC_DEVICE and pDev in the function with PCI_DEV_ADDR_DESC and pDevAddrDesc respectively.
- Remove the temp variable definition and replace the two COPY_FROM_USER calls (which use the temp variable) with the following call: COPY_FROM_USER(pDevAddrDesc, pOpenData, dwSize).
- As an additional unrelated fix to previous sample and generated WinDriver funcOpen implementations, move the call to the library initialization function (PCI_LibInit()) in the PCI sample / XXX_LibInit in the generated DriverWizard code - where XXX is the name of your driver project) to the beginning of the function - right after the variable declarations and before the first call to the Kernel PlugIn trace messages function (KP_PCI_→Trace() in the PCI sample /KP_XXX_Trace() in the wizard-generated code).
- Create your funcOpen_32_64 callback.

Copy your funcOpen callback function, and rename it. The renamed function will be modified to implement your funcOpen_32_64 callback - for opening a handle to a 64-bit Kernel PlugIn driver from a 32-bit application. In the WinDriver/samples/c/pci_diag/kp_pci/kp_pci.c sample, the funcOpen function is named [KP_PCI_Open\(\)](#), and the funcOpen_32_64 function is named [KP_PCI_Open_32_64\(\)](#). The generated Driver→Wizard Kernel PlugIn open functions are named in a similar manner, except that "PCI" is replaced with your selected driver project name.

** Attention**

If you only plan to support a 32-bit application and 64-bit Kernel Plugin configuration, you can skip this step, and in the next step modify your `KP_Open` callback to use it as your `funcOpen_32_64` callback, and then assign this callback to the `KP_INIT funcOpen_32_64` field, instead of to the `funcOpen` field.

However, to keep your code portable, it's best to implement both types of callbacks.

- Edit your Kernel Plugin code to handle the translation of the 32-bit data passed from the user mode, into 64-bit kernel data:
- Add a 32-bit definition of the Kernel Plugin open-data type.
- In your `funcOpen_32_64` callback, translate the 32-bit open-data received from the user mode, into the equivalent 64-bit type.

For example, if your original code was based on the sample or generated Kernel Plugin code in earlier versions of WinDriver, and you have selected to pass the address of a `PCI_DEV_ADDR_DESC` struct as your Kernel Plugin open-data modify the code as follows:

1. Add the following definitions at the beginning of the file:

```
#define PTR32 DWORD
typedef struct WDC_DEV_ADDR_DESC_32B {
    DWORD dwNumAddrSpaces; /* Total number of device address spaces */
    PTR32 pAddrDesc; /* Array of device address spaces information */
} WDC_DEV_ADDR_DESC_32B, *PWDC_DEV_ADDR_DESC_32B;
```

1. Modify your `funcOpen_32_64` callback as follows: Add the following definition:

```
WDC_DEV_ADDR_DESC_32B devAddrDesc_32;
```

Then add the following lines after the `kpOpenCall` field assignments, in order to copy to the kernel mode the 32-bit data that was received from the user mode:

```
/* Copy device information sent from a 32-bit user application */
COPY_FROM_USER(&devAddrDesc_32, pOpenData, sizeof(PCI_DEV_ADDR_DESC_32B));
```

Then replace the following line - `COPY_FROM_USER(pDevAddrDesc, pOpenData, dwSize)` - with these lines, in order to copy the 32-bit data into a 64-bit structure:

```
/* Copy the 32-bit data to a 64-bit struct */
pDevAddrDesc->dwNumAddrSpaces = devAddrDesc_32.dwNumAddrSpaces;
```

Replace the second `COPY_FROM_USER()` call - `COPY_FROM_USER(pAddrDesc, pDevAddrDesc->p->AddrDesc, dwSize)` - with the following:

```
COPY_FROM_USER(pAddrDesc, (PVOID) (KPTR) devAddrDesc_32.pAddrDesc, dwSize);
```

- Update your `KP_Init()` function.

Notify WinDriver of your `funcOpen_32_64` callback, by assigning your `funcOpen_32_64` callback to the `funcOpen_32_64` field of the `KP_INIT` struct that is initialized in `KP_Init()`:

```
kpInit->funcOpen_32_64 = <Your funcOpen_32_64 callback>;
```

For example, this is the assignment line from the updated `kp_pci.c` sample:

```
kpInit->funcOpen_32_64 = KP_PCI_Open_32_64;
```

** Attention**

If you select to only implement a `funcOpen_32_64` callback, and not to implement a standard `funcOpen` callback, remove the assignment to the `KP_INIT funcOpen` field in `KP_Init()`.

- Edit all remaining device-open type handling.

If you have changed the type of the open-data that is passed from the user mode to the driver, you need to edit the related type handling in your Kernel Plugin code. For example, if you previously passed `PWDC_DEVICE` as the open-data type, and you now pass `PCI_DEV_ADDR_DESC`, and your code was previously developed using the a WinDriver sample or wizard-generated Kernel Plugin driver, you would need to make the following additional changes:

- In [KP_Close](#), replace this line - `free(((PWDC_DEVICE)pDrvContext)->pAddrDesc)` - with the following, to free the correct type:

```
if (((PCI_DEV_ADDR_DESC *)pDrvContext)->pAddrDesc)
    free(((PCI_DEV_ADDR_DESC *)pDrvContext)->pAddrDesc);
```

- In [KP_IntAtIrql](#), replace this line - `PWDC_DEVICE pDev = (PWDC_DEVICE)pIntContext;` - with the following:

```
PCI_DEV_ADDR_DESC *pDevAddrDesc = (PCI_DEV_ADDR_DESC *)pIntContext;
```

and edit any remaining `pDev` instances in the function. For example, replace this line - `pAddrDesc = &p←Dev->pAddrDesc[INTCSR_ADDR_SPACE];` - with the following:
`pAddrDesc = &pDevAddrDesc->pAddrDesc[INTCSR_ADDR_SPACE];`

13.9. Install Your Kernel PlugIn Driver

13.9.1. Windows Kernel PlugIn Driver Installation

Verify that the WinDriver kernel module is installed: Before installing your Kernel PlugIn driver, verify that the WinDriver driver - `windrvrv<version>.sys/.o/.ko` in the newer WinDriver versions (e.g., `windrvrv1511.sys/.o/.ko`) - is installed, since the Kernel PlugIn module depends on the WinDriver module for its successful operation. You can run the Debug Monitor to verify that WinDriver is loaded.

Install your Kernel PlugIn driver:

- Remember to copy your Kernel PlugIn driver (`my_kp.sys/.kext/.o/.ko`) to the operating system's drivers/modules directory before attempting to install the driver.
- Don't forget to install your Kernel PlugIn driver before running your application: On Windows: Use the `wdreg` installation utility (or `wdreg_gui`/`wdreg_frontend` - depending on the WinDriver version and OS that you are using) to install the driver. To install a `my_kp.sys` driver, run this command:
`wdreg -name MY_KP install`

** Attention**

The Kernel PlugIn is not backward compatible. Therefore, when switching to a different version of WinDriver, you need to rebuild your Kernel PlugIn driver using the new version.

Driver installation on Windows requires administrator privileges.

The following steps can also be performed with the `wdreg_frontend.exe` GUI application. For additional information, refer to [16.3. The wdreg_frontend utility](#).

- Copy the driver file (`xxx.sys`) to the target platform's drivers directory: `windir%\system32\drivers` (e.g., `C:\WINDOWS\system32\drivers`).
- Register/load your driver, using the `wdreg.exe` or `wdreg_gui.exe` utility:

In the following instructions, `KP_NAME` stands for your Kernel PlugIn driver's name, without the `.sys` extension.

To install your driver, run this command:

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

Kernel PlugIn drivers are dynamically loadable - i.e., they can be loaded and unloaded without reboot. For additional information, refer to [16.2.3. Dynamically Loading/Unloading Your Kernel PlugIn Driver](#).

13.9.2. Linux Kernel PlugIn Driver Installation

On Linux, you do not need to copy the driver file, since the driver installation (`make install`) will handle this for you:

- Change directory to your Kernel PlugIn driver directory.

For example, when installing the sample KP_PCI driver, run

```
$ cd WinDriver/samples/c/pci_diag/kp_pci
```

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

```
$ cd <path>/kemode/
```

- Execute the following command to install your Kernel Plugin driver:

The following command must be executed with root privileges.

```
# make install
```

Kernel Plugin drivers are dynamically loadable - i.e., they can be loaded and unloaded without reboot. For additional information, refer to [16.4.1. Dynamically Loading/Unloading Your Kernel Plugin Driver](#).

13.10. FAQ

13.10.1. I would like to execute in the kernel some pieces of code written in languages other than C/C++ (Python/Java/C#/Visual Basic.NET), using the Kernel Plugin. Is it possible?

The Kernel Plugin is a kernel module, and therefore must be written in the C language. However, the user application that calls the Kernel Plugin can be of any language supported by WinDriver: C, C#.NET, Visual Basic.NET, Java, Python. When generating code in the DriverWizard in languages other than C with Kernel Plugin, the Kernel Plugin code will still be generated in C. This user application would communicate with the C-language Kernel Plugin.

If you select to implement such a design, in order to ensure the correct interaction between the user mode and Kernel Plugin applications, you will need to implement a file in your programming language that contains common definitions for the Kernel Plugin and the user mode applications - as done in the library header files of the sample and DriverWizard generated Kernel Plugin C projects (e.g. WinDriver\samples\pci_diag\pci_lib.h, used by the sample KP_PCI driver). WinDriver implements these files for you when you generate code with DriverWizard, and they should be generated in your project directory, for you to start as fast and easy as possible.

For further examples check: WinDriver/include/windrivr.h C header file and the corresponding WinDriver/samples/python/wdlib/windrivr.py (Python), WinDriver/src/wdapi_dotnet/windrivr.h (.NET), WinDriver/lib/wdapi_javaXXX.jar (Java) as examples of implementing the same code both in C and other supported/ languages.

13.10.2. How do I allocate locked memory in the kernel that can be used inside the interrupt handler?

WinDriver implements `malloc()` and `free()` in its Kernel Plugin library (kp_nt1511.lib/kp_linux1511.o_shipped), to which your Kernel Plugin code is linked.

These functions are implemented to allocate locked memory when called from the kernel mode, so you can use that memory in your interrupt handler as well.

13.10.3. How do I handle shared PCI interrupts in the Kernel Plugin?

PCI interrupts are normally handled at the kernel level. Therefore, the best way to handle PCI interrupts with WinDriver is using the Kernel Plugin feature, which lets you implement code directly at the kernel level. This is specifically true when handling interrupts that are shared between different hardware devices - a very common phenomenon in the case of PCI interrupts, which are by definition sharable.

The Kernel Plugin consists of two interrupt handler functions:

** Attention**

Replace “KP_” in the function names below with the name of your Kernel Plugin driver. For example, the Kernel Plugin interrupt handler functions for the sample KP_PCI driver are KP_PCI_IntAtIrql and KP_PCI_IntAtDpc.

- **KP_IntAtIrql** - this function is executed in the kernel at high IRQ (Interrupt Request) level immediately upon the detection of an interrupt. The function should contain the write/read commands for clearing the source of the interrupt (acknowledging the interrupt) and any additional high-priority interrupt-handling code.

If additional deferred kernel- or user-mode interrupt processing is required, **KP_IntAtIrql** should return TRUE, in which case the deferred **KP_IntAtDpc** routine (see below) will be executed once the high-level processing is completed. Otherwise, the function should return FALSE and **KP_IntAtDpc** (as well as any existing user-mode interrupt-handler routine) will not be executed. The generated and sample WinDriver Kernel Plugin projects, for example, schedule deferred interrupt processing once every five interrupts by implementing code that returns TRUE only for every fifth **KP_IntAtIrql** call.

- **KP_IntAtDpc** - this function is executed in the kernel at raised execution level, provided **KP_IntAtIrql** returned TRUE (see above), and should contain any lower-priority kernel interrupt handling that you wish to perform. The return value of this function determines how many times, if at all, the user-mode interrupt-handler routine (which can be set in the call to `WDC_InterruptEnable()` from the user-mode) will be executed once the control returns to the user mode.

To handle shared PCI interrupts with WinDriver, perform the following steps:

- Generate a Kernel Plugin project using WinDriver’s DriverWizard utility. When generating C code with the DriverWizard, you will be given the option to generate Kernel Plugin code - simply check the relevant check-box and proceed to generate the code. Alternatively, you can also use the generic WinDriver Kernel Plugin sample - KP_PCI(WinDriver\samples\c\pci_diag\kp_pci\kp_pci.c) - as the basis for your Kernel Plugin project. If you are developing a driver for a Xilinx PCI Express card with Bus Master DMA (BMD) design, you can use the sample Kernel Plugin driver for this design - KP_BMD(WinDriver\samples\c\xilinx\bmd_design\kp\kp_bmd.c) - or select to generate customized DriverWizard code for this design, including Kernel Plugin code.

The advantage of using the DriverWizard is that the generated code will utilize the specific device configuration information detected for your device, as well as any hardware-specific information that you define with the DriverWizard. When generating Kernel Plugin code for handling PCI interrupts, in the DriverWizard’s Registers tab define the registers that you wish to access upon the detection of an interrupt, and then in the Interrupts tab assign the registers read/write commands that you wish to perform at high IRQ level (**KP_IntAtIrql**) to the interrupt. The exact commands for acknowledging the interrupt are hardware-specific and you should therefore consult your hardware specification to determine the correct commands to set.

- The correct way to handle PCI interrupts with the Kernel Plugin, and shared interrupts in particular, is to include a command in **KP_IntAtIrql** that reads information from the hardware (normally you would read from the interrupt status register - INTCSR) in order to determine if your hardware generated the interrupt. (You can define a relevant read command in the DriverWizard before generating your Kernel Plugin code - refer to step #1 above - or manually modify the generated/sample code to add such a command.) If the interrupt was indeed generated by your hardware, the function can set the value of the `fIsMyInterrupt` parameter to TRUE in order to accept control of the interrupt, and then proceed to write/read the relevant hardware register(s) in order to acknowledge the interrupt, and either return TRUE to perform additional deferred processing or return FALSE if no such processing is required (see above). If, however, you determine that the interrupt was not generated by your hardware, `fIsMyInterrupt` should be set to FALSE, in order to ensure that the interrupt will be passed on to other drivers in the system, and **KP_IntAtIrql** should return FALSE. Important to mention that there is no real harm in setting `fIsMyInterrupt` to FALSE even if the interrupt belongs to you, as done by default in the generated and sample WinDriver code, since other drivers in the system, assuming they were implemented correctly, should not attempt to handle the interrupt if it was not generated by their hardware.

The portion of the code that performs the check whether your hardware generated the interrupt is based on hardware-specific logic that cannot be defined in the DriverWizard. You will therefore need to modify the generated/sample **KP_IntAtIrql** implementation and add a relevant “if” clause to ensure that you do not accept control of

an interrupt that was not generated by your hardware and do not attempt to clear the source of such an interrupt in the hardware.

Following is a sample `KP_IntAtIrql` implementation. The code reads the INTCSR memory register (defined elsewhere in the code) and only proceeds to accept control of the interrupt and acknowledge it if the value read from the INTCSR is 0xFF (which serves as an indication in this sample that our hardware generated the interrupt). The interrupt in this sample is acknowledged by writing back to the INTCSR the value that was read from it.

```
BOOL __cdecl KP_XXX_IntAtIrql(PVOID pIntContext, BOOL *pfIsMyInterrupt)
{
    XXX_HANDLE hXXX = (XXX_HANDLE) pIntContext;
    DWORD data = 0;
    PVOID pData = NULL;
    DWORD addrSpace;
    WD_ITEMS * pItem;
    addrSpace = XXX_INTCSR_SPACE;
    pItem = &hXXX->cardReg.Card.Item[hXXX->addrDesc[addrSpace].index];
    pData = (DWORD*)pItem->I.Mem.dwTransAddr;
    (DWORD)pData += XXX_INTCSR_OFFSET;
    data = dtoh32(*((DWORD*)pData));
    if (data == 0xFF)
        /* The interrupt was generated by our hardware */

    {
        /* Write 0x0 to INTCSR to acknowledge the interrupt */
        *((DWORD*)pData) = dtoh32(data);

        /* Accept control of the interrupt */
        *pfIsMyInterrupt = TRUE;

        /* Schedule deferred interrupt processing (XXX_IntAtDpc) */
        return TRUE;
    }
    else
    {
        /* (Do not acknowledge the interrupt) */
        /* Do not accept control of the interrupt */
        *pfIsMyInterrupt = FALSE;

        /* Do not schedule deferred interrupt processing */
        return FALSE;
    }
}
```

13.10.4. What is pIntContext in the Kernel Plugin interrupt functions?

`pIntContext` is private context data that is passed from `KP_IntEnable` to the Kernel Plugin interrupt handler functions - `KP_IntAtIrql()` and `KP_IntAtDpc()`.

13.10.5. I need to call WD_Transfer() in the Kernel Plugin. From where do I get hWD to pass to these functions?

Jungo strongly recommends using the high-level WDC (PCI) API instead of using the low-level API.

To obtain the handle to WinDriver's kernel module (`hWD`) from the Kernel Plugin, you can call `WD_Open()` directly from the Kernel Plugin.

For PCI/ISA, you can also use the `hWD` handle, which is passed from the user mode via `WD_KernelPluginOpen()`, and received as the second argument of the Kernel Plugin callback function `KP_Open()`. Alternatively, you can also pass the handle from the user mode to the Kernel Plugin using the `pData` field of the `WD_KERNEL_PLUGIN_CALL` struct, which is used in the call to `WD_KernelPluginCall()` in the user mode and results in a callback to `KP_Call()` in the Kernel Plugin.

After obtaining the handle to WinDriver, the call to `WD_Transfer()` (or any other WinDriver API that receives a handle to WinDriver as a parameter) is the same as in the user mode.

13.10.6. A restriction in KP_IntAtIrql is to use only non-pageable memory. What does this mean?

Variables defined in the Kernel Plugin project (such as global and local variables) are non paged. WinDriver also implements `malloc()` to allocate non-paged memory when used from within the kernel.

If you are using a pointer to the memory in the user mode from within the Kernel PlugIn, you need to make a copy of its contents (allocate memory and copy the data to the non-paged memory) in order to access it from your `KP_IntAtIrq` function, in order to ensure safe access to the data at all times.

13.10.7. How do I call `WD_Transfer()` in the Kernel PlugIn interrupt handler?

You can call `WD_Transfer()` from within the `KP_IntAtIrq` or `KP_IntAtDpc` functions (replace “KP” in the function names with your Kernel PlugIn driver name - e.g., “KP_PCI”) in order to access the hardware.

** Attention**

You can also access the memory or I/O directly from within the Kernel PlugIn interrupt functions. For direct memory access, use the kernel mapping of the memory, returned from `WD_CardRegister()` in `cardReg.Card.Item[i].I.Mem.pTransAddr`.

When calling `WD_Transfer()` you will need to pass as the first argument a handle to WinDriver (`hWD`). You can refer to the FAQ questions in this chapter to find out how to obtain a handle to WinDriver from within the Kernel PlugIn.

As specified in the aforementioned FAQ questions, you can store the handle to WinDriver in a global Kernel PlugIn variable (recommended), or pass it from one function to another. Below is an example of passing a handle to WinDriver from `KP_Open()` to `KP_IntAtIrq`:

Add the following line to `KP_Open()`

```
ppDrvContext = (PVOID) hWD;
```

Add the following line to `KP_IntEnable()`

```
pIntContext = pDrvContext;
```

You can now use `WD_Transfer()` to access memory/IO from within `KP_IntAtIrq`. For example (IO access):

```
HANDLE hWD = (HANDLE) pIntContext;
WD_TRANSFER trans;
BZERO(trans);
trans.cmdTrans = WP_BYTE;
trans.dwPort = 0x378;
trans.Data.Byte = 0x65;
WD_Transfer(hWD, &trans);
```

This will write 0x65 to port 0x378 upon interrupt. The `hWD` handle is passed from `KP_Open()` to `KP_IntEnable()` to `KP_IntAtIrq` via the context.

You can also view the generated DriverWizard Kernel PlugIn code for an example of calling `WD_MultiTransfer()` from `KP_IntAtIrq`, provided you have used the DriverWizard to define and assign the register/s for the interrupt acknowledgment before generating the code.

13.10.8. How do I share a memory buffer between Kernel PlugIn and user-mode projects for DMA or other purposes?

From your user application, use the `WDS_SharedBufferAlloc()` function to allocate a shared memory buffer, which can be used both from a user-mode application and a Kernel PlugIn driver.

Pass the returned `ppKerBuf->pKernelAddr` value from the above function to the Kernel PlugIn using the `pData` parameter of the function `WDC_CallKerPlug()`. Note that `WDS_SharedBufferAlloc()` is part of the Server APIs of WinDriver, these APIs require a special license.

** Note**

If you would like to get more information about Server APIs licensing, please send an e-mail to WinDriver@jungo.com, and our team will assist you.

13.10.9. If I write a new function in my SYS Kernel PlugIn driver, must it also be declared with `__cdecl`?

No. Only the callbacks used by WinDriver need to be declared as `__cdecl`.

Chapter 14

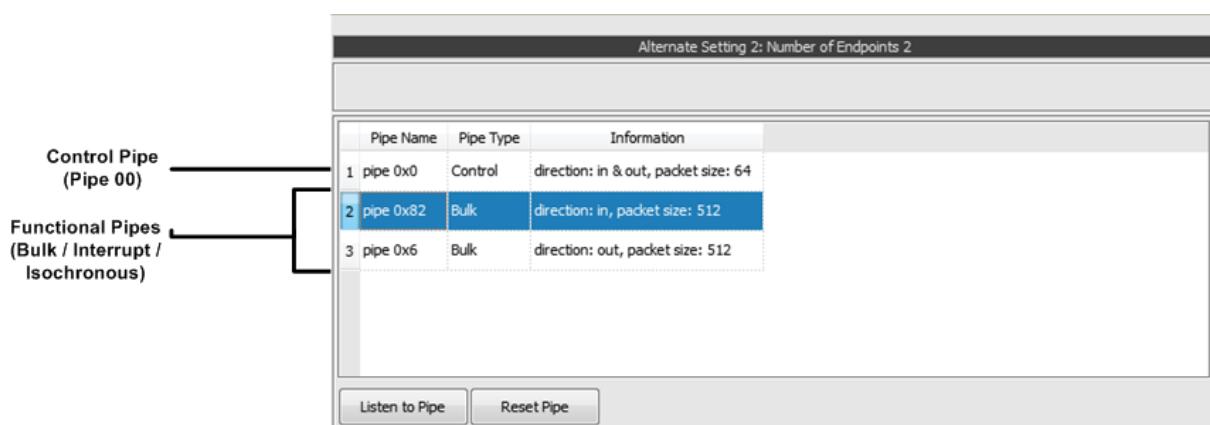
USB Advanced Features

14.1. USB Control Transfers

The USB standard supports two kinds of data exchange between the host and the device - control exchange and functional data exchange. The WinDriver APIs enable you to implement both control and functional data transfers.

The figure below demonstrates how a device's pipes are displayed in the DriverWizard utility, which enables you to perform transfers from a GUI environment.

USB Data Exchange



14.2. USB Control Transfers Overview

14.2.1. Control Data Exchange

USB control exchange is used to determine device identification and configuration requirements, and to configure a device; it can also be used for other device-specific purposes, including control of other pipes on the device.

Control exchange takes place via a control pipe - the default pipe 0, which always exists. The control transfer consists of a setup stage (in which a setup packet is sent from the host to the device), an optional data stage and a status stage.

14.2.2. More About the Control Transfer

The control transaction always begins with a setup stage. The setup stage is followed by zero or more control data transactions (data stage) that carry the specific information for the requested operation, and finally a status transaction completes the control transfer by returning the status to the host.

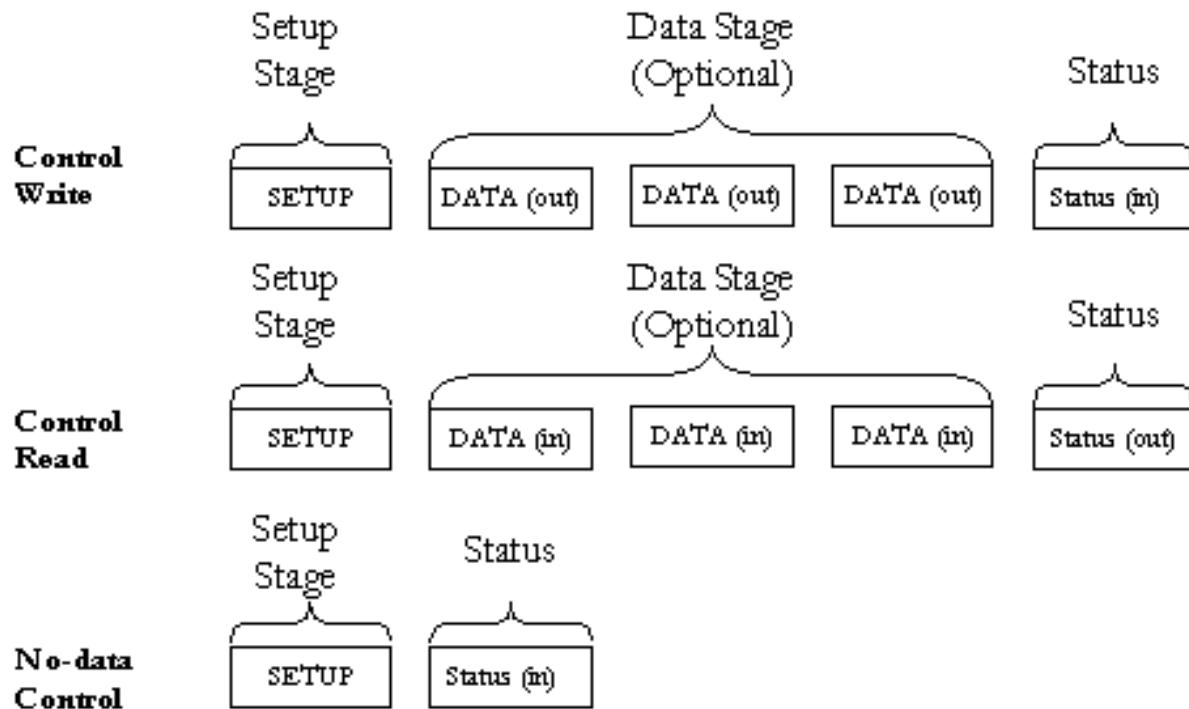
During the setup stage, an 8-byte setup packet is used to transmit information to the control endpoint of the device (endpoint 0). The setup packet's format is defined by the USB specification.

A control transfer can be a read transaction or a write transaction. In a read transaction the setup packet indicates the characteristics and amount of data to be read from the device. In a write transaction the setup packet contains the command sent (written) to the device and the number of control data bytes that will be sent to the device in the data stage.

Refer to the figure provided below (and taken from the USB specification) for a sequence of read and write transactions:

** Attention**

'(in)' indicates data flow from the device to the host. '(out)' indicates data flow from the host to the device.

USB Read and Write**14.2.3. The Setup Packet**

The setup packets (combined with the control data stage and the status stage) are used to configure and send commands to the device. USB requests such as these are sent from the host to the device, using setup packets. The USB device is required to respond properly to these requests. In addition, each vendor may define device-specific setup packets to perform device-specific operations. The standard setup packets (standard USB device requests) are detailed below. The vendor's device-specific setup packets are detailed in the vendor's data book for each USB device.

14.2.4. USB Setup Packet Format

The table below shows the format of the USB setup packet. For more information, please refer to the USB specification at <http://www.usb.org>.

Byte	Field	Description
0	bmRequest Type	Bit 7: Request direction (0=Host to device - Out, 1=Device to host- In). Bits 5-6: Request type (0=standard, 1=class, 2=vendor, 3=reserved). Bits 0-4: Recipient (0=device, 1=interface, 2=endpoint,3=other).
1	bRequest	The actual request (see the Standard Device Request Codes table below)
2	wValueL	A word-size value that varies according to the request. For example, in the CLEAR_FEATURE request the value is used to select the feature. In the GET_DESCRIPTOR request the value indicates the descriptor type and in the SET_ADDRESS request the value contains the device address.

Byte	Field	Description
3	wValueH	The upper byte of the Value word.
4	wIndexL	A word-size value that varies according to the request. The index is generally used to specify an endpoint or an interface.
5	wIndexH	The upper byte of the Index word.
6	wLengthL	A word-size value that indicates the number of bytes to be transferred if there is a data stage.
7	wLengthH	The upper byte of the Length word.

14.2.5. Standard Device Request Codes

The table below shows the standard device request codes.

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

14.2.6. Setup Packet Example

This example of a standard USB device request illustrates the setup packet format and its fields. The setup packet is in Hex format.

The following setup packet is for a control read transaction that retrieves the device descriptor from the USB device. The device descriptor includes information such as USB standard revision, vendor ID and product ID.

GET_DESCRIPTOR (Device) Setup Packet							
80	06	00	01	00	00	12	00

Setup packet meaning:

Byte	Field	Value	Description
0	bmRequest Type	80	8h=1000b bit 7=1 -> direction of data is from device to host. 0h=0000b bits 0..1=00 -> the recipient is the device.
1	bRequest	06	The Request is GET_DESCRIPTOR.
2	wValueL	00	
3	wValueH	01	The descriptor type is device (values defined in USB spec).
4	wIndexL	00	The index is not relevant in this setup packet since there is only one device descriptor.
5	wIndexH	00	

Byte	Field	Value	Description
6	wLengthL	12	Length of the data to be retrieved: 18(12h) bytes (this is the length of the device descriptor).
7	wLengthH	00	

In response, the device sends the device descriptor data. A device descriptor of the Cypress EZ-USB Integrated Circuit is provided as an example:

Byte No.	Content
0	0x12
1	0x01
2	0x00
3	0x01
4	0xff
5	0xff
6	0xff
7	0x40
8	0x47
9	0x05
10	0x80
11	0x00
12	0x01
13	0x00
14	0x00
15	0x00
16	0x00
17	0x01

As defined in the USB specification, byte 0 indicates the length of the descriptor, bytes 2-3 contain the USB specification release number, byte 7 is the maximum packet size for the control endpoint (endpoint 0), bytes 8-9 are the vendor ID, bytes 10-11 are the product ID, etc.

14.3. Performing Control Transfers with WinDriver

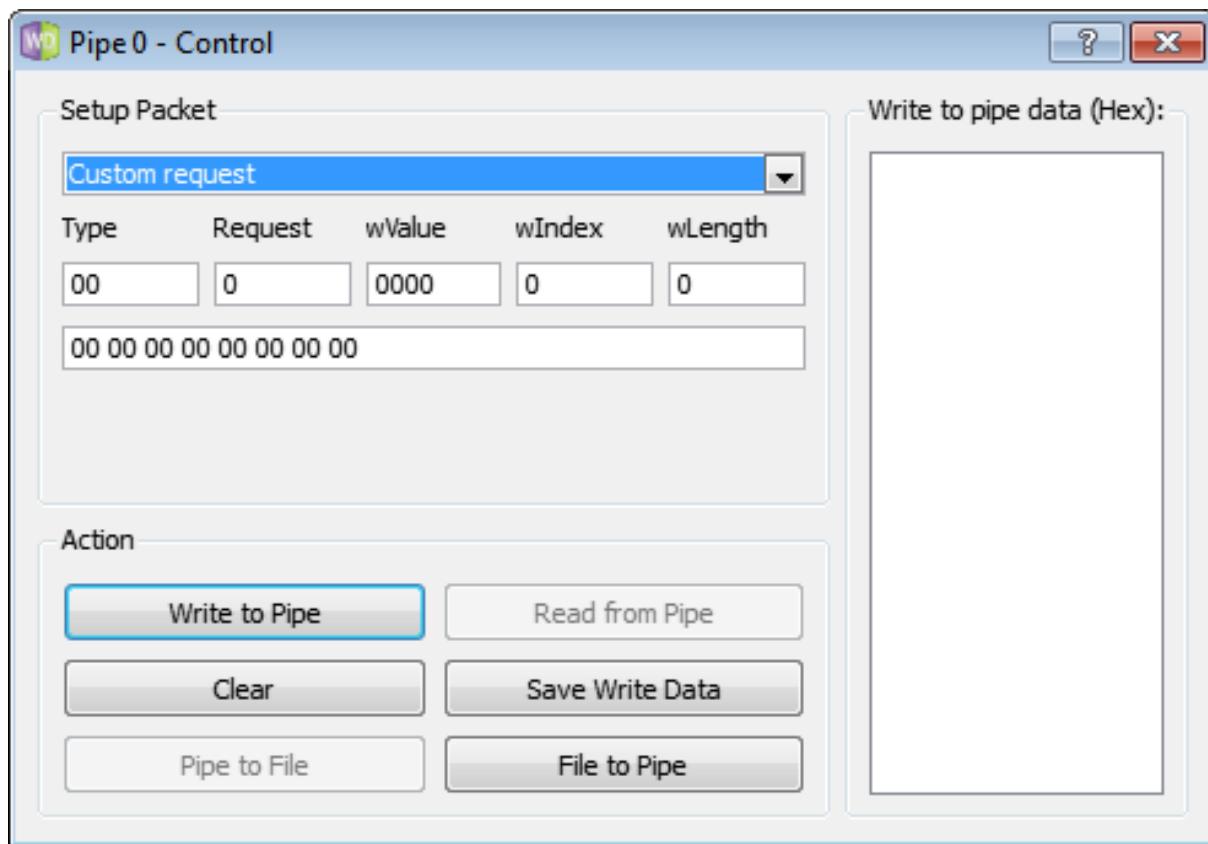
WinDriver allows you to easily send and receive control transfers on the control pipe (pipe0), while using Driver Wizard to test your device. You can either use the API generated by DriverWizard for your hardware, or directly call the WinDriver [WDU_Transfer\(\)](#) function from within your application.

14.3.1. Control Transfers with DriverWizard

- Choose Pipe 0x0 and click the Read / Write button.
- You can either enter a custom setup packet, or use a standard USB request.

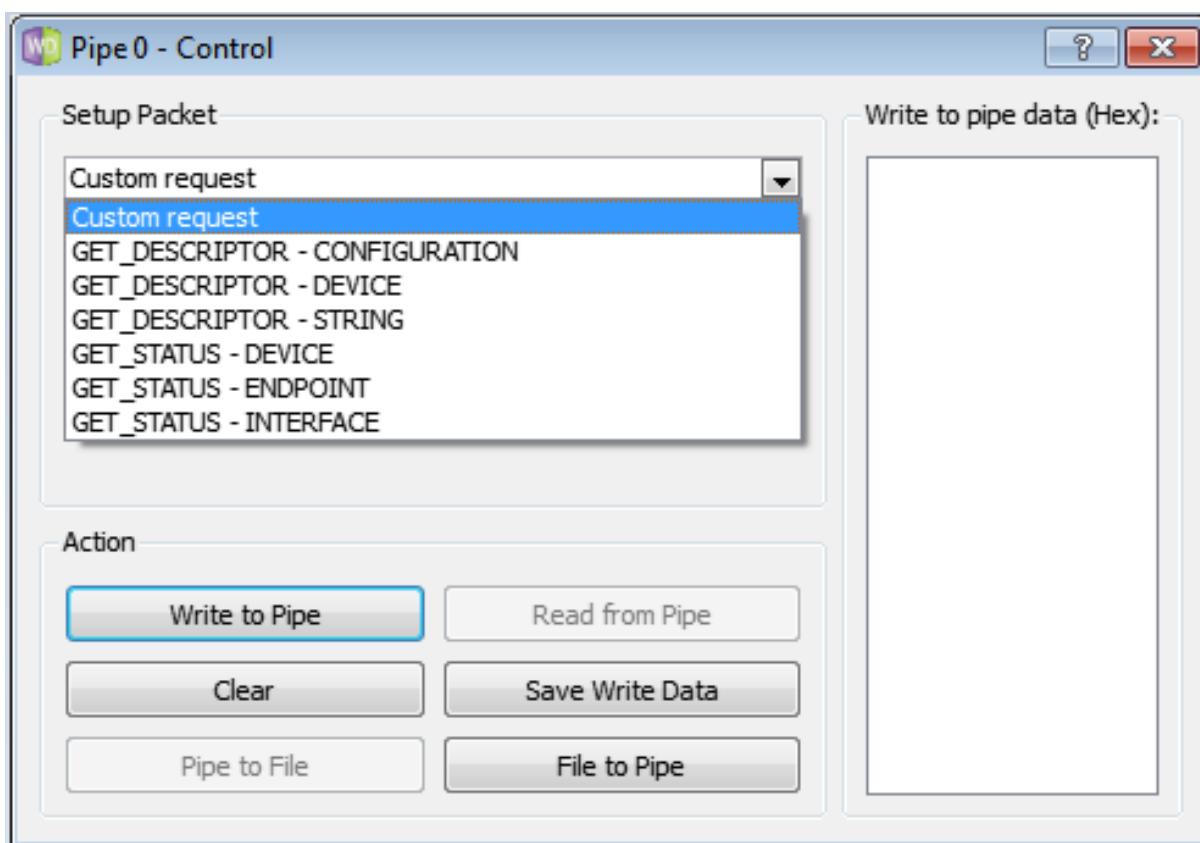
For a custom request: enter the required setup packet fields. For a write transaction that includes a data stage, enter the data in the Write to pipe data (Hex) field. Click Read From Pipe or Write To Pipe according to the required transaction.

Custom Request



For a standard USB request: select a USB request from the requests list, which includes requests such as GET_DESCRIPTOR CONFIGURATION, GET_DESCRIPTOR DEVICE, GET_STATUS DEVICE, etc. The description of the selected request will be displayed in the Request Description box on the right hand of the dialogue window.

Request List



- The results of the transfer, such as the data that was read or a relevant error, are displayed in DriverWizard's Log window. Below you can see the contents of the Log window after a successful GET_DESCRIPTOR DEVICE request.

USB Request Log



14.3.2. Control Transfers with WinDriver API

To perform a read or write transaction on the control pipe, you can either use the API generated by DriverWizard for your hardware, or directly call the WinDriver [WDU_Transfer] ([WDU_Transfer](#)) function from within your application. Fill the setup packet in the BYTE SetupPacket[8] array and call these functions to send setup packets on the control pipe (pipe 0) and to retrieve control and status data from the device.

The following sample demonstrates how to fill the SetupPacket[8] variable with a GET_DESCRIPTOR setup packet:

```
setupPacket[0] = 0x80; /* BmRequestType */
setupPacket[1] = 0x6; /* bRequest [0x6 == GET_DESCRIPTOR] */
setupPacket[2] = 0; /* wValue */
setupPacket[3] = 0x1; /* wValue [Descriptor Type: 0x1 == DEVICE] */
setupPacket[4] = 0; /* wIndex */
```

```
setupPacket[5] = 0;      /* wIndex */
setupPacket[6] = 0x12;    /* wLength [Size for the returned buffer] */
setupPacket[7] = 0;      /* wLength */
```

The following sample demonstrates how to send a setup packet to the control pipe (a GET instruction; the device will return the information requested in the pBuffer variable):

```
WDU_TransferDefaultPipe(hDev, TRUE, 0, pBuf, dwSize, bytes_transferred, &setupPacket[0], 10000);
```

The following sample demonstrates how to send a setup packet to the control pipe (a SET instruction):

```
WDU_TransferDefaultPipe(hDev, FALSE, 0, NULL, 0, bytes_transferred, &setupPacket[0], 10000);
```

Refer for further information to [WDU_Transfer\(\)](#).

14.4. Functional USB Data Transfers

14.4.1. Functional USB Data Transfers Overview

Functional USB data exchange is used to move data to and from the device. There are three types of USB data transfers: Bulk, Interrupt and Isochronous. Functional USB data transfers can be implemented using two alternative methods: single-blocking transfers and streaming transfers, both supported by WinDriver, as explained in the following sections. The generated DriverWizard USB code and the generic WinDriver/util/usb_diag.exe utility (source code located under the WinDriver/samples/c/usb_diag directory) enable the user to select which type of transfer to perform.

14.4.2. Single-Blocking Transfers

14.4.2.1. Performing Single-Blocking Transfers with WinDriver

WinDriver's [WDU_Transfer\(\)](#) function, and the [WDU_TransferBulk\(\)](#), [WDU_TransferIsoch\(\)](#), and [WDU_TransferInterrupt\(\)](#) convenience functions enable you to easily implement single-blocking USB data transfers. You can also perform single-blocking transfers using the DriverWizard utility (which uses the [WDU_Transfer\(\)](#) function).

14.4.3. Streaming Data Transfers

In the streaming USB data transfer scheme, data is continuously streamed between the host and the device, using internal buffers allocated by the host driver - "streams".

Stream transfers allow for a sequential data flow between the host and the device, and can be used to reduce single-blocking transfer overhead, which may occur as a result of multiple function calls and context switches between user and kernel modes. This is especially relevant for devices with small data buffers, which might, for example, overwrite data before the host is able to read it, due to a gap in the data flow between the host and device.

14.4.3.1. Performing Streaming with WinDriver

WinDriver's [WDU_StreamOpen\(\)](#) / [WDU_StreamClose\(\)](#) / [WDU_StreamStart\(\)](#) / [WDU_StreamStop\(\)](#) / [WDU_StreamFlush\(\)](#) / [WDU_StreamRead\(\)](#) / [WDU_StreamWrite\(\)](#) / [WDU_StreamGetStatus\(\)](#) functions enable you to implement USB streaming data transfers.

** Attention**

These functions are currently supported on Windows.

To begin performing stream transfers, call the [WDU_StreamOpen\(\)](#) function. When this function is called, WinDriver creates a new stream object for the specified data pipe. You can open a stream for any pipe except for the control pipe (pipe 0). The stream's data transfer direction- read/write - is derived from the direction of its pipe.

WinDriver supports both blocking and non-blocking stream transfers. The open function's fBlocking parameter indicates which type of transfer to perform (see explanation below). Streams that perform blocking transfers will

henceforth be referred to as "blocking streams", and streams that perform non-blocking transfers will be referred to as "non-blocking streams". The function's `dwRxTxTimeout` parameter indicates the desired timeout period for transfers between the stream and the device.

After opening a stream, call [WDU_StreamStart\(\)](#) to begin data transfers between the stream's data buffer and the device.

In the case of a read stream, the driver will constantly read data from the device into the stream's buffer, in blocks of a pre-defined size (as set in the `dwRxSize` parameter of the [WDU_StreamOpen\(\)](#) function). In the case of a write stream, the driver will constantly check for data in the stream's data buffer and write any data that is found to the device.

To read data from a read stream to the user-mode host application, call [WDU_StreamRead\(\)](#).

In case of a blocking stream, the read function blocks until the entire amount of data requested by the application is transferred from the stream to the application, or until the stream's attempt to read data from the device times out. In the case of a non-blocking stream, the function transfers to the application as much of the requested data as possible, subject to the amount of data currently available in the stream's data buffer, and returns immediately.

To write data from the user-mode host application to a write the stream, call [WDU_StreamWrite\(\)](#).

In case of a blocking stream, the function blocks until the entire data is written to the stream, or until the stream's attempt to write data to the device times out. In the case of a non-blocking stream, the function writes as much of the write data as currently possible to the stream, and returns immediately.

For both blocking and non-blocking transfers, the read/write function returns the amount of bytes actually transferred between the stream and the calling application within an output parameter `-pdwBytesRead/pdwBytesWritten`.

You can flush an active stream at any time by calling the [WDU_StreamFlush\(\)](#) function, which writes the entire contents of the stream's data buffer to the device (for a write stream), and blocks until all pending I/O for the stream is handled. You can flush both blocking and non-blocking streams.

You can call [WDU_StreamGetStatus\(\)](#) for any open stream in order to get the stream's current status information.

To stop the data streaming between an active stream and the device, call [WDU_StreamStop\(\)](#). In the case of a write stream, the function flushes the stream - i.e., writes its contents to the device - before stopping it. An open stream can be stopped and restarted at any time until it is closed.

To close an open stream, call [WDU_StreamClose\(\)](#). The function stops the stream, including flushing its data to the device (in the case of a write stream), before closing it.

** Attention**

Each call to [WDU_StreamOpen\(\)](#) must have a matching call to [WDU_StreamClose\(\)](#) later on in the code in order to perform the necessary cleanup.

14.5. FAQ

14.5.1. Buffer Overrun Error: `WDU_Transfer()` sometimes returns the `0xC000000C` error code. What does this error code mean? How do I solve this problem?

The `0xC000000C` error code, is defined in `windrvr.h` as [WD_USBD_STATUS_BUFFER_OVERRUN](#).

The `WD_USBD_XXX` status codes returned by WinDriver (see `windrvr.h`) comply with the URB status codes returned by the low-level USB stack driver (e.g., URB code `0XC000000CL` - `WD_USBD_STATUS_BUFFER_OVERRUN`). You can refer to the Debug Monitor log to see the URB and IRP values returned from the stack drivers.

For Windows, the URB and IRP codes can be found in the Windows Driver Kit (WDK) under the `inc\` directory. The URB status codes can be found in the `usbdi.h` file or the `usb.h` file (depending on the OS). The IRP status codes can be found in the `ntstatus.h` file. For Linux, WinDriver translates the error codes returned from the stack driver into equivalent USBD errors.

For information regarding the specific error you received and when it might occur, review the operating system's documentation.

The USBD_STATUS_BUFFER_OVERRUN error code (0xC000000C) is set by the USB stack drivers when the device transfers more data than requested by the host.

There are two possible solutions for this buffer overrun problem:

- Try setting the buffer sizes in the calls to [WDU_Transfer\(\)](#) in your code to multiples of the maximum packet size. For example, if the maximum packet size is 64 bytes, use buffer sizes that are multiples of 64 (64 bytes, 128 bytes, etc.).
- Define a protocol between the device and device driver, making sure that the device does not transfer more data than requested. When you have access to the device firmware code, this solution is recommended.

** Recommendation**

Recheck your firmware and the hardware specification to verify that you are implementing the communication with the device correctly. It is also recommended to use a USB bus analyzer to determine what is happening on the bus.

14.5.2. How do I extract the string descriptors contained in the Device and Configuration descriptor tables?

You can use WinDriver's [WDU_GetStringDesc\(\)](#) function to get the desired string descriptors.

14.5.3. How do I detect that a USB device has been plugged in or disconnected?

Use [WDU_Init\(\)](#) to register to listen for the notifications you are interested in.

14.5.4. How do I setup the transfer buffer to send a null data packet through the control pipe?

You should set the pBuffer parameter of the [WDU_Transfer\(\)](#) function to `NULL` and set the `dwBufferSize` parameter to 0.

14.5.5. Can I write a driver for a USB hub or a USB Host Controller card using WinDriver?

No. Windriver USB is designed for writing drivers for USB devices (USB client drivers). It cannot be used to write a USB hub or a USB Host Controller driver.

14.5.6. Does WinDriver USB support isochronous streaming mode?

Yes. WinDriver provides [WDU_Stream](#) functions for performing streaming USB data transfers on Windows.

Chapter 15

Distributing Your Driver

Read this chapter in the final stages of driver development. It will guide you in preparing your driver for distribution. The recommended way to build and pack a WinDriver based driver today is using our CMake+CPack packaging methods as described in [15.2.4. Redistribute Your WinDriver-based package as a self-extracting EXE](#) and [15.3.5. Redistribute Your WinDriver-based package as a self-extracting SH \(STGZ\)](#) (depending on your operating system) as this method automates many steps in the redistribution process. If you wish to manually create the distribution package, follow [15.2. Windows Driver Distribution](#) or [15.3. Linux Driver Distribution](#).

15.1. Getting a Valid WinDriver License

Before distributing your driver you must purchase a WinDriver license.

Then install the registered version of WinDriver on your development machine by following the installation instructions. If you have already installed an evaluation version of WinDriver, you can jump directly to the installation steps for registered users to activate your license.

** Note**

If you wish to distribute the developed driver with your own driver development kit providing your customers with APIs, or if your driver will eventually be part of a development product or environment, please send an e-mail to sales@jungo.com so our team could give you some additional information which concerns such distribution method.

15.2. Windows Driver Distribution

** Attention**

All references to wdreg in this section can be replaced with wdreg_gui, which offers the same functionality as wdreg but displays GUI messages instead of console-mode messages.

If you have renamed the WinDriver kernel module (`windrivr1511.sys`), replace the relevant `windrivr1511` references with the name of your driver, and replace references to the `WinDriver\redist` directory with the path to the directory that contains your modified installation files.

For example, when using the generated DriverWizard renamed driver files for your driver project, you can replace references to the `WinDriver\redist` directory with references to the generated `xxx_\installation\redist` directory (where `xxx` is the name of your generated driver project).

Note also the option to simplify the installation using the generated DriverWizard `xxx_install.bat` script and the copies of the `WinDriver\util` installation files in the generated `xxx_\installation\redist` directory.

If you have created new INF and/or catalog files for your driver, replace the references to the original WinDriver INF files and/or to the `windrivr1511.cat` catalog file with the names of your new files.

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 `windrivr1511.sys` and `windrivr1511.inf`, installing the specific INF file for your device, 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.

It is also important to mention that all OS references in this document are applicable only to WinDriver versions that officially support these operating systems.

** Attention**

The `windrvr<version>.sys`, `windrvr<version>.inf`, and `windrvr<version>.cat` files, mentioned in this chapter, can be found in the `WinDriver\redist` directory on the development PC. `wdreg.exe` / `wdreg_gui.exe` and `difxapi.dll` can be found in the `WinDriver\util` directory. The source code of the `wdreg` utility is found in the `WinDriver\samples\c\wdreg` directory.

15.2.1. Preparing the Distribution Package

Prepare a distribution package that includes the following files.

** Attention**

If you wish to distribute drivers for both 32-bit and 64-bit target platforms, you must prepare separate distribution packages for each platform. The required files for each package are provided in the `WinDriver` installation directory for the respective platform.

- Your hardware-control application/DLL.
- `windrvr1511.sys`. Get this file from the `WinDriver\redist` directory of the `WinDriver` package.
- `windrvr1511.inf`. Get this file from the `WinDriver\redist` directory of the `WinDriver` package.
- `windrvr1511.cat` Get this file from the `WinDriver\redist` directory of the `WinDriver` package.
- `wdapi1511.dll` (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms) or `wdapi1511_32.dll` (for distribution of 32-bit binaries to 64-bit platforms). Get this file from the `WinDriver\redist` directory of the `WinDriver` package.
- If your user application is a C#.NET/VB.NET application/DLL you must also add `wdapi_dotnet1511.dll` (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms) or `wdapi_dotnet1511_32.dll` (for distribution of 32-bit binaries to 64-bit platforms). Get this file from the `WinDriver\lib` directory of the `WinDriver` package, or from the Visual Studio project's Release directory.
- If your user application is a Java application/DLL you must also add `wdapi_java1511.dll` (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms). You should also add `wdapi_java1511.jar` to your package. Get those files from the `WinDriver\lib` directory of the `WinDriver` package (specifically from its subdirectory relevant to your platform).
- `difxapi.dll` (required by the `wdreg.exe` utility). Get this file from the `WinDriver\util` directory of the `WinDriver` package.
- An INF file for your device. You can generate this file with DriverWizard.
- If you have created a Kernel PlugIn driver: Your Kernel PlugIn driver - <KP driver name>.sys.

15.2.2. Installing Your Driver on the Target Computer

Driver installation on Windows requires administrator privileges.

Follow the instructions below in the order specified to properly install your driver on the target computer:

Preliminary Steps:

To successfully install your driver, make sure that there are no open handles to the WinDriver service (windrvr1511.sys or your renamed driver), and that there are no connected and enabled Plug-and-Play devices that are registered with this service. If the service is being used, attempts to install the new driver using wdreg will fail. This is relevant, for example, when upgrading from an earlier version of the driver that uses the same driver name. You can disable or uninstall connected devices from the Device Manager (**Properties | Disable/Uninstall**) or using wdreg, or otherwise physically disconnect the device(s) from the PC.

This includes closing any applications that may be using the driver, uninstalling your old Kernel PlugIndriver (if you had created such a driver) and either disabling, uninstalling, or physically disconnecting any device that is registered to work with the WinDriver service:

```
wdreg -name OLD_KP uninstall
```

** Attention**

Since version 11.9.0 of WinDriver, the default driver module name includes the WinDriver version, so if you do not rename the driver to a previously-used name there should not be conflicts with older drivers.

Install WinDriver's kernel module:

- Copy windrvr1511.sys, windrvr1511.inf, and windrvr1511.cat to the same directory.

windrvr1511.cat contains the driver's Authenticode digital signature. To maintain the signature's validity this file must be found in the same installation directory as the windrvr1511.inf file. If you select to distribute the catalog and INF files in different directories, or make any changes to these files or to any other files referred to by the catalog file (such as windrvr1511.sys), you will need to do either of the following:

- Create a new catalog file and re-sign the driver using this file.
- Comment-out or remove the following line in the windrvr1511.inf file:

```
CatalogFile=windrvr1511.cat
```

and do not include the catalog file in your driver distribution. However, note that this option invalidates the driver's digital signature.

- Use the utility wdreg.exe to install WinDriver's kernel module on the target computer:

```
wdreg -inf <path to windrvr1511.inf> install
```

For example, if windrvr1511.inf and windrvr1511.sys are in the d:\MyDevice directory on the target computer, the command should be:

```
wdreg -inf d:\MyDevice\windrvr1511.inf install
```

You can find the executable of wdreg in the WinDriver package under the WinDriver\util directory:

** Attention**

wdreg is dependent on devcon.exe. 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.

When distributing your driver, you should attempt to ensure that the installation does not overwrite a newer version of windrvr1511.sys with an older version of the file in Windows drivers directory (windir%\system32\drivers). For example, by configuring 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. The provided windrvr1511.inf file uses the COPYFLG←_NO_VERSION_DIALOG directive, which is designed to avoid overwriting a file in the destination directory with the source file if the existing file is newer than the source file. There is also a similar COPYFLG_OVERWRITE←_ONLY_INF directive that is designed to ensure that the source file is copied to the destination directory only if the destination file is superseded by a newer version. Note, however, that both of these INF directives are irrelevant to digitally signed drivers. As explained in the Microsoft INF CopyFiles Directive documentation -

<https://msdn.microsoft.com/en-us/library/ff546346%28v=vs.85%29.aspx> - if a driver package is digitally signed, Windows installs the package as a whole and does not selectively omit files in the package based on other versions already present on the computer.

The `windrvr1511.sys` driver provided by Jungs is digitally signed.

- Install the INF file for your device (registering your Plug-and-Play device with `windrvr1511.sys`):

Run the utility `wdreg` with the `install` command to automatically install the INF file and update Windows Device Manager:

```
wdreg -inf <path to your INF file> install
```

You can also use the `wdreg` utility's `preinstall` command to pre-install an INF file for a device that is not currently connected to the PC:

```
wdreg -inf <path to your INF file> preinstall
```

If the installation fails with an `ERROR_FILE_NOT_FOUND` error, inspect the Windows registry to see if the `RunOnce` key exists in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the `RunOnce` key is missing, create it; then try installing the INF file again.

- Install your Kernel Plugin driver (if you created one).
- Install `wdapi1511.dll`:

If your hardware-control application/DLL uses `wdapi1511.dll` (as is the case for the sample and generated DriverWizard WinDriver projects), copy this DLL to the target's `windir%\system32` directory.

When distributing 32-bit applications/DLLs to 32-bit targets OR when distributing 64-bit applications/DLLs to 64-bit targets, copy `WinDriver\redist\wdapi<version>.dll` (e.g. `wdapi900.dll`) to the target's `windir%\system32` directory.

If you are distributing a 32-bit application/DLL to a target 64-bit platform, rename `wdapi1511_32.dll` in your distribution package to `wdapi1511.dll`, and copy the renamed file to the target's `windir%\sysWOW64` directory.

If you attempt to write a 32-bit installation program that installs a 64-bit program, and therefore copies the 64-bit `wdapi1511.dll` DLL to the `windir%\system32` directory, you may find that the file is actually copied to the 32-bit `windir%\sysWOW64` directory. The reason for this is that Windows x64 platforms translate references to 64-bit directories from 32-bit commands into references to 32-bit directories. You can avoid the problem by using 64-bit commands to perform the necessary installation steps from your 32-bit installation program. The `system64.exe` program, provided in the `WinDriver\redist` directory of the Windows x64 WinDriver distributions, enables you to do this.

- Install your hardware-control application/DLL:

Copy your hardware-control application / DLL to the target and run it!

15.2.3. Installing Your Kernel Plugin on the Target Computer

Driver installation on Windows requires administrator privileges.

If you have created a Kernel Plugin driver, follow the additional instructions below:

- Copy your Kernel Plugin driver (`<KP driver name>.sys`) to Windows drivers directory on the target computer (`windir%\system32\drivers`).
- Use the utility `wdreg.exe` 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\util` directory.

15.2.4. Redistribute Your WinDriver-based package as a self-extracting EXE

Starting from WinDriver version 14.3, WinDriver supports creation of NSIS installers for Windows for your generated code and user applications, via a CMake Project compiled on Visual Studio 2019. Currently this feature is supported only on Windows 64 bit applications without a Kernel PlugIn.

15.2.4.1. The Installer

What the Installer does:

- Packs your driver and your user application into a single self-extracting EXE file.
- Installs the user application files to the target computer.
- Installs WinDriver to the target computer using the `wdreg` utility.
- Provides the user with an uninstall EXE file that removes the user application and the driver from the target computer.

This allows to save users precious time in manually creating a redistribution package.

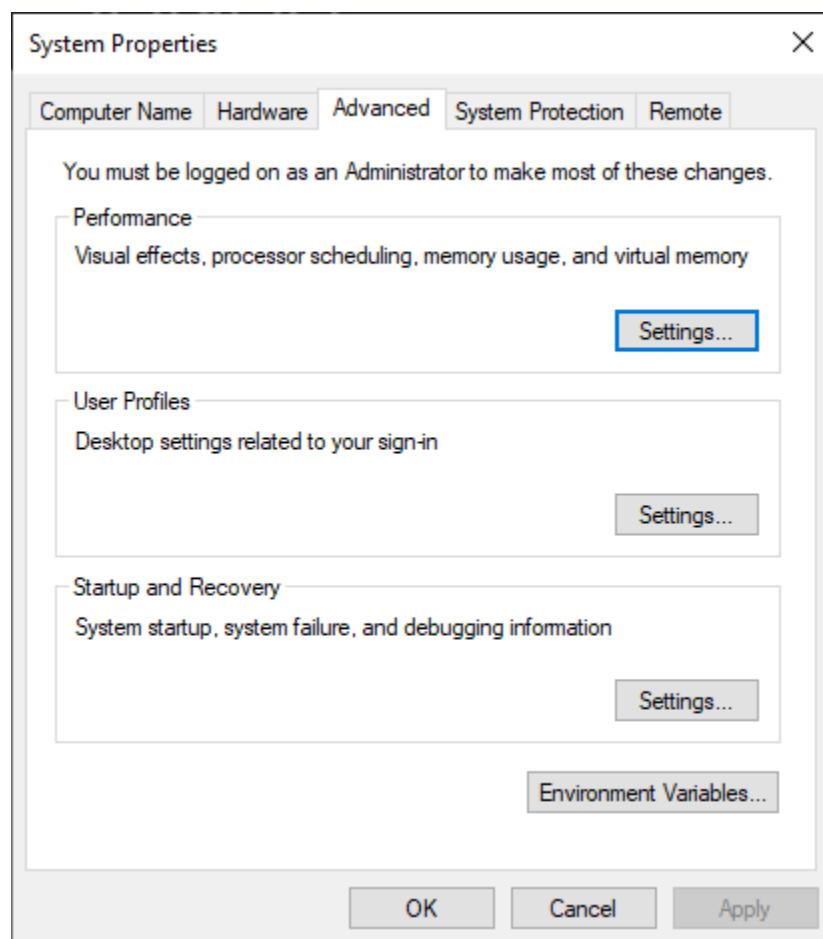
15.2.4.2. Requirements

Make sure Visual Studio 2019 is installed along with the “C++ CMake Tools for Windows” and “Desktop development with C++” components installed.

As the next step, please make sure NSIS is installed if you haven’t already installed it on your development machine. If it is not, you can download it from the NSIS website. Make sure that the PATH environment variable includes the path to `makensis.exe`. This can be done the following way:

- In the Start Menu type `Edit the system environment variables`, this will open the System Properties Window.
- Click the **Environment Variables...** button.
- Edit the Path Variable and make sure it includes the path to `MakeNSIS.exe` (e.g. `C:\Program Files\NSIS`)

Editing Environment Variables

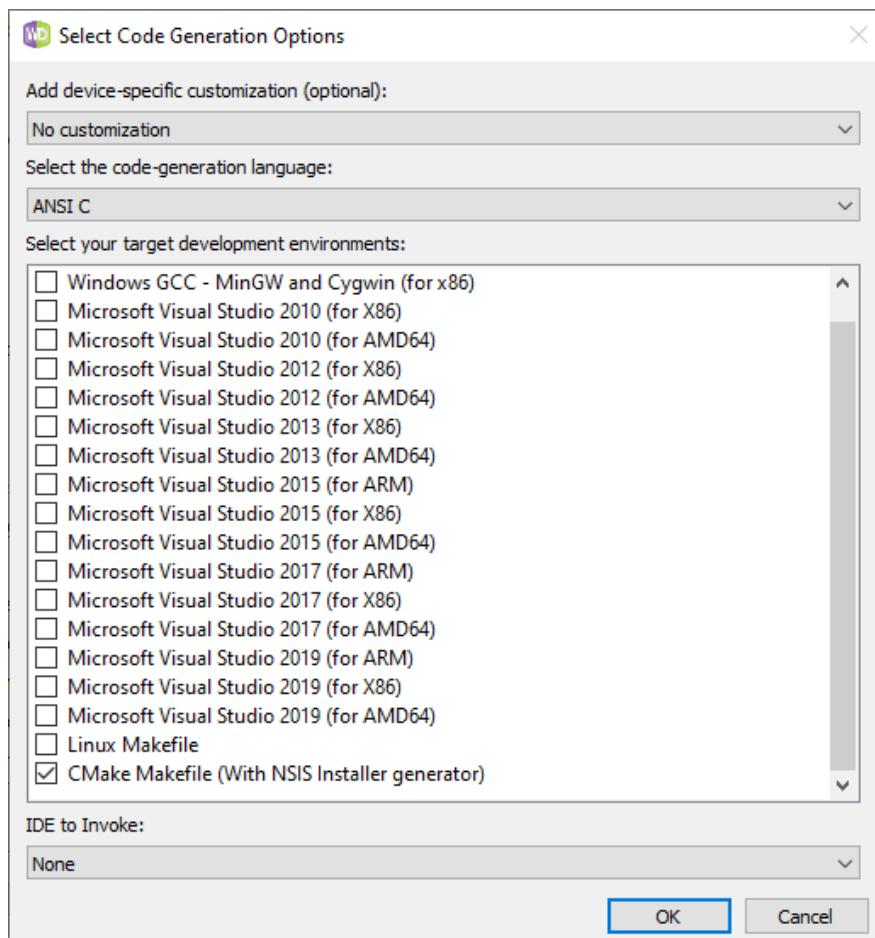


15.2.4.3. Instructions

Follow these steps:

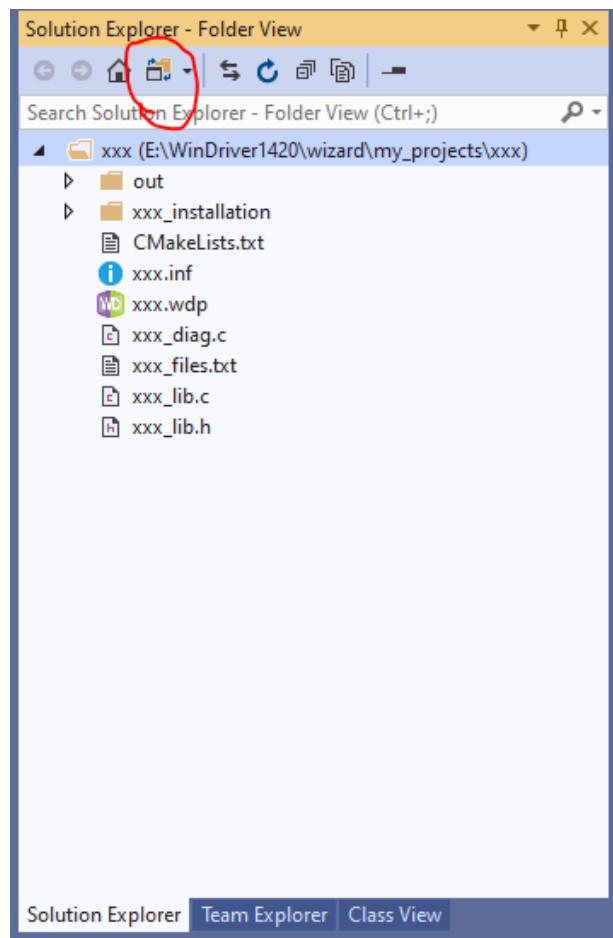
- From the DriverWizard code generation window – check the CMake Makefile (With NSIS Installer Generator) Checkbox.

WinDriver Code Generation Window

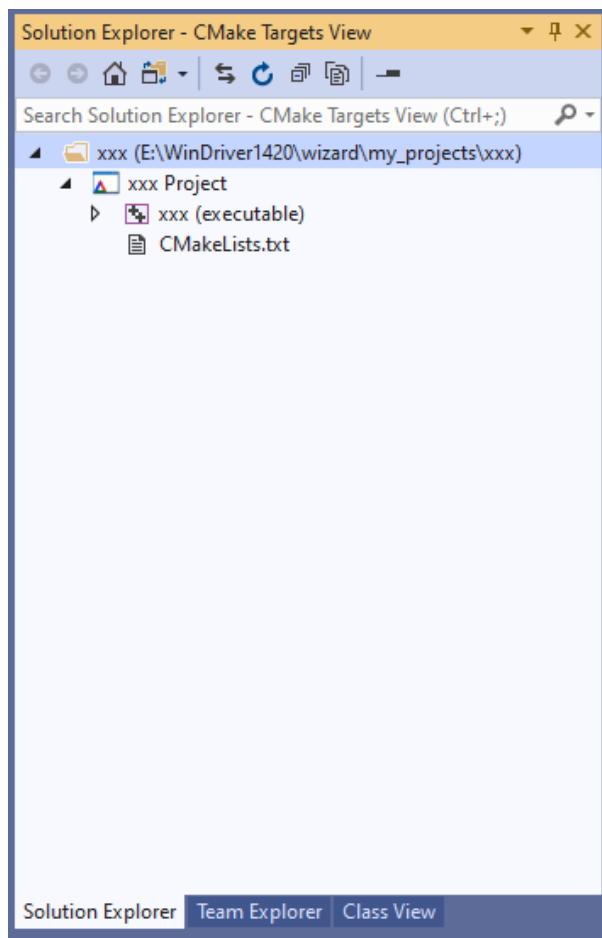


- In your generated code directory for your driver (xxx will be the driver name in this tutorial) found in WinDriver\wizard\my_projects\xxx you should find a CMakeLists.txt file.
- Open Visual Studio 2019.
- Go to **File | Open | CMake** and choose WinDriver\wizard\my_projects\xxx\CMakeLists.txt.
- After all background tasks are complete you will be able to change Solution Explorer to the CMake Targets View from the Switch Views icon in the Solution Explorer.

Changing to CMake Targets View



The CMake Targets View



- In order to start preparing your distribution package – you must compile it in Release configuration.
- In order to redistribute your driver – edit `xxx.lib.c` such that `XXX_DEFAULT_LICENSE_STRING` will be defined to be your license string, and that `XXX_DEFAULT_DRIVER_NAME` will be defined to be your renamed driver name (in this case `xxx`). If you do not perform this step your user application might work on your development computer but will not work on the target computer.

License String replacement in code

```
#define XXX_DEFAULT_LICENSE_STRING "your_license_string"  
#define XXX_DEFAULT_DRIVER_NAME "xxx"
```

- Build the project by right clicking **xxx Project** | **Build All** from the Solution Explorer. You should find a compiled `xxx.exe` file of your generated code in `WinDriver\wizard\my_projects\xxx\out\build\ (PLATFORM_NAME) \WIN32` and an additional installer EXE file `xxx_1.0.0-win64.exe` in `WinDriver\wizard\my_projects\xxx\out\build\ (PLATFORM_NAME)`. Jungo recommends using the installer EXE file for distributing your driver package.
- Feel free to modify the `CMakeLists.txt` file to better suit for your needs.

** Attention**

Before installing the driver on a Windows-based target machine, either digitally sign the driver or disable Digital Signature enforcement. In addition, make sure to run the installer EXE as Administrator to make sure it has privileges to install drivers. Failure to do so might result in the driver not being installed properly on the target machine.

15.3. Linux Driver Distribution

To distribute your driver, prepare a distribution package containing the required files and then build and install the required driver components on the target.

If you have renamed the WinDriver driver module, replace references to `windrvr1511` in the following instructions with the name of your renamed driver module.

It is recommended that you supply an installation shell script to automate the build and installation processes on the target.

15.3.1. Preparing the Distribution Package

Prepare a distribution package containing the required files, as described in this section.

If you wish to distribute drivers for both 32-bit and 64-bit target platforms, you must prepare separate distribution packages for each platform. The required files for each package are provided in the WinDriver installation directory for the respective platform.

In the following instructions, `<source_dir>` represents the source directory from which to copy the distribution files. The default source directory is your WinDriver installation directory. However, if you have renamed the WinDriver driver module, the source directory is a directory containing modified files for compiling and installing the renamed drivers. When using DriverWizard to generate the driver code, the source directory for the renamed driver is the generated `xxx_installation` directory, where `xxx` is the name of your generated driver project.

15.3.1.1. Kernel Module Components

Your WinDriver-based driver relies on the `windrvr1511.o/.ko` kernel driver module, which implements the WinDriver API - `windrvr1511.o/.ko`. In addition, if you have created a Kernel PlugIn driver, the functionality of this driver is implemented in a `kp_xxx_module.o/.ko` kernel driver module (where `xxx` is your selected driver project name).

Your kernel driver modules cannot be distributed as-is; they must be recompiled on each target machine, to match the kernel version on the target. This is due to the following reason: 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, which is checked against the version number encoded into the kernel. This forces Linux driver developers to support recompilation of their driver with the target system's kernel version.

Following is a list of the components you need to distribute to enable compilation of your kernel driver modules on the target machine.

It is recommended that you copy the files to subdirectories in the distribution directory that match the source sub-directories, such as `redist` and `include`, except where otherwise specified. If you select not do so, you will need to modify the file paths in the configuration scripts and related makefile templates, to match the location of the files in your distribution directory.

- From the `<source_dir>/include` directory, copy `windrvr.h`, `wd_ver.h`, and `windrvr_usb.h` - header files required for building the kernel modules on the target.

Note that `windrvr_usb.h` is required also for non-USB drivers.

- From the `<WinDriver installation directory>/util` directory (or from the generated DriverWizard `xxx_installation/redist` directory), copy `wdreg` - a script for loading the WinDriver kernel driver module - to the `redist` distribution directory.
- From the `<source_dir>/redist` directory, unless where otherwise specified, copy the following files:
 - `setup_inst_dir` - a script for installing the WinDriver driver module, using `wdreg` (see above).
 - `linux_wrappers.c/.h` - wrapper library source code files that bind the kernel module to the Linux kernel.

- `linux_common.h` and `wdusb_interface.h` - header files required for building the kernel modules on the target.
- `wdusb_linux.c` - source file used by WinDriver to utilize the USB stack (for USB drivers).

Note that `wdusb_interface.h` is required also for non-USB drivers.

The compiled object code for building the WinDriver kernel driver module:

- `windrivr_gcc_v3.o_shipped` - for GCC v3.x.x compilation (also have been tested to work with GCC up to version 8, for 64-bit versions of Linux)
- `windrivr_gcc_v3_reparm.o_shipped` - for GCC v3.x.x compilation (for 32-bit versions of Linux)

Configuration scripts and makefile templates for creating makefiles for building and installing the WinDriver kernel driver module.

Files that include `.kbuild` in their names use kbuild for the driver compilation.

- `configure` - a configuration script that uses the `makefile.in` template to create a makefile for building and installing the WinDriver driver module, and executes the `configure.wd` script (see below).
- `configure.wd` - a configuration script that uses the `makefile.wd[.kbuild].in` template to create a `makefile.wd[.kbuild]` makefile for building the `windrivr1511.o/.ko` driver module.
- `configure.usb` - a configuration script that uses the `makefile.usb[.kbuild].in` template to create a `makefile.usb[.kbuild]` makefile for building the `windrivr1511_usb.o/.ko` driver module (for USB drivers).
- `makefile.in` - a template for the main makefile for building and installing the WinDriver kernel driver module, using `makefile.wd[.kbuild]` (and `makefile.usb[.kbuild]`).
- `makefile.wd.in` and `makefile.wd.kdbuild.in` - templates for creating `makefile.wd[.kbuild]` makefiles for building and installing the `windrivr1511.o/.ko` driver module.
- `makefile.usb.in` and `makefile.usb.kdbuild.in` - templates for creating `makefile.usb[.kbuild]` makefiles for building and installing the `windrivr1511_usb.o/.ko` driver module.

If you have created a Kernel PlugIn driver - copy the following files as well:

- From the generated DriverWizard `xxx_installation/redist` directory (where `xxx` is the name of your driver project), copy the following configuration script and makefile templates, for creating a makefile for building and installing the Kernel PlugIn driver.

If you did not generate your Kernel PlugIn driver using the DriverWizard, copy the files from your Kernel PlugIn project; the files for the `KP_PCI` sample, for example, are found in the `WinDriver/samples/c/pci←diag/kp_pci` directory.

Note: before copying the files, rename them to add a ".kp" indication - as in the `xxx_installation/redist` file names listed below - in order to distinguish them from the WinDriver driver module files. You will also need to edit the file names and paths in the files, to match the structure of the distribution directory.

- `configure.kp` - a configuration script that uses the `makefile.kp[.kbuild].in` template (see below) to create a `makefile.kp` makefile for building and installing the Kernel PlugIn driver module.

** Attention**

If you have renamed the WinDriver kernel module, be sure to uncomment the following line in your Kernel PlugIn configuration script (by removing the pound sign - "#"), before executing the script, in order to build the driver with the `-DWD_DRIVER_NAME_CHANGE` flag: `# ADDITIONAL_FLAGS="-DWD_DRIVER_NAME_CHANGE"`

- makefile.kp.in and makefile.kp.kbuild.in - templates for creating a makefile.kp makefile for building and installing the Kernel PlugIn driver module. The makefile created from makefile.kp.build.in uses kbuild for the compilation.

From the <source_dir>/lib directory, copy the compiled WinDriver-API object code:

- kp_wdapi1511_gcc_v3.o_shipped - for GCC v3.x.x compilation (also been tested to work with GCC up to version 5.4.0).
- kp_wdapi1511_gcc_v3_reparm.o_shipped - for GCC v3.x.x compilation with the regparm flag.

From the kermode/linux/LINUX.<kernel_version>.<CPU> directory that is created when building the Kernel PlugIn driver on the development machine, copy to the lib distribution subdirectory the compiled object code for building your Kernel PlugIn driver module (where xxx is the name of your Kernel PlugIn driver project):

- kp_xxx_gcc_v3.o_shipped - for GCC v3.x.x compilation (also have been tested to work on GCC versions up to 8, for 64-bit versions of Linux).
- kp_xxx_gcc_v3_reparm.o_shipped - for GCC v3.x.x compilation (for 32-bit versions of Linux).

15.3.1.2. User-Mode Hardware-Control Application or Shared Object

Copy the user-mode hardware-control application or shared object that you created with WinDriver, to the distribution package.

If your hardware-control application/shared object uses libwdapi1511.so - as is the case for the WinDriver samples and generated DriverWizard projects - copy this file from the <source_dir>/lib directory to your distribution package.

If you are distributing a 32-bit application/shared object to a target 64-bit platform - copy libwdapi1511_32.so from the WinDriver/lib directory to your distribution package, and rename the copy to libwdapi1511.so.

If your user application is a Java application/shared object you must also add libwdapi_java1511.so (for distribution of 32-bit binaries to 32-bit target platforms or for distribution of 64-bit binaries to 64-bit platforms). You should also add wdapi_java1511.jar to your package. Get those files from the WinDriver\lib directory of the WinDriver package (specifically from its subdirectory relevant to your platform).

Since your hardware-control application/shared object does not have to be matched against the Linux kernel version number, you may distribute it as a binary object (to protect your code from unauthorized copying). If you select to distribute your driver's source code, note that under the license agreement with Jungo you may not distribute the source code of the libwdapi1511.so shared object, or the WinDriver license string used in your code.

15.3.2. Building and Installing the WinDriver Driver Module on the Target

From the distribution package subdirectory containing the configure script and related build and installation files - normally the redist subdirectory - perform the following steps to build and install the driver module on the target:

- Generate the required makefiles:

```
# for PCI run:  
$ ./configure --disable-usb-support  
# for USB run:  
$ ./configure
```

The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the --with-kernel-source=<path> option, where <path> is the full path to the kernel source directory - e.g., /usr/src/linux.

If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use kbuild to compile the kernel modules. You can force the use of kbuild on earlier versions of Linux, by executing the configuration script with the --enable-kbuild flag.

** Attention**

For a full list of the configuration script options, use the --help option: ./configure --help.

- Build the WinDriver driver module:

```
$ make
```

This will create a LINUX.<kernel version>.<CPU> directory, containing the newly compiled driver module - windrvr1511.o/.ko.

- Install the windrvr1511.o/.ko driver module.

The following command must be executed with root privileges.

```
# make install
```

The installation is performed using the setup_inst_dir script, which copies the driver module to the target's loadable kernel modules directory, and uses the wdreg script to load the driver module.

- Change the user and group IDs and give read/write permissions to the device file /dev/windrvr1511, depending on how you wish to allow users to access hardware through the device.

Due to security reasons, by default the device file is created with permissions only for the root user. 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:

```
'windrvr1511:root:root:0666'
```

Use the wdreg script to dynamically load the WinDriver driver module on the target after each boot. To automate this, copy wdreg to the target machine, and add the following line to the target's Linux boot file (for example, /etc/rc.local):

```
<path to wdreg> windrvr1511
```

15.3.3. Building and Installing Your Kernel Plugin Driver on the Target

If you have created a Kernel Plugin driver [11], build and install this driver - kp_xxx_module.o/.ko - on the target, by performing the following steps from the distribution package subdirectory containing the configure.kp script and related build and installation files - normally the redistrib subdirectory.

- Generate the Kernel Plugin makefile - makefile.kp:

```
$ ./configure.kp
```

The configuration script creates a makefile based on the running kernel. You may select to use another installed kernel source, by executing the script with the --with-kernel-source=<path> option, where <path> is the full path to the kernel source directory - e.g., /usr/src/linux.

If the Linux kernel version is 2.6.26 or higher, the configuration script generates makefiles that use kbuild to compile the kernel modules. You can force the use of kbuild on earlier versions of Linux, by executing the configuration script with the --enable-kbuild flag.

** Attention**

For a full list of the configuration script options, use the --help option: ./configure --help.

- Build the Kernel Plugin driver module:

```
$ make -f makefile.kp
```

This will create a LINUX.<kernel version>.<CPU>.KP directory, containing the newly compiled driver module - kp_xxx_module.o/.ko.

- Install the Kernel Plugin module.

The following command must be executed with root privileges.

```
# make install -f makefile.kp
```

To automatically load your Kernel PlugIn driver on each boot, add the following line to the target's Linux boot file (for example, /etc/rc.local), after the WinDriver driver module load command (replace <path to kp_xxx->_module.o/.ko> with the path to your Kernel PlugIn driver module, which is found in your LINUX.<kernel version>.<CPU>.KP distribution directory):
'/sbin/insmod <path to kp_xxx_module.o/.ko>'`

15.3.4. Installing the User-Mode Hardware-Control Application or Shared Object

If your user-mode hardware-control application or shared object uses libwdapi1511.so (and libwdapi->_java1511.so if it is a Java application), copy it (them) from the distribution package to the target's library directory:

- /usr/lib - when distributing a 32-bit application/shared object to a 32-bit or 64-bit target.
- /usr/lib64 - when distributing a 64-bit application/shared object to a 64-bit target.

If you decided to distribute the source code of the application/shared object, copy the source code to the target as well.

** Attention**

Remember that you may not distribute the source code of the libwdapi1511.so shared object or your WinDriver license string as part of the source code distribution.

15.3.5. Redistribute Your WinDriver-based package as a self-extracting SH (STGZ)

Starting from WinDriver version 14.6, WinDriver supports creation of self extracting installers for Linux for your generated code and user applications, via a CMake Project.

15.3.5.1. The Installer

What the Installer does:

- Packs your driver and your user application into a single self-extracting SH file.
- Installs the user application files to the target computer.
- Installs WinDriver to the target computer running configure and make install.
- Provides the user with an uninstall SH script that removes the user application and the driver from the target computer.

This allows to save users precious time in manually creating a redistribution package.

15.3.5.2. Requirements

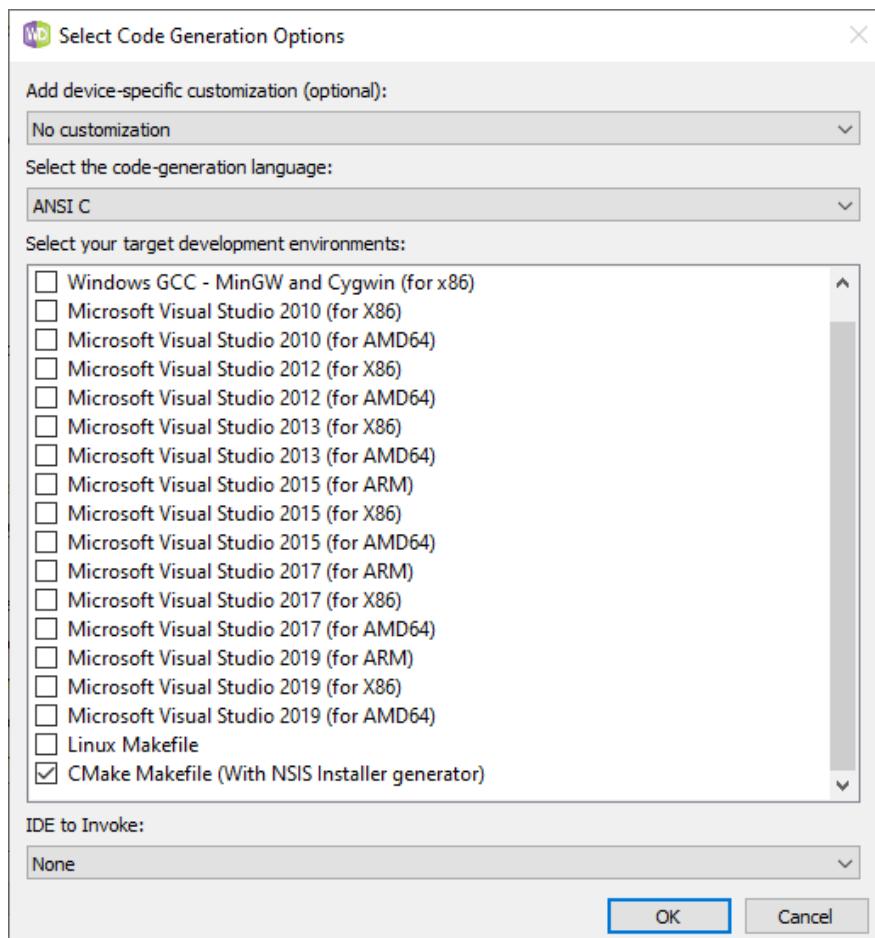
Please make sure CMake version 3.17 or higher is installed on the development computer.

15.3.5.3. Instructions

Please follow these steps:

- From the DriverWizard code generation window – check the CMake Makefile Checkbox.

WinDriver Code Generation Window



- In your generated code directory for your driver (xxx will be the driver name in this tutorial) found in WinDriver\wizard\my_projects\xxx you should find a CMakeLists.txt file.

```
$ cd WinDriver\wizard\my_projects\xxx  
$ cmake . -B build
```

- In order to redistribute your driver – edit xxx_lib.c such that XXX_DEFAULT_LICENSE_STRING will be defined to be your license string, and that XXX_DEFAULT_DRIVER_NAME will be defined to be your renamed driver name (in this case xxx). If you do not perform this step your user application might work on your development computer but will not work on the target computer.
- Build the project
 \$ cd build
 \$ make package

You should find in your user application binary WinDriver\wizard\my_projects\xxx\build and an additional installer SH file xxx_1.0.0-Linux.sh in WinDriver\wizard\my_projects\xxx\build. Jungo recommends using the installer SH file for distributing your driver package.

- Feel free to modify the CMakeLists.txt file to better suit for your needs.

Chapter 16

Dynamically Loading Your Driver

16.1. 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 the need for reboot.

You can dynamically load your driver whether you have created a user-mode or a kernel-mode (see Kernel PlugIn in [Understanding the Kernel PlugIn](#) driver).

To successfully unload your driver, make sure that there are no open handles to the WinDriver service (windrvr1511.sys or your renamed driver), and that there are no connected and enabled Plug-and-Play devices that are registered with this service.

16.2. Windows Dynamic Driver Loading

Windows 7 and higher uses Windows Driver Model (WDM) drivers - files with the extension *.sys (e.g., windrvr1511.sys). WDM drivers are installed via the installation of an INF file (see below).

The WinDriver Windows kernel module - windrvr1511.sys - is a fully WDM driver, which can be installed using the wdreg utility, as explained in the following sections.

16.2.1. 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). This utility is provided in two forms: **wdreg** and **wdreg_gui**.

Both versions can be found in the `WinDriver\util` directory, they 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.

This section describes the use of `wdreg` / `wdreg_gui` on Windows operating systems.

`wdreg` is dependent on `devcon.exe`.

The explanations and examples below refer to `wdreg`, but any references to `wdreg` can be replaced with `wdreg->_gui`.

16.2.1.1. WDM Drivers

This section explains how to use the `wdreg` utility to install the WDM `windrvr1511.sys` driver, or to install INF files that register Plug-and-Play devices to work with this driver, on Windows. You can rename the `windrvr1511.sys` kernel module and modify your device INF file to register with your renamed driver, as explained in [17.2.1. Windows Driver Renaming](#).

To install your modified INF files using `wdreg`, simply replace any references to `windrvr1511` below with the name of your new driver.

This section is not relevant for Kernel PlugIn drivers, since these are not WDM drivers and are not installed via an INF file. For an explanation on how to use `wdreg` to install Kernel PlugIn drivers on Windows, refer to [16.2.1.2. Non-WDM Drivers](#).

Usage: The `wdreg` utility can be used in two ways as demonstrated below:

```
wdreg -inf <filename> [-silent] [-log <logfile>] [install | preinstall | uninstall | enable | disable]
wdreg -rescan <enumerator> [-silent] [-log <logfile>]
```

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

Options	Description
-inf	The path of the INF file to be dynamically installed.
-rescan <enumerator>	Rescan enumerator (ROOT, ACPI, PCI, USB, etc.) for hardware changes. Only one enumerator can be specified.
-silent	Suppress display of all messages (optional).
-log <logfile>	Log all messages to the specified file (optional).

ACTIONS wdreg supports several basic ACTIONS:

Action	Description
install	Installs the INF file, copies the relevant files to their target locations, and dynamically loads the driver specified in the INF file name by replacing the older version (if needed).
preinstall	Pre-installs the INF file for a non-present device.
uninstall	Removes your driver from the registry so that it will not load on next boot (see note below).
enable	Enables your driver.
disable	Disables your driver, i.e., dynamically unloads it, but the driver will reload after system boot (see note below).

To successfully disable/uninstall your driver, make sure that there are no open handles to the WinDriver service (windrvr1511.sys or your renamed driver), and that there are no connected and enabled Plug-and-Play devices that are registered with this service.

16.2.1.2. Non-WDM Drivers

This section explains how to use the wdreg utility to install non-WDM drivers, namely Kernel PlugIn drivers, on Windows.

During development stages, it may be useful to use the wdreg_frontend GUI utility to quickly install/uninstall Kernel Plugins.

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:

Option	Description
-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. The default setting for the -startup option is automatic.
-name	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 function of your Kernel PlugIn project: strcpy (kp->Init->cDriverName, XXX_DRIVER_NAME)

Option	Description
-file	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
```

Flag	Description
-silent	Suppresses the display of messages of any kind.
-log <logfile>	Logs all messages to the specified file.

ACTIONS wdreg supports several basic ACTIONS:

Action	Description
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.

Shortcuts wdreg supports a few shortcut operations for your convenience:

Shortcut	Description
install	Creates and starts your driver. This is the same as first using the wdreg stop action (if a version of the driver is currently loaded) or the wdreg create action (if no version of the driver is currently loaded), and then the wdreg start action.
preinstall	Creates and starts your driver for a non-connected device.
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.

16.2.2. Dynamically Loading/Unloading windrvr1511.sys INF Files

When using WinDriver, you develop a user-mode application that controls and accesses your hardware by using the generic `windrvr1511.sys` driver (WinDriver's kernel module). Therefore, you might want to dynamically load and unload the driver `windrvr1511.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. This section includes `wdreg` usage examples, which are based on the detailed description of `wdreg` contained in the previous section.

Examples:

To load `windrvr1511.inf` and start the `windrvr1511.sys` service - `wdreg -inf <path to windrvr1511@.inf> install` To load an INF file named `device.inf`, located in the `c:\tmp` directory - `wdreg -inf c:\tmp\device.inf install`

You can replace the `install` option in the example above with `preinstall` to pre-install the device INF file for a device that is not currently connected to the PC.

If the installation fails with an `ERROR_FILE_NOT_FOUND` error, inspect the Windows registry to see if the `RunOnce` key exists in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the

RunOnce key is missing, create it; then try installing the INF file again.

To unload the driver/INF file, use the same commands, but simply replace install in the examples above with unin-stall.

16.2.3. Dynamically Loading/Unloading Your Kernel PlugIn Driver

If you used WinDriver to develop a Kernel PlugIn driver, you must load this driver only after loading the generic WinDriver driver - windrvr1511.sys.

When unloading the drivers, unload your Kernel PlugIn driver before unloading windrvr1511.sys.

Kernel PlugIn drivers are dynamically loadable - i.e., they can be loaded and unloaded without reboot. To load/unload your Kernel PlugIn driver (XXX_DRIVER_NAME.sys) use the wdreg command as described above for windrvr1511, with the addition of the name flag, after which you must add the name of your Kernel PlugIn driver.

** Attention**

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

** Recommendation**

During development stages, it may be useful to use the wdreg_frontend GUI utility to quickly install/uninstall Kernel PlugIns.

Examples:

- To load a Kernel PlugIn driver called KPDriver.sys, run this command:

```
wdreg -name KPDriver install
```

- To load a Kernel PlugIn driver called MPEG_Encoder, with file name MPEGENC.sys, run this command:

```
wdreg -name MPEG_Encoder -file MPEGENC install
```

- To uninstall a Kernel PlugIn driver called KPDriver.sys, run this command:

```
wdreg -name KPDriver uninstall
```

- To uninstall a Kernel PlugIn driver called MPEG_Encoder, with file name MPEGENC.sys, run this command:

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

16.3. The wdreg_frontend utility

Under Windows, WinDriver features a GUI frontend utility that is designed to ease installataion / uninstallation of WinDriver and Kernel PlugIn drivers during development phases.

Every operation that can be performed with wdreg can also be performend from wdreg_frontend and their output is identical. wdreg_frontend prints out the complete wdreg command that it performed, making it easier for the developer to copy the desired command and integrate it in their own installation scripts.

** Attention**

Driver installation on Windows requires administrator privileges.

Loading wdreg_frontend:

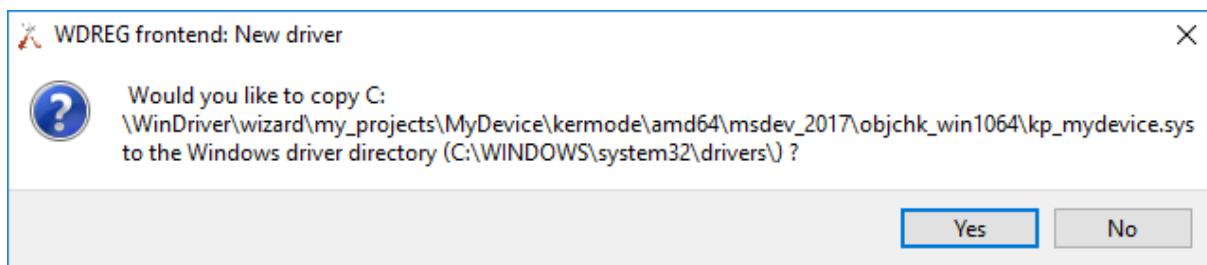
The wdreg_frontend icon



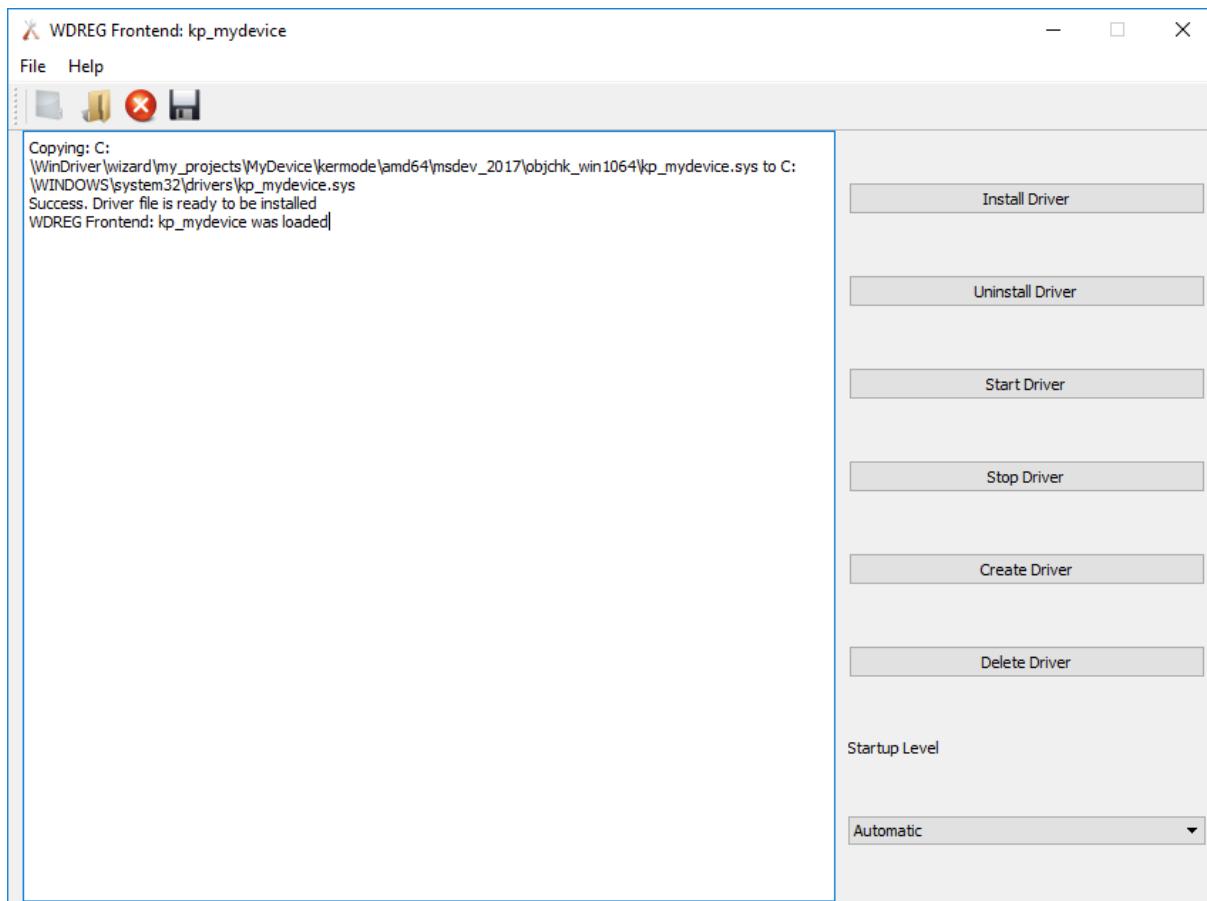
Click the wdreg_frontend icon from the DriverWizard window, and then run WinDriver/util/wdreg_frontend.exe directly.

To load/unload a Kernel Plugin:

The dialog box wdreg_frontend will open if driver file is not located in the Windows system directory



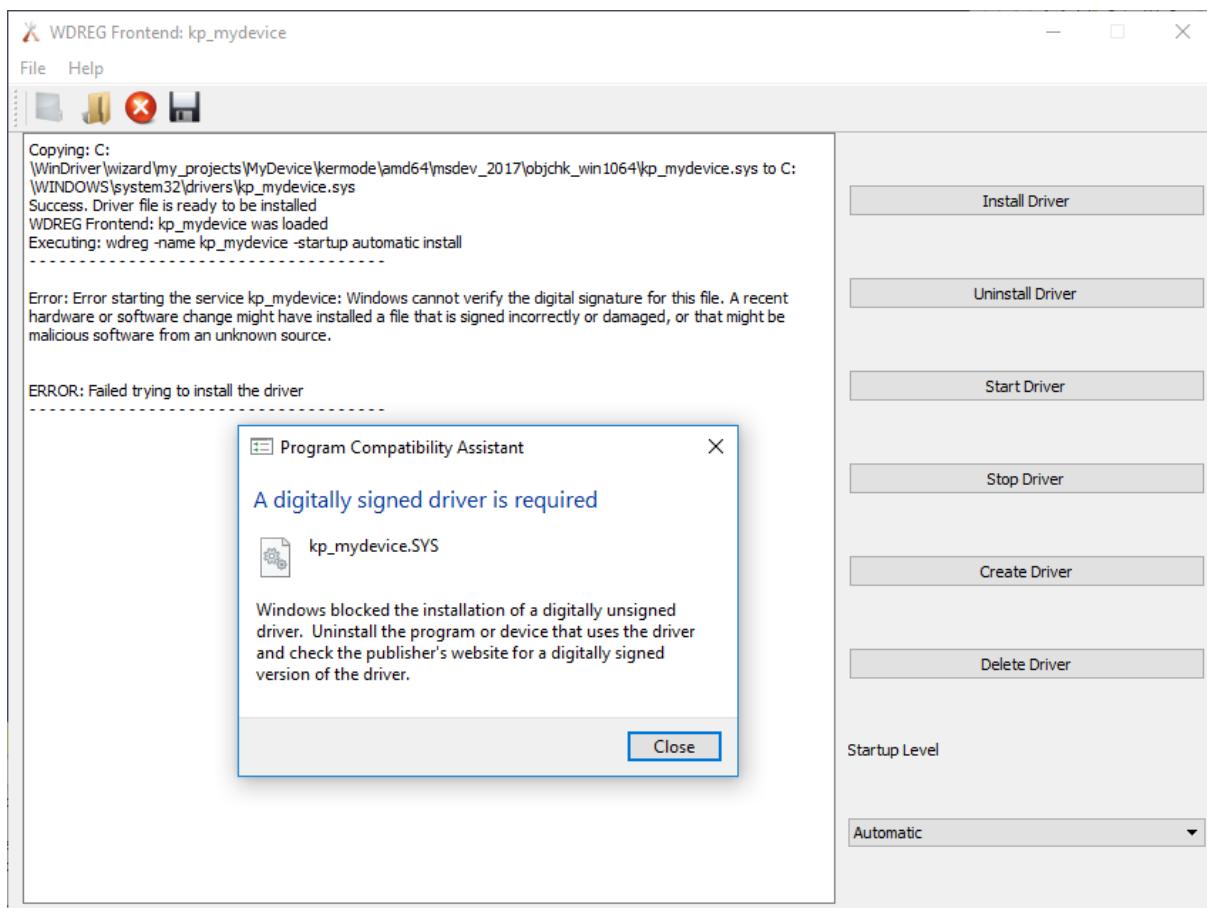
wdreg_frontend after successfully loading a sys file



Click **File | Open** and open the desired sys file. If the sys file is not located in the Windows system directory, a dialog box will appear and ask if you wish wdreg_frontend to copy the sys file to the Windows system directory. If the file loaded correctly, all actions available for .sys files will become enabled: Install Driver, Uninstall Driver, Start Driver, Stop Driver, Create Driver, Delete Driver and choice of Startup level.

Further information on these actions is available on the wdreg [16.2.1.2. Non-WDM Drivers](#) of this document:

Error message given by Windows when trying to install an unsigned driver

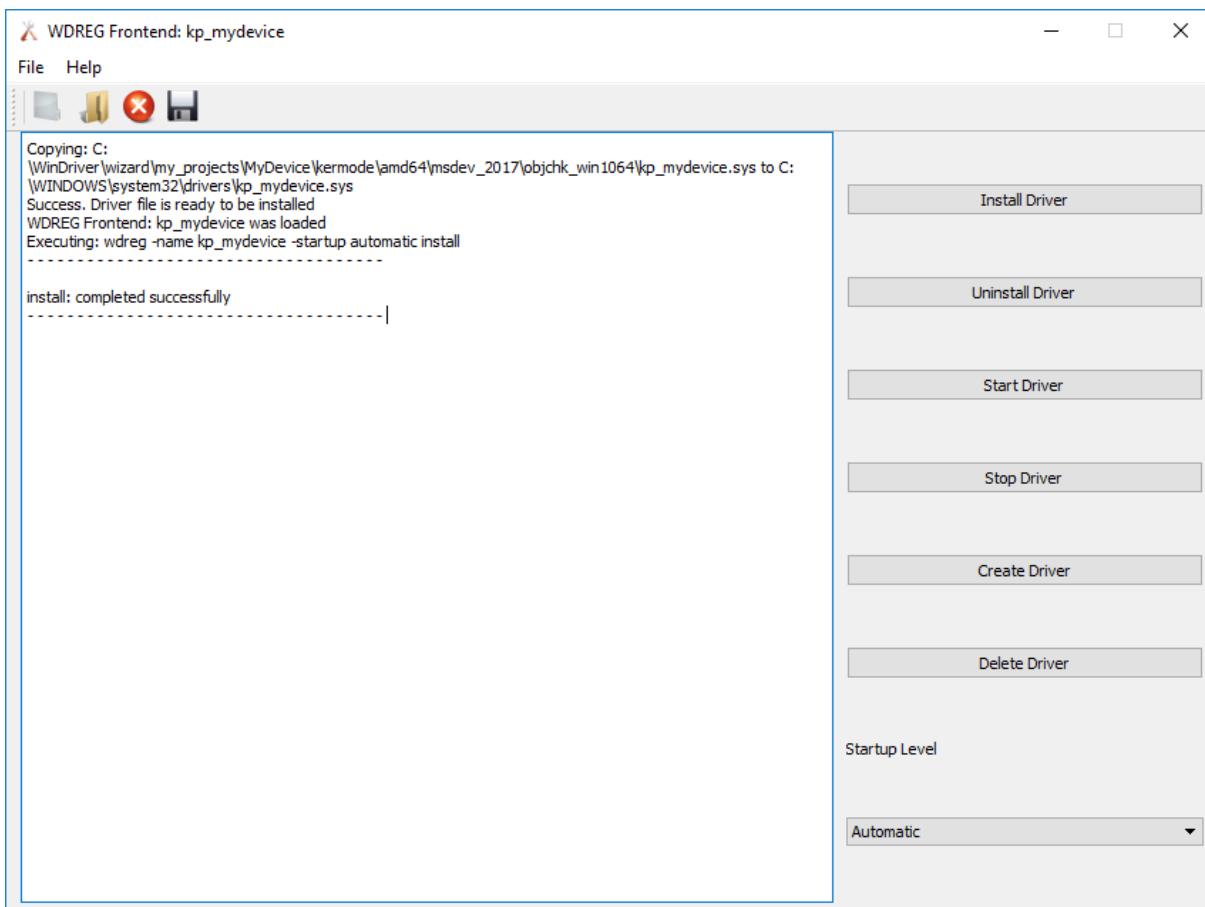


You may need to either digitally sign your sys file or disable digital signature enforcement for the driver to properly load and work under Windows.

To install/uninstall an INF file:

- Click **File | Open** and open the desired sys file.
- If the file loaded correctly, all actions available for .inf files will become enabled: Install Driver, Uninstall Driver, Enable Driver, Disable Driver, Preinstall Driver.

wdreg_frontend upon successfully installing a sys driver



16.4. Linux Dynamic Driver Loading

** Attention**

The following commands must be executed with root privileges.

To dynamically load WinDriver, run the following command:

```
# <path to wdreg> windrvr1511
```

To dynamically unload WinDriver, run the following command:

```
# /sbin/modprobe -r windrvr1511.
```

wdreg is provided in the WinDriver/util directory.

To automatically load WinDriver on each boot, add the following line to the target's Linux boot file (for example, /etc/rc.local):
<path to wdreg> windrvr1511

16.4.1. Dynamically Loading/Unloading Your Kernel Plugin Driver

If you used WinDriver to develop a Kernel Plugin driver, you must load this driver only after loading the generic WinDriver driver - windrvr1511.o/.ko.

When unloading the drivers, unload your Kernel Plugin driver before unloading windrvr1511.o/.ko.

Kernel Plugin drivers are dynamically loadable - i.e., they can be loaded and unloaded without reboot. Use the following commands to dynamically load or unload your Kernel Plugin driver.

** Attention**

The following commands must be executed with root privileges.

xxx in the commands signifies your selected Kernel PlugIn driver project name.

To dynamically load your Kernel PlugIn driver, run this command:

```
# /sbin/insmod <path to kp_xxx_module.o/.ko>
```

When building the Kernel PlugIn driver on the development machine, the Kernel PlugIn driver module is created in your Kernel PlugIn project's kermode/linux/LINUX.<kernel version>.<CPU> directory (see [13.8.2. Linux Kernel PlugIn Driver Compilation](#)). When building the driver on a target distribution machine, the driver module is normally created in an xxx_installation/redist/LINUX.<kernel version>.<CPU>.KP directory (see [15.3.3. Building and Installing Your Kernel PlugIn Driver on the Target](#)).

To dynamically unload your Kernel PlugIn, run this command:

```
# /sbin/rmmod kp_xxx_module
```

To automatically load your Kernel PlugIn driver on each boot, add the following line to the target's Linux boot file (for example, /etc/rc.local), after the WinDriver driver module (windrvr1511) load command (replace <path to kp_xxx_module.o/.ko> with the path to your Kernel PlugIn driver module):

```
/sbin/insmod <path to kp_xxx_module.o/.ko>
```

Chapter 17

Driver Installation - Advanced Issues

17.1. Windows INF Files

Device information (INF) files are text files that provide information used by the Windows Plug-and-Play mechanism 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.

For USB devices, you will not be able to access the device with WinDriver (either from DriverWizard or from the code) without first registering the device to work with `windrivr1511.sys`. This is done by installing an INF file for the device. DriverWizard will offer to automatically generate the INF file for your device.

You can use DriverWizard to generate the INF file on the development machine and then install the INF file on any machine to which you distribute the driver, as explained in the following sections.

17.1.1. Why Should I Create an INF File?

We recommend creating an INF file for the following reasons:

- To bind the WinDriver kernel module to a specific device.
- To override the existing driver (if any).
- To enable WinDriver applications and DriverWizard to access a device.

17.1.2. How Do I Install an INF File When No Driver Exists?

You must have administrative privileges in order to install an INF file.

You can use the `wdreg` utility with the `install` command to automatically install the INF file:
`wdreg -inf <path to the INF file> install`

On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with DriverWizard, by checking the Automatically Install the INF file option in the DriverWizard's INF generation window (refer to [6.2. DriverWizard Walkthrough](#)).

On the development PC, you can also use the `wdreg_frontend` utility to install/uninstall INFs and perform other advanced tasks.

It is also possible to install the INF file manually, 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: locate the device in the Device Manager devices list and select the Update Driver... option from the right-click mouse menu or from the Device Manager's Action menu.

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 wdreg utility to install the INF file automatically, instead of installing it manually.

If the installation fails with an `ERROR_FILE_NOT_FOUND` error, inspect the Windows registry to see if the `RunOnce` key exists in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the `RunOnce` key is missing, create it; then try installing the INF file again.

17.1.3. How Do I Replace an Existing Driver Using the INF File?

You must have administrative privileges in order to replace a driver.

- Install your INF file.

You can use the `wdreg` utility with the `install` command to automatically install the INF file:

```
wdreg -inf <path to INF file> install
```

On the development PC, you can have the INF file automatically installed when selecting to generate the INF file with DriverWizard, by checking the Automatically Install the INF file option in the DriverWizard's INF generation window. On the development PC, you can also use the `wdreg_frontned` utility to install/uninstall INFs and perform other advanced tasks.

It is also possible to install the INF file manually, 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: locate the device in the Device Manager devices list and select the Update Driver... option from the right-click mouse menu or from the Device Manager's Action menu.

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.

If the installation fails with an `ERROR_FILE_NOT_FOUND` error, inspect the Windows registry to see if the `RunOnce` key exists in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. This registry key is required by Windows Plug-and-Play in order to properly install drivers using INF files. If the `RunOnce` key is missing, create it; then try installing the INF file again.

17.2. Renaming the WinDriver Kernel Driver

The WinDriver APIs are implemented within the WinDriver kernel driver module (`windrvr1511.sys/.dll/.ko` - depending on the OS), which provides the main driver functionality and enables you to code your specific driver logic from the user mode.

On Windows and Linux you can change the name of the WinDriver kernel module to your preferred driver name, and then distribute the renamed driver instead of the default kernel module - `windrvr1511.sys/.dll/.ko`. The following sections explain how to rename the driver for each of the supported operating systems.

For information on how to use the Debug Monitor to log debug messages from your renamed driver, refer to [8.2.1.3. Running wddebug_gui for a Renamed Driver](#) : Running `wddebug_gui` for a Renamed Driver.

A renamed WinDriver kernel driver can be installed on the same machine as the original kernel module. You can also install multiple renamed WinDriver drivers on the same machine, simultaneously.

** Recommendation**

Try to give your driver a unique name in order to avoid a potential conflict with other drivers on the target machine on which your driver will be installed.

17.2.1. Windows Driver Renaming

DriverWizard automates most of the work of renaming the Windows WinDriver kernel driver - `windrivr1511.sys`.

When renaming the driver, the CPU architecture (32-/64-bit) of the development platform and its WinDriver installation, should match the target platform.

Renaming the signed `windrivr1511.sys` driver nullifies its signature. In such cases you can select either to sign your new driver, or to distribute an unsigned driver. For more information on driver signing and certification, refer to [17.3. Windows Digital Driver Signing and Certification](#). For guidelines for signing and certifying your renamed driver, refer to [17.3.2. Driver Signing and Certification of WinDriver-Based Drivers](#).

** Attention**

References to `xxx` in this section should be replaced with the name of your generated DriverWizard driver project.

To rename your Windows WinDriver kernel driver, follow these steps:

- Use the DriverWizard utility to generate driver code for your hardware on Windows (refer to [6.2. DriverWizard Walkthrough](#)), using your preferred driver name (`xxx`) as the name of the generated driver project. The generated project directory (`xxx`) will include an `xxx_installation` directory with the following files and directories:

- `redist` directory:

File	Description
<code>xxx.sys</code>	Your new driver, which is actually a renamed copy of the <code>windrivr1511.sys</code> driver. The properties of the generated driver file (such as the file's version, company name, etc.) are identical to the properties of the original <code>windrivr1511.sys</code> driver. You can rebuild the driver with new properties using the files from the generated <code>xxx_installation\sys</code> directory, as explained below.
<code>xxx_driver.inf</code>	A modified version of the <code>windrivr1511.inf</code> file, which will be used to install your new <code>xxx.sys</code> driver. You can make additional modifications to this file, if you wish - namely, changing the string definitions and/or comments in the file.
<code>xxx_device.inf</code>	A modified version of the standard generated DriverWizard INF file for your device, which registers your device with your driver (<code>xxx.sys</code>). You can make additional modifications to this file, if you wish, such as changing the manufacturer or driver provider strings.
<code>wdapi1511.dll</code>	A copy of the WinDriver-API DLL. The DLL is copied here in order to simplify the driver distribution, allowing you to use the generated <code>xxx\redist</code> directory as the main installation directory for your driver, instead of the original <code>WinDriver\redist</code> directory.
<code>wdreg.exe</code> , <code>wdreg_gui.exe</code> , and <code>devcon.exe</code>	Copies of the CUI and GUI versions of the wdreg WinDriver driver installation utility, and the Device Console app required by this utility, (respectively). These files are copied from the <code>WinDriver\util</code> directory, to simplify the installation of the renamed driver.

File	Description
<code>xxx_install.bat</code>	An installation script that executes the <code>wdreg</code> commands for installing the <code>xxx_driver.inf</code> and <code>xxx_device.inf</code> files. This script is designed to simplify the installation of the renamed <code>xxx_driver.sys</code> driver, and the registration of your device with this driver.

– `sys` directory:

This directory contains files for advanced users, who wish to change the properties of their driver file. Changing the file's properties requires rebuilding of the driver module using the Windows Driver Kit (WDK).

To modify the properties of your `xxx.sys` driver file:

- * Verify that you have WDK 8 or above and its corresponding Windows SDK installed on your development PC.
- * Modify the `xxx.rc` resources file in the generated sys directory in order to set different driver file properties.
- * Compile `xxx.vcxproj` with MsBuild or Visual Studio.
- * After rebuilding the `xxx.sys` driver, copy the new driver file to the generated `xxx_installation\redist` directory.
- Verify that your user-mode application calls the `WD_DriverName()` function with your new driver name before calling any other WinDriver function. Note that the sample and generated DriverWizard WinDriver applications already include a call to this function, but with the default driver name (`windrivr1511`), so all you need to do is replace the driver name that is passed to the function in the code with your new driver name.
- Verify that your user-mode driver project is built with the `WD_DRIVER_NAME_CHANGE` preprocessor flag (e.g., `-DWD_DRIVER_NAME_CHANGE`). Note: The sample and generated DriverWizard WinDriver kernel projects/makefiles already set this preprocessor flag by default.
- Install your new driver by following the instructions in [15.2. Windows Driver Distribution](#) of the manual, using the modified files from the generated `xxx_installation` directory instead of the installation files from the original WinDriver distribution. Note that you can use the generated `xxx_install.bat` installation script to simplify the installation.

17.2.1.1 Rebuilding with Custom Version Information (Windows)

WinDriver for Windows allows building your driver with custom version information (Provider, Date, Version, etc.). This information can be seen upon entering the Device Properties of your device in the Device Manager or by right-clicking the driver binary and selecting the “Details” tab.

Visual Studio is required in order to perform the following procedure. In the following instructions `xxx` stands for your generated driver name.

- In your generated code folder go to the `xxx_installation\sys` subfolder.
- Edit `xxx.rc` and modify the details in it to suit your needs.
- Open the `xxx.vcxproj` with Visual Studio and build the project. Make sure you compile the driver in Release configuration for your platform (32 or 64 bits).
- The resulting `xxx.sys` file should include your custom version information.

17.2.2. Linux Driver Renaming

DriverWizard automates most of the work of renaming the Linux WinDriver kernel driver - `windrivr1511.ko`.

** Attention**

References to xxx in this section should be replaced with the name of your generated DriverWizard driver project.

To rename your Linux WinDriver kernel driver, follow these steps:

- Use the DriverWizard utility to generate driver code for your hardware on Linux (refer to [6.2. DriverWizard Walkthrough](#)), using your preferred driver name (xxx) as the name of the generated driver project. The generated project directory (xxx) will include an xxx_installation directory with the following files and directories:

redist directory: This directory contains copies of the files from the original WinDriver/redist installation directory, but with the required modifications for building your xxx.ko driver instead of windrvr1511.ko. **lib** and **include** directories: Copies of the library and include directories from the original WinDriver distribution. These copies are created since the supported Linux WinDriver kernel driver build method relies on the existence of these directories directly under the same parent directory as the redist directory.

- Verify that your user-mode application calls the [WD_DriverName\(\)](#) function with your new driver name before calling any other WinDriver function. Note that the sample and generated DriverWizard WinDriver applications already include a call to this function, but with the default driver name (windrvr1511), so all you need to do is replace the driver name that is passed to the function in the code with your new driver name.
- Verify that your user-mode driver project is built with the `WD_DRIVER_NAME_CHANGE` preprocessor flag (`-DWD_DRIVER_NAME_CHANGE`). Note that the sample and generated DriverWizard WinDriver kernel projects/makefiles already set this preprocessor flag by default. If you have created a Kernel PlugIn driver, you will need to add this flag by uncommenting the following line in the Kernel PlugIn driver's configuration script:

```
# ADDITIONAL_FLAGS="-DWD_DRIVER_NAME_CHANGE"
```

- Install your new driver by following the instructions in [15.3. Linux Driver Distribution](#) of the manual, using the modified files from the generated xxx_installation directory instead of the installation files from the original WinDriver distribution.

As part of the installation, build your new kernel driver module(s) by following the instructions in [15.3. Linux Driver Distribution](#), using the files from your new installation directory.

17.3. Windows Digital Driver Signing and Certification

17.3.1. Windows Digital Driver Signing and Certification Overview

Before distributing your driver, you may digitally sign it using Microsoft's Authenticode mechanism, and/or certify it by submitting it to Microsoft's Windows Certification Program (HLK/HCK/WLP).

Some Windows operating systems, do not require installed drivers to be digitally signed or certified. Only a popup with a warning will appear. There are, however, advantages to getting your driver digitally signed or fully certified, including the following:

- Driver installation on systems where installing unsigned drivers has been blocked.
- Avoiding warnings during driver installation.
- Full pre-installation of INF files.

64-bit versions of Windows 8 and higher require Kernel-Mode Code Signing (KMCS) of software that loads in kernel mode. This has the following implications for WinDriver-based drivers:

- Drivers that are installed via an INF file must be distributed together with a signed catalog file. During driver development, please configure your Windows OS to temporarily allow the installation of unsigned drivers.

For more information about digital driver signing and certification, refer to the following documentation in the Microsoft Development Network (MSDN) library:

- Driver Signing Requirements for Windows.
- Introduction to Code Signing.
- Digital Signatures for Kernel Modules on Windows.

This white paper contains information about kernel-mode code signing, test signing, and disabling signature enforcement during development.

- https://msdn.microsoft.com/en-us/windows-drivers/develop/signing_a_driver

Some of the documentation may still use old terminology. For example, references to the Windows Logo Program (WLP) or to the Windows Hardware Quality Labs (WHQL) or to the Windows Certification Program or to the Windows Hardware Certification Kit (HCK) should be replaced with the Windows Hardware Lab Kit (HLK), and references to the Windows Quality Online Services (Winqual) should be replaced with the Windows Dev Center Hardware Dashboard Services (the Hardware Dashboard).

17.3.1.1. Authenticode Driver Signature

The Microsoft Authenticode mechanism verifies the authenticity of a driver's provider. It allows driver developers to include information about themselves and their code with their programs through the use of digital signatures, and informs users of the driver that the driver's publisher is participating in an infrastructure of trusted entities.

The Authenticode signature does not, however, guarantee the code's safety or functionality.

The WinDriver\redist\windrvr1511.sys driver has an Authenticode digital signature.

17.3.1.2. Windows Certification Program

Microsoft's Windows Certification Program (previously known as the Windows Logo Program (WLP)), lays out procedures for submitting hardware and software modules, including drivers, for Microsoft quality assurance tests. Passing the tests qualifies the hardware/software for Microsoft certification, which verifies both the driver provider's authenticity and the driver's safety and functionality.

To digitally sign and certify a device driver, a Windows Hardware Lab Kit (HLK) package, which includes the driver and the related hardware, should be submitted to the Windows Certification Program for testing, using the Windows Dev Center Hardware Dashboard Services (the Hardware Dashboard).

Jungo's professional services unit provides a complete Windows driver certification service for Jungo-based drivers. Professional engineers efficiently perform all the tests required by the Windows Certification Program, relieving customers of the expense and stress of in-house testing. Jungo prepares an HLK / HCK submission package containing the test results, and delivers the package to the customer, ready for submission to Microsoft.

For more information, refer to https://www.jungo.com/st/services/windows_drivers_certification/.

For detailed information regarding Microsoft's Windows Certification Program and the certification process, refer to the MSDN Windows Hardware Certification page - <https://msdn.microsoft.com/library/windows/hardware/gg463010.aspx> - and to the documentation referenced from that page, including the MSDN Windows Dev Center - Hardware Dashboard Services page - <https://msdn.microsoft.com/library/windows/hardware/gg463091>.

17.3.2. Driver Signing and Certification of WinDriver-Based Drivers

As indicated above, The WinDriver\redist\windrvr1511.sys driver has an embedded Authenticode signature. Since WinDriver's kernel module (windrvr1511.sys) is a generic driver, which can be used as a driver for different types of hardware devices, it cannot be submitted to Microsoft's Windows Certification Program

as a standalone driver. However, once you have used WinDriver to develop a Windows driver for your selected hardware, you can submit both the hardware and driver for Microsoft certification, as explained below.

The driver certification and signature procedures - either via Authenticode or the Windows Certification Program - require the creation of a catalog file for the driver. This file is a sort of hash, which describes other files. The signed `windrivr1511.sys` driver is provided with a matching catalog file - `WinDriver\redist\windrvr1511.cat`. This file is assigned to the `CatalogFile` entry in the `windrivr1511.inf` file (provided as well in the `redist` directory). This entry is used to inform Windows of the driver's signature and the relevant catalog file during the driver's installation.

When the name, contents, or even the date of the files described in a driver's catalog file is modified, the catalog file, and consequently the driver signature associated with it, become invalid. Therefore, if you select to rename the `windrivr1511.sys` driver ([17.2. Renaming the WinDriver Kernel Driver](#)) and/or the related `windrivr1511.inf` file, the `windrivr1511.cat` catalog file and the related driver signature will become invalid.

In addition, when using WinDriver to develop a driver for your Plug-and-Play device, you normally also create a device-specific INF file that registers your device to work with the `windrivr1511.sys` driver module (or a renamed version of this driver). Since this INF file is created at your site, for your specific hardware, it is not referenced from the `windrivr1511.cat` catalog file and cannot be signed by Jungo a priori.

When renaming `windrivr1511.sys` and/or creating a device-specific INF file for your device, you have two alternative options regarding your driver's digital signing:

- Do not digitally sign your driver. If you select this option, remove or comment-out the reference to the `windrivr1511.cat` file from the `windrivr1511.inf` file (or your renamed version of this file).
- Submit your driver to the Windows Certification Program, or have it Authenticode signed.

Note that while renaming `WinDriver\redist\windrvr1511.sys` nullifies the driver's digital signature, the driver is still compliant with the certification requirements of the Windows Certification Program.

To digitally sign/certify your driver, follow these steps:

- Create a new catalog file for your driver, as explained in the Windows Certification Program documentation. The new file should reference both `windrivr1511.sys` (or your renamed driver) and any INF files used in your driver's installation.
- Assign the name of your new catalog file to the `CatalogFile` entry in your driver's INF file(s). (You can either change the `CatalogFile` entry in the `windrivr1511.inf` file to refer to your new catalog file, and add a similar entry in your device-specific INF file; or incorporate both `windrivr1511.inf` and your device INF file into a single INF file that contains such a `CatalogFile` entry).
- Submit your driver to Microsoft's Windows Certification Program or for an Authenticode signature. If you wish to submit your driver to the Windows Certification Program, refer to the additional guidelines in [17.3.2.1. HCK/HLK Test Notes](#).

Many WinDriver customers have already successfully digitally signed and certified their WinDriver-based drivers.

17.3.2.1. HCK/HLK Test Notes

As indicated in Microsoft's documentation, before submitting the driver for testing and certification you need to download the Windows Hardware Certification Kit (HCK) (for Windows 8) or the Windows Hardware Lab Kit (HLK) (for Windows 10), and run the relevant tests for your hardware/software. After you have verified that you can successfully pass the HCK/HLK tests, create the required logs package and proceed according to Microsoft's documentation.

For more information, refer to the MSDN Windows Hardware Certification Kit (HCK) page - <https://msdn.microsoft.com/library/windows/hardware/hh833788>.

17.3.3. Secure Boot and Driver Development

Secure boot is a security standard developed by Microsoft to help make sure that a PC boots using only software that is trusted by the Original Equipment Manufacturer (OEM). When the PC starts, the firmware checks the signature of each piece of boot software (UEFI, EFI apps, etc.) and blocks the loading of unsigned drivers.

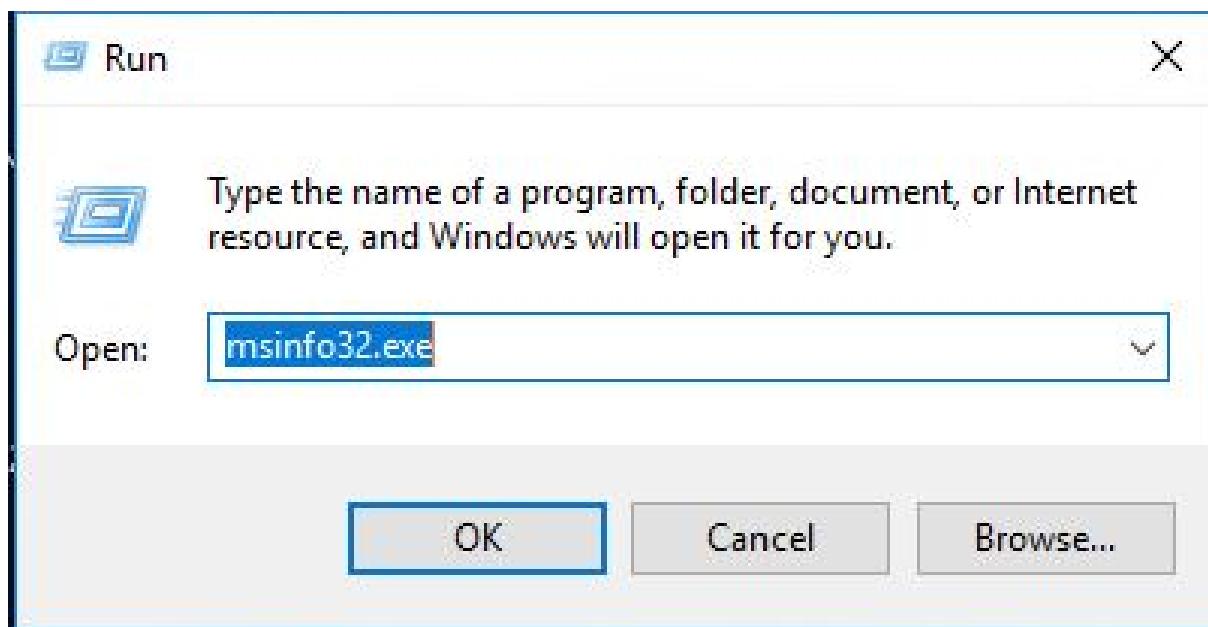
**** Attention****

Disabling Secure Boot and Digital signature enforcement is recommended during driver development and testing. However, a signed driver (Attestation Signing or HLK/WHQL) is required in order to install the driver after it has been distributed.

You can check if Secure Boot is enabled on your system by following one of the procedures below:

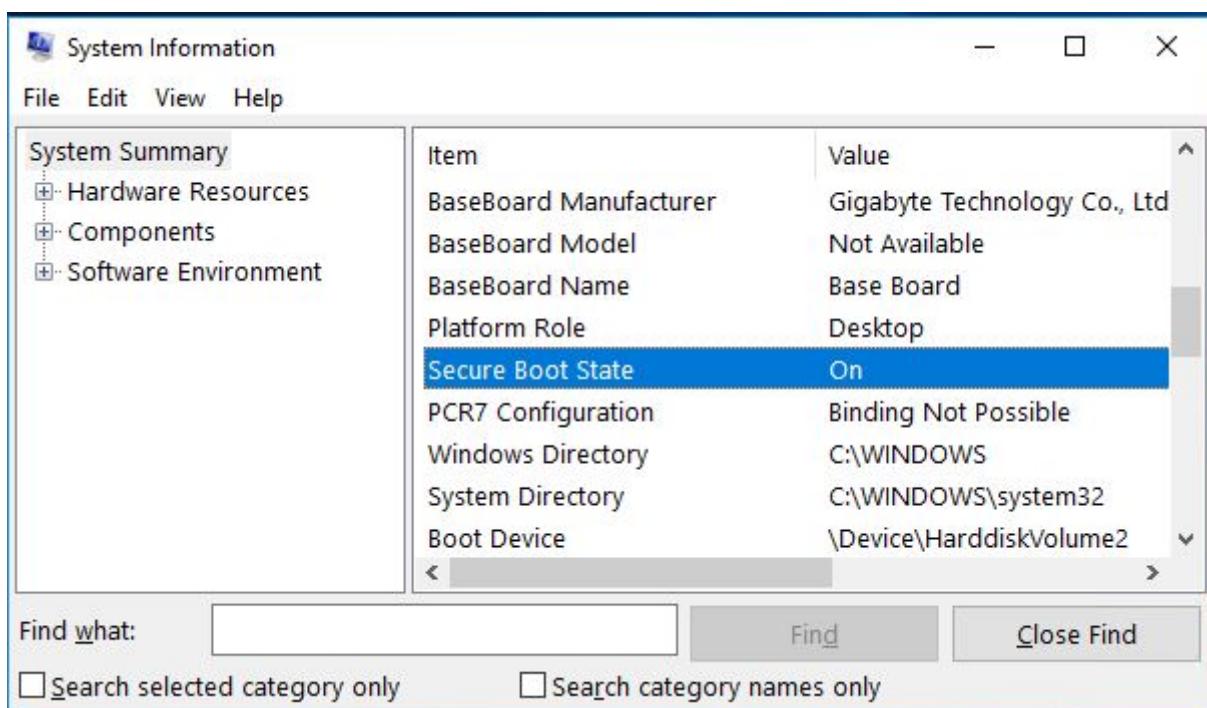
17.3.3.1. Through System Information

- Press on Windows + R to open Run window.

Run window

- Run msinfo32.exe to open the system information.
- Look for "Secure Boot State" and check its status under "System Summary".

System Information



- If the state is “Off” / “Unsupported” or if “Secure Boot State” is not listed under “System Summary” – that means that Secure Boot is disabled.

17.3.3.2. Through PowerShell

- Open PowerShell with administrator privileges.
- Enter the command `Confirm-SecureBootUEFI` and press Enter.

PowerShell

The screenshot shows an Administrator: Windows PowerShell window. The command `Confirm-SecureBootUEFI` was run, and it returned the value `True`, indicating that Secure Boot is enabled. The PowerShell window also displays the standard copyright notice from Microsoft.

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> Confirm-SecureBootUEFI
True
PS C:\WINDOWS\system32>
```

- Any result other than True means that Secure Boot is disabled

17.3.3.3. Through WinDriver User-Mode API

Starting from WinDriver 12.8 the user-mode API (e.g. `wdapi1280.dll`) includes the function `check_secureBoot_enabled()` that can be used as follows:

```
DWORD dwStatus = check_secureBoot_enabled();
if (dwStatus == WD_STATUS_SUCCESS)
    printf("Secure Boot is enabled");
else
    printf("Secure Boot is disabled / unsupported");
```

17.3.3.4. Disabling Secure Boot

In order to install and load unsigned drivers Secure Boot must be disabled.

Secure Boot can be disabled as follows:

- Open PC BIOS menu (this can be done by restarting your PC).
- Find the Secure Boot setting (usually under “Boot Manager”) and set it to “Disabled”.
- Save the settings and Exit. PC will reboot.

For more information on Secure Boot setting, please see Microsoft Documentation.

17.3.4. Temporary disabling digital signature enforcement in Windows 10

Windows 10 (and other Windows versions) requires kernel drivers to be digitally signed. The WinDriver virtual device, which is installed during WinDriver installation process, is digitally signed. However, INF files that are generated by the DriverWizard utility are not signed and therefore will fail to install on Windows 10. In order to install them, it is required to digitally sign them. To bypass this requirement, it is possible to temporarily disable digital driver signing enforcement and install your INF file while the enforcement is disabled.

You can do this by following the procedure below:

- Click on the Start menu button.
- Hold down the Shift button in the keyboard, and click the Power icon on the left side of the Start menu.
- From the menu that will open, choose Restart.
- In the screen that will open, choose Troubleshoot.
- Choose Advanced options.
- Choose Start-up Settings.
- The computer will restart and load a Startup Settings screen. Press F7 to disable driver signature enforcement.
- Your INF should install successfully.

This operation mode (Disabled Digital Signature Enforcement), which allows you to freely install INF files generated by WinDriver, will last until the next reboot. Your newly installed INF files should work properly even after the Digital Signature Enforcement will automatically be switched on again in the next reboot. Other ways are available to permanently disable Digital Signature Enforcement, and are dependent on your Windows version. Full explanation for these ways can be found online.

** Note**

Jungo provides Digital Signature Services for drivers developed with WinDriver. In order to find more information, please send us an e-mail to sales@jungo.com.

Chapter 18

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

CCString	203
KP_INIT	209
KP_OPEN_CALL	211
WD_BUS	212
WD_CARD	213
WD_CARD_CLEANUP	214
WD_CARD_REGISTER	215
WD_DEBUG	217
WD_DEBUG_ADD	218
WD_DEBUG_DUMP	219
WD_DMA	219
WD_DMA_PAGE	222
WD_EVENT	223
WD_GET_DEVICE_PROPERTY	226
WD_INTERRUPT	228
WD_IPC_PROCESS	230
WD_IPC_REGISTER	231
WD_IPC_SCAN_PROCS	232
WD_IPC_SEND	233
WD_ITEMS	234
WD_KERNEL_BUFFER	238
WD_KERNEL_PLUGIN	239
WD_KERNEL_PLUGIN_CALL	240
WD_LICENSE	241
WD_OS_INFO	242
WD_PCI_CAP	243
WD_PCI_CARD_INFO	244
WD_PCI_CONFIG_DUMP	245
WD_PCI_ID	246
WD_PCI_SCAN_CAPS	247
WD_PCI_SCAN_CARDS	248
WD_PCI_SLOT	250
WD_PCI_SRIOV	251
WD_SLEEP	251
WD_TRANSFER	252
WD_USAGE	255
WD_VERSION	255
WDC_ADDR_DESC Address space information struct	256
WDC_DEVICE Device information struct	257
WDC_INTERRUPT_PARAMS	260
WDC_PCI_SCAN_CAPS_RESULT PCI capabilities scan results	261
WDC_PCI_SCAN_RESULT PCI scan results	262

WDS_IPC_MSG_RX	
IPC message received	263
WDS_IPC_SCAN_RESULT	
IPC scan processes results	264
WDU_ALTERNATE_SETTING	264
WDU_CONFIGURATION	265
WDU_CONFIGURATION_DESCRIPTOR	266
WDU_DEVICE	268
WDU_DEVICE_DESCRIPTOR	270
WDU_ENDPOINT_DESCRIPTOR	272
WDU_EVENT_TABLE	274
WDU_GET_DESCRIPTOR	275
WDU_GET_DEVICE_DATA	276
WDU_HALT_TRANSFER	277
WDU_INTERFACE	278
WDU_INTERFACE_DESCRIPTOR	279
WDU_MATCH_TABLE	281
WDU_PIPE_INFO	282
WDU_RESET_DEVICE	284
WDU_RESET_PIPE	285
WDU_SELECTIVE_SUSPEND	285
WDU_SET_INTERFACE	286
WDU_STREAM	287
WDU_STREAM_STATUS	288
WDU_TRANSFER	289
WDU_WAKEUP	291

Chapter 19

File Index

File List

Here is a list of all files with brief descriptions:

bits.h	293
cstring.h	294
kpstdlib.h	296
pci_regs.h	306
pci_strings.h	373
status_strings.h	375
utils.h	376
wd_kp.h	385
wd_log.h	389
wd_types.h	392
wd_ver.h	393
wdc_defs.h	396
wdc_lib.h	403
wds_lib.h	472
wdu_lib.h	486
windrvr.h	506
windrvr_32bit.h	594
windrvr_events.h	597
windrvr_int_thread.h	599
windrvr_usb.h	600
kp_pci.c	610

Chapter 20

Data Structure Documentation

CCString Class Reference

```
#include <cstring.h>
```

Public Member Functions

- `CCString ()`
- `CCString (const CCString &stringSrc)`
- `CCString (PCSTR pcwStr)`
- `virtual ~CCString ()`
- `operator PCSTR () const`
- `operator char * () const`
- `const int operator== (const CCString &s)`
- `const int operator== (const char *s)`
- `const int operator!= (const CCString &s)`
- `const int operator!= (const char *s)`
- `const CCString & operator= (const CCString &s)`
- `const CCString & operator= (PCSTR s)`
- `const CCString & operator+= (const PCSTR s)`
- `const CCString & cat_printf (const PCSTR format,...)`
- `char & operator[] (int i)`
- `int Compare (PCSTR s)`
- `int CompareNoCase (PCSTR s)`
- `const int operator!= (char *s)`
- `void MakeUpper ()`
- `void MakeLower ()`
- `CCString Mid (int nFirst, int nCount)`
- `CCString Mid (int nFirst)`
- `CCString StrRemove (PCSTR str)`
- `CCString StrReplace (PCSTR str, PCSTR new_str)`
- `int Length () const`
- `bool contains (PCSTR substr)`
- `bool is_empty ()`
- `void Format (const PCSTR format,...)`
- `int IsAllocOK ()`
- `int GetBuffer (unsigned long size)`
- `int strcmp (const PCSTR s)`
- `int stricmp (const PCSTR s)`
- `void sprintf (const PCSTR format,...)`
- `void toupper ()`
- `void tolower ()`
- `CCString tolower_copy (void) const`
- `CCString toupper_copy (void) const`
- `int find_first (char c) const`
- `int find_last (char c) const`
- `CCString substr (int start, int end) const`
- `CCString trim (int index) const`

Data Fields

- char * `m_str`

Protected Member Functions

- void `Init ()`
- void `vsprintf (const PCSTR format, va_list ap)`

Protected Attributes

- int `m_buf_size`

Detailed Description

Definition at line 9 of file `cstring.h`.

Constructor & Destructor Documentation

`CCString()` [1/3]

```
CCString::CCString ( )
```

`CCString()` [2/3]

```
CCString::CCString (
    const CCString & stringSrc )
```

`CCString()` [3/3]

```
CCString::CCString (
    PCSTR pcwStr )
```

`~CCString()`

```
virtual CCString::~CCString ( ) [virtual]
```

Member Function Documentation

cat_printf()

```
const CCString & CCString::cat_printf (
    const PCSTR format,
    ... )
```

Compare()

```
int CCString::Compare (
    PCSTR s )
```

CompareNoCase()

```
int CCString::CompareNoCase (
    PCSTR s )
```

contains()

```
bool CCString::contains (
    PCSTR substr )
```

find_first()

```
int CCString::find_first (
    char c ) const
```

find_last()

```
int CCString::find_last (
    char c ) const
```

Format()

```
void CCString::Format (
    const PCSTR format,
    ... )
```

GetBuffer()

```
int CCString::GetBuffer (
    unsigned long size )
```

Init()

```
void CCString::Init ( ) [protected]
```

is_empty()

```
bool CCString::is_empty ( )
```

IsAllocOK()

```
int CCString::IsAllocOK ( )
```

Length()

```
int CCString::Length ( ) const
```

MakeLower()

```
void CCString::MakeLower ( )
```

MakeUpper()

```
void CCString::MakeUpper ( )
```

Mid() [1/2]

```
CCString CCString::Mid ( int nFirst )
```

Mid() [2/2]

```
CCString CCString::Mid ( int nFirst, int nCount )
```

operator char *()

```
CCString::operator char * ( ) const
```

operator PCSTR()

```
CCString::operator PCSTR ( ) const
```

operator"!=() [1/3]

```
const int CCString::operator!= (
    char * s )
```

operator"!=() [2/3]

```
const int CCString::operator!= (
    const CCString & s )
```

operator"!=() [3/3]

```
const int CCString::operator!= (
    const char * s )
```

operator+=()

```
const CCString & CCString::operator+= (
    const PCSTR s )
```

operator=() [1/2]

```
const CCString & CCString::operator= (
    const CCString & s )
```

operator=() [2/2]

```
const CCString & CCString::operator= (
    PCSTR s )
```

operator==(1/2)

```
const int CCString::operator== (
    const CCString & s )
```

operator==(2/2)

```
const int CCString::operator== (
    const char * s )
```

operator[]()

```
char & CCString::operator[ ] (
    int i )
```

sprintf()

```
void CCString::sprintf (
    const PCSTR format,
    ... )
```

strcmp()

```
int CCString::strcmp (
    const PCSTR s )
```

stricmp()

```
int CCString::stricmp (
    const PCSTR s )
```

StrRemove()

```
CCString CCString::StrRemove (
    PCSTR str )
```

StrReplace()

```
CCString CCString::StrReplace (
    PCSTR str,
    PCSTR new_str )
```

substr()

```
CCString CCString::substr (
    int start,
    int end ) const
```

tolower()

```
void CCString::tolower ( )
```

tolower_copy()

```
CCString CCString::tolower_copy (
    void ) const
```

toupper()

```
void CCString::toupper ( )
```

toupper_copy()

```
CCString CCString::toupper_copy (
    void ) const
```

trim()

```
CCString CCString::trim (
    int index ) const
```

vsprintf()

```
void CCString::vsprintf (
    const PCSTR format,
    va_list ap ) [protected]
```

Field Documentation

m_buf_size

```
int CCString::m_buf_size [protected]
```

Definition at line 71 of file [cstring.h](#).

m_str

```
char* CCString::m_str
```

Definition at line 49 of file [cstring.h](#).

The documentation for this class was generated from the following file:

- [cstring.h](#)

KP_INIT Struct Reference

```
#include <wd_kp.h>
```

Data Fields

- DWORD `dwVerWD`
version of the WinDriver Kernel PlugIn library
- CHAR `cDriverName [WD_MAX_KP_NAME_LENGTH]`
return the device driver name
- `KP_FUNC_OPEN funcOpen`
returns the KP_Open function
- `KP_FUNC_OPEN funcOpen_32_64`
returns the KP_Open function for 32 bit app with 64 bit KP.

Detailed Description

Definition at line 66 of file [wd_kp.h](#).

Field Documentation

`cDriverName`

CHAR `KP_INIT::cDriverName [WD_MAX_KP_NAME_LENGTH]`

return the device driver name

Definition at line 69 of file [wd_kp.h](#).

`dwVerWD`

DWORD `KP_INIT::dwVerWD`

version of the WinDriver Kernel PlugIn library

Definition at line 68 of file [wd_kp.h](#).

`funcOpen`

`KP_FUNC_OPEN KP_INIT::funcOpen`

returns the KP_Open function

Definition at line 71 of file [wd_kp.h](#).

`funcOpen_32_64`

`KP_FUNC_OPEN KP_INIT::funcOpen_32_64`

returns the KP_Open function for 32 bit app with 64 bit KP.

Definition at line 72 of file [wd_kp.h](#).

The documentation for this struct was generated from the following file:

- [wd_kp.h](#)

KP_OPEN_CALL Struct Reference

```
#include <wd_kp.h>
```

Data Fields

- KP_FUNC_CLOSE funcClose
- KP_FUNC_CALL funcCall
- KP_FUNC_INT_ENABLE funcIntEnable
- KP_FUNC_INT_DISABLE funcIntDisable
- KP_FUNC_INT_AT_IRQL funcIntAtIrql
- KP_FUNC_INT_AT_DPC funcIntAtDpc
- KP_FUNC_INT_AT_IRQL_MSI funcIntAtIrqlMSI
- KP_FUNC_INT_AT_DPC_MSI funcIntAtDpcMSI
- KP_FUNC_EVENT funcEvent

Detailed Description

Definition at line 48 of file [wd_kp.h](#).

Field Documentation

funcCall

```
KP_FUNC_CALL KP_OPEN_CALL::funcCall
```

Definition at line 51 of file [wd_kp.h](#).

funcClose

```
KP_FUNC_CLOSE KP_OPEN_CALL::funcClose
```

Definition at line 50 of file [wd_kp.h](#).

funcEvent

```
KP_FUNC_EVENT KP_OPEN_CALL::funcEvent
```

Definition at line 58 of file [wd_kp.h](#).

funcIntAtDpc

```
KP_FUNC_INT_AT_DPC KP_OPEN_CALL::funcIntAtDpc
```

Definition at line 55 of file [wd_kp.h](#).

funcIntAtDpcMSI

KP_FUNC_INT_AT_DPC_MS1 KP_OPEN_CALL::funcIntAtDpcMSI

Definition at line 57 of file [wd_kp.h](#).

funcIntAtIrql

KP_FUNC_INT_AT_IRQL KP_OPEN_CALL::funcIntAtIrql

Definition at line 54 of file [wd_kp.h](#).

funcIntAtIrqlMSI

KP_FUNC_INT_AT_IRQL_MS1 KP_OPEN_CALL::funcIntAtIrqlMSI

Definition at line 56 of file [wd_kp.h](#).

funcIntDisable

KP_FUNC_INT_DISABLE KP_OPEN_CALL::funcIntDisable

Definition at line 53 of file [wd_kp.h](#).

funcIntEnable

KP_FUNC_INT_ENABLE KP_OPEN_CALL::funcIntEnable

Definition at line 52 of file [wd_kp.h](#).

The documentation for this struct was generated from the following file:

- [wd_kp.h](#)

WD_BUS Struct Reference

```
#include <windrvr.h>
```

Data Fields

- **WD_BUS_TYPE dwBusType**
Bus Type: WD_BUS_PCI/ISA/EISA.
- **DWORD dwDomainNum**
Domain number.
- **DWORD dwBusNum**
Bus number.
- **DWORD dwSlotFunc**
Slot number on the bus.

Detailed Description

Definition at line 663 of file [windrvr.h](#).

Field Documentation

dwBusNum

DWORD WD_BUS::dwBusNum

Bus number.

Definition at line 667 of file [windrvr.h](#).

dwBusType

WD_BUS_TYPE WD_BUS::dwBusType

Bus Type: WD_BUS_PCI/ISA/EISA.

Definition at line 665 of file [windrvr.h](#).

dwDomainNum

DWORD WD_BUS::dwDomainNum

Domain number.

Definition at line 666 of file [windrvr.h](#).

dwSlotFunc

DWORD WD_BUS::dwSlotFunc

Slot number on the bus.

Definition at line 668 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_CARD Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD dwItems
- WD_ITEMS Item [WD_CARD_ITEMS]

Detailed Description

Definition at line 745 of file [windrvr.h](#).

Field Documentation

dwItems

`DWORD WD_CARD::dwItems`

Definition at line 747 of file [windrvr.h](#).

Item

`WD_ITEMS WD_CARD::Item[WD_CARD_ITEMS]`

Definition at line 749 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_CARD_CLEANUP Struct Reference

`#include <windrvr.h>`

Data Fields

- `DWORD hCard`
Handle of card.
- `WD_TRANSFER * Cmd`
Buffer with `WD_TRANSFER` commands.
- `DWORD dwCmds`
Number of commands.
- `DWORD dwOptions`
0 (default) or `WD_FORCE_CLEANUP`

Detailed Description

Definition at line 810 of file [windrvr.h](#).

Field Documentation

Cmd

`WD_TRANSFER* WD_CARD_CLEANUP::Cmd`

Buffer with `WD_TRANSFER` commands.

Definition at line 814 of file `windrvr.h`.

dwCmds

`DWORD WD_CARD_CLEANUP::dwCmds`

Number of commands.

Definition at line 816 of file `windrvr.h`.

dwOptions

`DWORD WD_CARD_CLEANUP::dwOptions`

0 (default) or `WD_FORCE_CLEANUP`

Definition at line 817 of file `windrvr.h`.

hCard

`DWORD WD_CARD_CLEANUP::hCard`

Handle of card.

Definition at line 812 of file `windrvr.h`.

The documentation for this struct was generated from the following file:

- `windrvr.h`

WD_CARD_REGISTER Struct Reference

`#include <windrvr.h>`

Data Fields

- `WD_CARD Card`
Card to register.
- `DWORD fCheckLockOnly`
Only check if card is lockable, return hCard=1 if OK.
- `DWORD hCard`
Handle of card.
- `DWORD dwOptions`
Should be zero.
- `CHAR cName [32]`
Name of card.
- `CHAR cDescription [100]`
Description.

Detailed Description

Definition at line 752 of file [windrvr.h](#).

Field Documentation

Card

`WD_CARD WD_CARD_REGISTER::Card`

Card to register.

Definition at line 754 of file [windrvr.h](#).

cDescription

`CHAR WD_CARD_REGISTER::cDescription[100]`

Description.

Definition at line 760 of file [windrvr.h](#).

cName

`CHAR WD_CARD_REGISTER::cName[32]`

Name of card.

Definition at line 759 of file [windrvr.h](#).

dwOptions

`DWORD WD_CARD_REGISTER::dwOptions`

Should be zero.

Definition at line 758 of file [windrvr.h](#).

fCheckLockOnly

`DWORD WD_CARD_REGISTER::fCheckLockOnly`

Only check if card is lockable, return hCard=1 if OK.

Definition at line 755 of file [windrvr.h](#).

hCard

`DWORD WD_CARD_REGISTER::hCard`

Handle of card.

Definition at line 757 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_DEBUG Struct Reference

```
#include <windrvr.h>
```

Data Fields

- **DWORD dwCmd**
DEBUG_COMMAND: DEBUG_STATUS, DEBUG_SET_FILTER, DEBUG_SET_BUFFER, DEBUG_CLEAR←BUFFER.
- **DWORD dwLevel**
DEBUG_LEVEL: D_ERROR, D_WARN..., or D_OFF to turn debugging off.
- **DWORD dwSection**
DEBUG_SECTION: for all sections in driver: S_ALL for partial sections: S_IO, S_MEM...
- **DWORD dwLevelMessageBox**
DEBUG_LEVEL to print in a message box.
- **DWORD dwBufferSize**
size of buffer in kernel

Detailed Description

Definition at line [978](#) of file [windrvr.h](#).

Field Documentation

dwBufferSize

`DWORD WD_DEBUG::dwBufferSize`

size of buffer in kernel

Definition at line [989](#) of file [windrvr.h](#).

dwCmd

`DWORD WD_DEBUG::dwCmd`

DEBUG_COMMAND: DEBUG_STATUS, DEBUG_SET_FILTER, DEBUG_SET_BUFFER, DEBUG_CLEAR←BUFFER.

Definition at line [980](#) of file [windrvr.h](#).

dwLevel

DWORD WD_DEBUG::dwLevel
DEBUG_LEVEL: D_ERROR, D_WARN..., or D_OFF to turn debugging off.
Definition at line 983 of file [windrvr.h](#).

dwLevelMessageBox

DWORD WD_DEBUG::dwLevelMessageBox
DEBUG_LEVEL to print in a message box.
Definition at line 987 of file [windrvr.h](#).

dwSection

DWORD WD_DEBUG::dwSection
DEBUG_SECTION: for all sections in driver: S_ALL for partial sections: S_IO, S_MEM...
Definition at line 985 of file [windrvr.h](#).
The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_DEBUG_ADD Struct Reference

#include <windrvr.h>

Data Fields

- CHAR pcBuffer [256]
- DWORD dwLevel
- DWORD dwSection

Detailed Description

Definition at line 998 of file [windrvr.h](#).

Field Documentation

dwLevel

DWORD WD_DEBUG_ADD::dwLevel
Definition at line 1001 of file [windrvr.h](#).

dwSection

DWORD WD_DEBUG_ADD::dwSection

Definition at line 1002 of file [windrvr.h](#).

pcBuffer

CHAR WD_DEBUG_ADD::pcBuffer[256]

Definition at line 1000 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_DEBUG_DUMP Struct Reference

```
#include <windrvr.h>
```

Data Fields

- CHAR cBuffer [DEBUG_USER_BUF_LEN]
buffer to receive debug messages

Detailed Description

Definition at line 993 of file [windrvr.h](#).

Field Documentation

cBuffer

CHAR WD_DEBUG_DUMP::cBuffer [DEBUG_USER_BUF_LEN]

buffer to receive debug messages

Definition at line 995 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_DMA Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD hDma
Handle of DMA buffer.

- PVOID **pUserAddr**
User address.
- KPTR **pKernelAddr**
Kernel address.
- DWORD **dwBytes**
Size of buffer.
- DWORD **dwOptions**
The first three bytes refer to WD_DMA_OPTIONS The fourth byte is used for specifying the amount of DMA bits in the requested buffer.
- DWORD **dwPages**
Number of pages in buffer.
- DWORD **hCard**
Handle of relevant card as received from WD_CardRegister()
- DMA_TRANSACTION_CALLBACK **DMATransactionCallback**
- PVOID **DMATransactionCallbackCtx**
- DWORD **dwAlignment**
required alignment, used for contiguous mode only
- DWORD **dwMaxTransferSize**
used for scatter gather mode only
- DWORD **dwTransferElementSize**
used for scatter gather mode only
- DWORD **dwBytesTransferred**
bytes transferred count
- WD_DMA_PAGE **Page** [WD_DMA_PAGES]

Detailed Description

Definition at line 508 of file [windrvr.h](#).

Field Documentation

DMATransactionCallback

DMA_TRANSACTION_CALLBACK WD_DMA::DMATransactionCallback

Definition at line 527 of file [windrvr.h](#).

DMATransactionCallbackCtx

PVOID WD_DMA::DMATransactionCallbackCtx

Definition at line 530 of file [windrvr.h](#).

dwAlignment

DWORD WD_DMA::dwAlignment

required alignment, used for contiguous mode only

Definition at line 533 of file [windrvr.h](#).

dwBytes

DWORD WD_DMA::dwBytes

Size of buffer.

Definition at line 517 of file [windrvr.h](#).

dwBytesTransferred

DWORD WD_DMA::dwBytesTransferred

bytes transferred count

Definition at line 536 of file [windrvr.h](#).

dwMaxTransferSize

DWORD WD_DMA::dwMaxTransferSize

used for scatter gather mode only

Definition at line 534 of file [windrvr.h](#).

dwOptions

DWORD WD_DMA::dwOptions

The first three bytes refer to WD_DMA_OPTIONS The fourth byte is used for specifying the amount of DMA bits in the requested buffer.

Definition at line 518 of file [windrvr.h](#).

dwPages

DWORD WD_DMA::dwPages

Number of pages in buffer.

Definition at line 521 of file [windrvr.h](#).

dwTransferElementSize

DWORD WD_DMA::dwTransferElementSize

used for scatter gather mode only

Definition at line 535 of file [windrvr.h](#).

hCard

DWORD WD_DMA::hCard

Handle of relevant card as received from [WD_CardRegister\(\)](#)

Definition at line 522 of file [windrvr.h](#).

hDma

`DWORD WD_DMA::hDma`

Handle of DMA buffer.

Definition at line 510 of file [windrvr.h](#).

Page

`WD_DMA_PAGE WD_DMA::Page [WD_DMA_PAGES]`

Definition at line 538 of file [windrvr.h](#).

pKernelAddr

`KPTR WD_DMA::pKernelAddr`

Kernel address.

Definition at line 516 of file [windrvr.h](#).

pUserAddr

`PVOID WD_DMA::pUserAddr`

User address.

Definition at line 513 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_DMA_PAGE Struct Reference

`#include <windrvr.h>`

Data Fields

- **DMA_ADDR pPhysicalAddr**

Physical address of page.

- **DWORD dwBytes**

Size of page.

Detailed Description

Definition at line 499 of file [windrvr.h](#).

Field Documentation

dwBytes

DWORD WD_DMA_PAGE::dwBytes

Size of page.

Definition at line 502 of file [windrvr.h](#).

pPhysicalAddr

DMA_ADDR WD_DMA_PAGE::pPhysicalAddr

Physical address of page.

Definition at line 501 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_EVENT Struct Reference

#include <windrvr.h>

Data Fields

- DWORD [hEvent](#)
- DWORD [dwEventType](#)
WD_EVENT_TYPE.
- DWORD [dwAction](#)
WD_EVENT_ACTION.
- DWORD [dwEventId](#)
- DWORD [hKernelPlugin](#)
- DWORD [dwOptions](#)
WD_EVENT_OPTION.
- union {
 struct {
 WD_PCI_ID [cardId](#)
 WD_PCI_SLOT [pciSlot](#)
 } [Pci](#)
 struct {
 DWORD [dwUniqueId](#)
 } [Usb](#)
 struct {
 DWORD [hIpc](#)
 Acts as a unique identifier.
 DWORD [dwSubGroupID](#)
 Might be identical to same process running twice (User implementation dependant)
 DWORD [dwGroupId](#)
 DWORD [dwSenderUID](#)
 DWORD [dwMsgID](#)
 UINT64 [qwMsgData](#)
 } [Ipc](#)
}
- DWORD [dwNumMatchTables](#)
- [WDU_MATCH_TABLE](#) [matchTables](#) [1]

Detailed Description

Definition at line 1477 of file [windrvr.h](#).

Field Documentation

cardId

`WD_PCI_ID WD_EVENT::cardId`

Definition at line 1490 of file [windrvr.h](#).

dwAction

`DWORD WD_EVENT::dwAction`

`WD_EVENT_ACTION.`

Definition at line 1482 of file [windrvr.h](#).

dwEventId

`DWORD WD_EVENT::dwEventId`

Definition at line 1483 of file [windrvr.h](#).

dwEventType

`DWORD WD_EVENT::dwEventType`

`WD_EVENT_TYPE.`

Definition at line 1480 of file [windrvr.h](#).

dwGroupID

`DWORD WD_EVENT::dwGroupID`

Definition at line 1502 of file [windrvr.h](#).

dwMsgID

`DWORD WD_EVENT::dwMsgID`

Definition at line 1505 of file [windrvr.h](#).

dwNumMatchTables

DWORD WD_EVENT::dwNumMatchTables

Definition at line 1511 of file [windrvr.h](#).

dwOptions

DWORD WD_EVENT::dwOptions

WD_EVENT_OPTION.

Definition at line 1485 of file [windrvr.h](#).

dwSenderUID

DWORD WD_EVENT::dwSenderUID

Definition at line 1504 of file [windrvr.h](#).

dwSubGroupID

DWORD WD_EVENT::dwSubGroupID

Might be identical to same process running twice (User implementation dependant)

Definition at line 1500 of file [windrvr.h](#).

dwUniqueId

DWORD WD_EVENT::dwUniqueId

Definition at line 1495 of file [windrvr.h](#).

hEvent

DWORD WD_EVENT::hEvent

Definition at line 1479 of file [windrvr.h](#).

hIpc

DWORD WD_EVENT::hIpc

Acts as a unique identifier.

Definition at line 1499 of file [windrvr.h](#).

hKernelPlugin

DWORD WD_EVENT::hKernelPlugIn

Definition at line 1484 of file [windrvr.h](#).

```
struct { ... } WD_EVENT::Ipc
```

matchTables

[WDU_MATCH_TABLE](#) WD_EVENT::matchTables[1]

Definition at line 1512 of file [windrvr.h](#).

```
struct { ... } WD_EVENT::Pci
```

pciSlot

[WD_PCI_SLOT](#) WD_EVENT::pcislot

Definition at line 1491 of file [windrvr.h](#).

qwMsgData

[UINT64](#) WD_EVENT::qwMsgData

Definition at line 1507 of file [windrvr.h](#).

```
union { ... } WD_EVENT::u
```

```
struct { ... } WD_EVENT::Usb
```

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_GET_DEVICE_PROPERTY Struct Reference

```
#include <windrvr.h>
```

Data Fields

- union {
 - HANDLE [hDevice](#)
 - DWORD [dwUniqueId](#)}
 - PVOID [pBuf](#)
 - DWORD [dwBytes](#)
 - DWORD [dwProperty](#)
 - DWORD [dwOptions](#)
- WD_GET_DEVICE_PROPERTY_OPTION.*

Detailed Description

Definition at line 1025 of file [windrvr.h](#).

Field Documentation

dwBytes

DWORD [WD_GET_DEVICE_PROPERTY::dwBytes](#)

Definition at line 1035 of file [windrvr.h](#).

dwOptions

DWORD [WD_GET_DEVICE_PROPERTY::dwOptions](#)

WD_GET_DEVICE_PROPERTY_OPTION.

Definition at line 1037 of file [windrvr.h](#).

dwProperty

DWORD [WD_GET_DEVICE_PROPERTY::dwProperty](#)

Definition at line 1036 of file [windrvr.h](#).

dwUniqueId

DWORD [WD_GET_DEVICE_PROPERTY::dwUniqueId](#)

Definition at line 1031 of file [windrvr.h](#).

union { ... } [WD_GET_DEVICE_PROPERTY::h](#)

hDevice

HANDLE WD_GET_DEVICE_PROPERTY::hDevice

Definition at line 1029 of file [windrvr.h](#).

pBuf

PVOID WD_GET_DEVICE_PROPERTY::pBuf

Definition at line 1033 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_INTERRUPT Struct Reference

#include <windrvr.h>

Data Fields

- DWORD **hInterrupt**
Handle of interrupt.
- DWORD **dwOptions**
Interrupt options: can be INTERRUPT_CMD_COPY.
- **WD_TRANSFER * Cmd**
Commands to do on interrupt.
- DWORD **dwCmds**
Number of commands.
- **WD_KERNEL_PLUGIN_CALL kpCall**
Kernel Plugin call.
- DWORD **fEnableOk**
TRUE if interrupt was enabled ([WD_IntEnable\(\)](#) succeeded).
- DWORD **dwCounter**
Number of interrupts received.
- DWORD **dwLost**
Number of interrupts not yet dealt with.
- DWORD **fStopped**
Was interrupt disabled during wait.
- DWORD **dwLastMessage**
Message data of the last received MSI (Windows)
- DWORD **dwEnabledIntType**
Interrupt type that was actually enabled.

Detailed Description

Definition at line 618 of file [windrvr.h](#).

Field Documentation

Cmd

WD_TRANSFER* WD_INTERRUPT::Cmd

Commands to do on interrupt.

Definition at line 623 of file [windrvr.h](#).

dwCmds

DWORD WD_INTERRUPT::dwCmds

Number of commands.

Definition at line 625 of file [windrvr.h](#).

dwCounter

DWORD WD_INTERRUPT::dwCounter

Number of interrupts received.

Definition at line 634 of file [windrvr.h](#).

dwEnabledIntType

DWORD WD_INTERRUPT::dwEnabledIntType

Interrupt type that was actually enabled.

Definition at line 638 of file [windrvr.h](#).

dwLastMessage

DWORD WD_INTERRUPT::dwLastMessage

Message data of the last received MSI (Windows)

Definition at line 637 of file [windrvr.h](#).

dwLost

DWORD WD_INTERRUPT::dwLost

Number of interrupts not yet dealt with.

Definition at line 635 of file [windrvr.h](#).

dwOptions

DWORD WD_INTERRUPT::dwOptions

Interrupt options: can be INTERRUPT_CMD_COPY.

Definition at line 621 of file [windrvr.h](#).

fEnableOk

DWORD WD_INTERRUPT::fEnableOk

TRUE if interrupt was enabled ([WD_IntEnable\(\)](#) succeeded).

Definition at line [630](#) of file [windrvr.h](#).

fStopped

DWORD WD_INTERRUPT::fStopped

Was interrupt disabled during wait.

Definition at line [636](#) of file [windrvr.h](#).

hInterrupt

DWORD WD_INTERRUPT::hInterrupt

Handle of interrupt.

Definition at line [620](#) of file [windrvr.h](#).

kpCall

WD_KERNEL_PLUGIN_CALL WD_INTERRUPT::kpCall

Kernel Plugin call.

Definition at line [629](#) of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_IPC_PROCESS Struct Reference

```
#include <windrvr.h>
```

Data Fields

- CHAR cProcessName [WD_PROCESS_NAME_LENGTH]
- DWORD dwSubGroupID

Identifier of the processes type.
- DWORD dwGroupID

Unique identifier of the processes group for discarding unrelated process.
- DWORD hIpc

Returned from [WD_IpcRegister\(\)](#)

Detailed Description

Definition at line [764](#) of file [windrvr.h](#).

Field Documentation

cProcessName

CHAR WD_IPC_PROCESS::cProcessName [WD_PROCESS_NAME_LENGTH]

Definition at line 766 of file [windrvr.h](#).

dwGroupID

DWORD WD_IPC_PROCESS::dwGroupID

Unique identifier of the processes group for discarding unrelated process.

WinDriver developers are encouraged to change their driver name before distribution to avoid this issue entirely.

Definition at line 768 of file [windrvr.h](#).

dwSubGroupID

DWORD WD_IPC_PROCESS::dwSubGroupID

Identifier of the processes type.

Definition at line 767 of file [windrvr.h](#).

hIpc

DWORD WD_IPC_PROCESS::hIpc

Returned from [WD_IpcRegister\(\)](#)

Definition at line 772 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_IPC_REGISTER Struct Reference

```
#include <windrvr.h>
```

Data Fields

- [WD_IPC_PROCESS proclInfo](#)
- DWORD [dwOptions](#)

Reserved for future use; set to 0.

Detailed Description

Definition at line 775 of file [windrvr.h](#).

Field Documentation

dwOptions

DWORD WD_IPC_REGISTER::dwOptions

Reserved for future use; set to 0.

Definition at line 778 of file [windrvr.h](#).

procInfo

[WD_IPC_PROCESS](#) WD_IPC_REGISTER::procInfo

Definition at line 777 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_IPC_SCAN_PROCS Struct Reference

#include <[windrvr.h](#)>

Data Fields

- DWORD [hIpc](#)
Returned from WD_IpcRegister()
- DWORD [dwNumProcs](#)
Result processes.
- [WD_IPC_PROCESS](#) [procInfo](#) [[WD_IPC_MAX_PROCS](#)]

Detailed Description

Definition at line 783 of file [windrvr.h](#).

Field Documentation

dwNumProcs

DWORD WD_IPC_SCAN_PROCS::dwNumProcs

Result processes.

Number of matching processes

Definition at line 788 of file [windrvr.h](#).

hIpc

DWORD WD_IPC_SCAN_PROCS::hIpc

Returned from [WD_IpcRegister\(\)](#)

Definition at line 785 of file [windrvr.h](#).

procInfo

WD_IPC_PROCESS WD_IPC_SCAN_PROCS::procInfo[WD_IPC_MAX_PROCS]

Definition at line 789 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_IPC_SEND Struct Reference

#include <windrvr.h>

Data Fields

- DWORD **hIpc**
Returned from [WD_IpcRegister\(\)](#)
- DWORD **dwOptions**
WD_IPC_SUBGROUP_MULTICAST, WD_IPC_UID_UNICAST, WD_IPC_MULTICAST.
- DWORD **dwRecipientID**
used only on WD_IPC_UNICAST
- DWORD **dwMsgID**
- UINT64 **qwMsgData**

Detailed Description

Definition at line 799 of file [windrvr.h](#).

Field Documentation

dwMsgID

DWORD WD_IPC_SEND::dwMsgID

Definition at line 806 of file [windrvr.h](#).

dwOptions

DWORD WD_IPC_SEND::dwOptions

WD_IPC_SUBGROUP_MULTICAST, WD_IPC_UID_UNICAST, WD_IPC_MULTICAST.

Definition at line 802 of file [windrvr.h](#).

dwRecipientID

DWORD WD_IPC_SEND::dwRecipientID

used only on WD_IPC_UNICAST

Definition at line 805 of file [windrvr.h](#).

hIpc

DWORD WD_IPC_SEND::hIpc

Returned from [WD_IpcRegister\(\)](#)

Definition at line 801 of file [windrvr.h](#).

qwMsgData

UINT64 WD_IPC_SEND::qwMsgData

Definition at line 807 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_ITEMS Struct Reference

#include <windrvr.h>

Data Fields

- DWORD item
- DWORD fNotSharable
 - ITEM_TYPE.*
- union {
 - struct {
 - PHYS_ADDR pPhysicalAddr
 - Physical address on card.*
 - UINT64 qwBytes
 - Address range.*
 - KPTR pTransAddr
 - Kernel-mode mapping of the physical base address, to be used for transfer commands; returned by [WD_CardRegister\(\)](#).*
 - UPTR pUserDirectAddr
 - User-mode mapping of the physical base address, for direct user read/write transfers; returned by [WD_CardRegister\(\)](#).*
 - DWORD dwBar
 - PCI Base Address Register number.*
 - DWORD dwOptions
 - Bitmask of WD_ITEM_MEM_OPTIONS flags.*
 - KPTR pReserved
 - Reserved for internal use.*
 - } Mem
 - ITEM_MEMORY.*
 - struct {
 - KPTR pAddr
 - Beginning of I/O address.*

```
DWORD dwBytes
    I/O range.
DWORD dwBar
    PCI Base Address Register number.
} IO
    ITEM_IO.
struct {
    DWORD dwInterrupt
        Number of interrupt to install.
    DWORD dwOptions
        Interrupt options: INTERRUPT_LATCHED – latched INTERRUPT_LEVEL_SENSITIVE – level sensitive INTERRUPT_M
    DWORD hInterrupt
        Handle of the installed interrupt; returned by WD_CardRegister()
    DWORD dwReserved1
        For internal use.
    KPTR pReserved2
        For internal use.
} Int
    ITEM_INTERRUPT.
WD_BUS Bus
    ITEM_BUS.
} I
```

Detailed Description

Definition at line 691 of file [windrvr.h](#).

Field Documentation

Bus

WD_BUS WD_ITEMS::Bus

ITEM_BUS.

Definition at line 739 of file [windrvr.h](#).

dwBar

DWORD WD_ITEMS::dwBar

PCI Base Address Register number.

Definition at line 711 of file [windrvr.h](#).

dwBytes

DWORD WD_ITEMS::dwBytes

I/O range.

Definition at line 721 of file [windrvr.h](#).

dwInterrupt

DWORD WD_ITEMS::dwInterrupt

Number of interrupt to install.

Definition at line 728 of file [windrvr.h](#).

dwOptions

DWORD WD_ITEMS::dwOptions

Bitmask of WD_ITEM_MEM_OPTIONS flags.

Interrupt options: INTERRUPT_LATCHED – latched INTERRUPT_LEVEL_SENSITIVE – level sensitive INTERRUPT_MESSAGE – Message-Signaled Interrupts (MSI) INTERRUPT_MESSAGE_X – Extended MSI (MSI-X)

Definition at line 712 of file [windrvr.h](#).

dwReserved1

DWORD WD_ITEMS::dwReserved1

For internal use.

Definition at line 736 of file [windrvr.h](#).

fNotSharable

DWORD WD_ITEMS::fNotSharable

ITEM_TYPE.

Definition at line 694 of file [windrvr.h](#).

hInterrupt

DWORD WD_ITEMS::hInterrupt

Handle of the installed interrupt; returned by [WD_CardRegister\(\)](#)

Definition at line 734 of file [windrvr.h](#).

union { ... } WD_ITEMS::I

struct { ... } WD_ITEMS::Int

ITEM_INTERRUPT.

```
struct { ... } WD_ITEMS::IO  
ITEM_IO.
```

Item

DWORD WD_ITEMS::item

Definition at line [693](#) of file [windrvr.h](#).

```
struct { ... } WD_ITEMS::Mem  
ITEM_MEMORY.
```

pAddr

KPTR WD_ITEMS::pAddr

Beginning of I/O address.

Definition at line [720](#) of file [windrvr.h](#).

pPhysicalAddr

PHYS_ADDR WD_ITEMS::pPhysicalAddr

Physical address on card.

Definition at line [700](#) of file [windrvr.h](#).

pReserved

KPTR WD_ITEMS::pReserved

Reserved for internal use.

Definition at line [714](#) of file [windrvr.h](#).

pReserved2

KPTR WD_ITEMS::pReserved2

For internal use.

Definition at line [737](#) of file [windrvr.h](#).

pTransAddr

KPTR WD_ITEMS::pTransAddr

Kernel-mode mapping of the physical base address, to be used for transfer commands; returned by [WD_CardRegister\(\)](#)

Definition at line 702 of file [windrvr.h](#).

pUserDirectAddr

UPTR WD_ITEMS::pUserDirectAddr

User-mode mapping of the physical base address, for direct user read/write transfers; returned by [WD_CardRegister\(\)](#)

Definition at line 706 of file [windrvr.h](#).

qwBytes

UINT64 WD_ITEMS::qwBytes

Address range.

Definition at line 701 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_KERNEL_BUFFER Struct Reference

#include <[windrvr.h](#)>

Data Fields

- DWORD [hKerBuf](#)
Handle of Kernel Buffer.
- DWORD [dwOptions](#)
Refer to WD_KER_BUF_OPTION.
- [UINT64](#) [qwBytes](#)
Size of buffer.
- KPTR [pKernelAddr](#)
Kernel address.
- UPT [pUserAddr](#)
User address.

Detailed Description

Definition at line 553 of file [windrvr.h](#).

Field Documentation

dwOptions

DWORD WD_KERNEL_BUFFER::dwOptions

Refer to WD_KER_BUF_OPTION.

Definition at line 556 of file [windrvr.h](#).

hKerBuf

DWORD WD_KERNEL_BUFFER::hKerBuf

Handle of Kernel Buffer.

Definition at line 555 of file [windrvr.h](#).

pKernelAddr

KPTR WD_KERNEL_BUFFER::pKernelAddr

Kernel address.

Definition at line 558 of file [windrvr.h](#).

pUserAddr

UPTR WD_KERNEL_BUFFER::pUserAddr

User address.

Definition at line 559 of file [windrvr.h](#).

qwBytes

UINT64 WD_KERNEL_BUFFER::qwBytes

Size of buffer.

Definition at line 557 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_KERNEL_PLUGIN Struct Reference

```
#include <windrvr.h>
```

Public Member Functions

- [PAD_TO_64 \(hKernelPlugIn\) PVOID pOpenData](#)

Data Fields

- DWORD [hKernelPlugIn](#)

- CHAR cDriverName [WD_MAX_KP_NAME_LENGTH]
- CHAR cDriverPath [WD_MAX_KP_NAME_LENGTH]
Should be NULL (exists for backward compatibility).

Detailed Description

Definition at line 1005 of file [windrvr.h](#).

Member Function Documentation

PAD_TO_64()

```
WD_KERNEL_PLUGIN::PAD_TO_64 (
    hKernelPlugIn )
```

Field Documentation

cDriverName

CHAR WD_KERNEL_PLUGIN::cDriverName [WD_MAX_KP_NAME_LENGTH]

Definition at line 1008 of file [windrvr.h](#).

cDriverPath

CHAR WD_KERNEL_PLUGIN::cDriverPath [WD_MAX_KP_NAME_LENGTH]

Should be NULL (exists for backward compatibility).

The driver will be searched in the operating system's drivers/modules directory.

Definition at line 1009 of file [windrvr.h](#).

hKernelPlugIn

DWORD WD_KERNEL_PLUGIN::hKernelPlugIn

Definition at line 1007 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_KERNEL_PLUGIN_CALL Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD `hKernelPlugin`
- DWORD `dwMessage`
- PVOID `pData`
- DWORD `dwResult`

Detailed Description

Definition at line 601 of file [windrvr.h](#).

Field Documentation

`dwMessage`

DWORD WD_KERNEL_PLUGIN_CALL::`dwMessage`

Definition at line 604 of file [windrvr.h](#).

`dwResult`

DWORD WD_KERNEL_PLUGIN_CALL::`dwResult`

Definition at line 607 of file [windrvr.h](#).

`hKernelPlugin`

DWORD WD_KERNEL_PLUGIN_CALL::`hKernelPlugin`

Definition at line 603 of file [windrvr.h](#).

`pData`

PVOID WD_KERNEL_PLUGIN_CALL::`pData`

Definition at line 605 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_LICENSE Struct Reference

#include <windrvr.h>

Data Fields

- CHAR `cLicense` [WD_LICENSE_LENGTH]
Buffer with license string to put.

Detailed Description

Definition at line 648 of file [windrvr.h](#).

Field Documentation

cLicense

CHAR WD_LICENSE::cLicense[WD_LICENSE_LENGTH]

Buffer with license string to put.

Definition at line 650 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_OS_INFO Struct Reference

```
#include <windrvr.h>
```

Data Fields

- CHAR cProdName [REGKEY_BUFSIZE]
- CHAR clnstallationType [REGKEY_BUFSIZE]
- CHAR cCurrentVersion [REGKEY_BUFSIZE]
- CHAR cBuild [REGKEY_BUFSIZE]
- CHAR cCsdVersion [REGKEY_BUFSIZE]
- DWORD dwMajorVersion
- DWORD dwMinorVersion

Detailed Description

Definition at line 1673 of file [windrvr.h](#).

Field Documentation

cBuild

CHAR WD_OS_INFO::cBuild[REGKEY_BUFSIZE]

Definition at line 1679 of file [windrvr.h](#).

cCsdVersion

CHAR WD_OS_INFO::cCsdVersion[REGKEY_BUFSIZE]

Definition at line 1680 of file [windrvr.h](#).

cCurrentVersion

CHAR WD_OS_INFO::cCurrentVersion[REGKEY_BUFSIZE]

Definition at line 1678 of file [windrvr.h](#).

cInstallationType

CHAR WD_OS_INFO::cInstallationType[REGKEY_BUFSIZE]

Definition at line 1676 of file [windrvr.h](#).

cProdName

CHAR WD_OS_INFO::cProdName[REGKEY_BUFSIZE]

Definition at line 1675 of file [windrvr.h](#).

dwMajorVersion

DWORD WD_OS_INFO::dwMajorVersion

Definition at line 1681 of file [windrvr.h](#).

dwMinorVersion

DWORD WD_OS_INFO::dwMinorVersion

Definition at line 1682 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_CAP Struct Reference

#include <windrvr.h>

Data Fields

- DWORD [dwCapId](#)
PCI capability ID.
- DWORD [dwCapOffset](#)
PCI capability register offset.

Detailed Description

Definition at line 866 of file [windrvr.h](#).

Field Documentation

dwCapId

DWORD WD_PCI_CAP::dwCapId

PCI capability ID.

Definition at line 868 of file [windrvr.h](#).

dwCapOffset

DWORD WD_PCI_CAP::dwCapOffset

PCI capability register offset.

Definition at line 869 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_CARD_INFO Struct Reference

#include <windrvr.h>

Data Fields

- [WD_PCI_SLOT](#) pciSlot

PCI slot information.

- [WD_CARD](#) Card

Card information.

Detailed Description

Definition at line 899 of file [windrvr.h](#).

Field Documentation

Card

[WD_CARD](#) WD_PCI_CARD_INFO::Card

Card information.

Definition at line 902 of file [windrvr.h](#).

pciSlot

`WD_PCI_SLOT` `WD_PCI_CARD_INFO::pcislot`

PCI slot information.

Definition at line 901 of file `windrvr.h`.

The documentation for this struct was generated from the following file:

- `windrvr.h`

WD_PCI_CONFIG_DUMP Struct Reference

```
#include <windrvr.h>
```

Data Fields

- `WD_PCI_SLOT` `pciSlot`
PCI slot information.
- `PVOID` `pBuffer`
Pointer to a read/write data buffer.
- `DWORD` `dwOffset`
PCI configuration space offset to read/write.
- `DWORD` `dwBytes`
Input – number of bytes to read/write; Output – number of bytes read/written.
- `DWORD` `fIsRead`
1 – read data; 0 – write data
- `DWORD` `dwResult`
PCI_ACCESS_RESULT.

Detailed Description

Definition at line 913 of file `windrvr.h`.

Field Documentation

dwBytes

`DWORD` `WD_PCI_CONFIG_DUMP::dwBytes`

Input – number of bytes to read/write; Output – number of bytes read/written.

Definition at line 920 of file `windrvr.h`.

dwOffset

`DWORD` `WD_PCI_CONFIG_DUMP::dwOffset`

PCI configuration space offset to read/write.

Definition at line 919 of file `windrvr.h`.

dwResult

DWORD WD_PCI_CONFIG_DUMP::dwResult

PCI_ACCESS_RESULT.

Definition at line 923 of file [windrvr.h](#).

fIsRead

DWORD WD_PCI_CONFIG_DUMP::fIsRead

1 – read data; 0 – write data

Definition at line 922 of file [windrvr.h](#).

pBuffer

PVOID WD_PCI_CONFIG_DUMP::pBuffer

Pointer to a read/write data buffer.

Definition at line 916 of file [windrvr.h](#).

pciSlot

WD_PCI_SLOT WD_PCI_CONFIG_DUMP::pciSlot

PCI slot information.

Definition at line 915 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_ID Struct Reference

#include <[windrvr.h](#)>

Data Fields

- DWORD dwVendorId
- DWORD dwDeviceId

Detailed Description

Definition at line 833 of file [windrvr.h](#).

Field Documentation

dwDeviceId

DWORD WD_PCI_ID::dwDeviceId

Definition at line 836 of file [windrvr.h](#).

dwVendorId

DWORD WD_PCI_ID::dwVendorId

Definition at line 835 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_SCAN_CAPS Struct Reference

```
#include <windrvr.h>
```

Data Fields

- [WD_PCI_SLOT](#) pciSlot
Scan Parameters.
- DWORD dwCapId
PCI capability ID to search for, or WD_PCI_CAP_ID_ALL to scan all PCI capabilities.
- DWORD dwOptions
Scan options – WD_PCI_SCAN_CAPS_OPTIONS; default – WD_PCI_SCAN_CAPS_BASIC.
- DWORD dwNumCaps
Scan Results.
- [WD_PCI_CAP](#) pciCaps [[WD_PCI_MAX_CAPS](#)]
Array of matching PCI capabilities.

Detailed Description

Definition at line 878 of file [windrvr.h](#).

Field Documentation

dwCapId

DWORD WD_PCI_SCAN_CAPS::dwCapId

PCI capability ID to search for, or WD_PCI_CAP_ID_ALL to scan all PCI capabilities.

Definition at line 882 of file [windrvr.h](#).

dwNumCaps

DWORD WD_PCI_SCAN_CAPS::dwNumCaps

Scan Results.

Number of matching PCI capabilities

Definition at line 888 of file [windrvr.h](#).

dwOptions

DWORD WD_PCI_SCAN_CAPS::dwOptions

Scan options – WD_PCI_SCAN_CAPS_OPTIONS; default – WD_PCI_SCAN_CAPS_BASIC.

Definition at line 884 of file [windrvr.h](#).

pciCaps

WD_PCI_CAP WD_PCI_SCAN_CAPS::pciCaps[WD_PCI_MAX_CAPS]

Array of matching PCI capabilities.

Definition at line 889 of file [windrvr.h](#).

pciSlot

WD_PCI_SLOT WD_PCI_SCAN_CAPS::pcislot

Scan Parameters.

PCI slot information

Definition at line 881 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_SCAN_CARDS Struct Reference

#include <windrvr.h>

Data Fields

- [WD_PCI_ID](#) searchId
 - Scan Parameters.*
- DWORD dwCards
 - Number of matching PCI cards.*
- [WD_PCI_ID](#) cardId [WD_PCI_CARDS]
 - Scan Results.*
- [WD_PCI_SLOT](#) cardSlot [WD_PCI_CARDS]
 - Array of matching PCI slots info.*
- DWORD dwOptions
 - Scan Options.*

Detailed Description

Definition at line 839 of file [windrvr.h](#).

Field Documentation

cardId

`WD_PCI_ID WD_PCI_SCAN_CARDS::cardId[WD_PCI_CARDS]`

Scan Results.

Array of matching card IDs

Definition at line 848 of file [windrvr.h](#).

cardSlot

`WD_PCI_SLOT WD_PCI_SCAN_CARDS::cardSlot [WD_PCI_CARDS]`

Array of matching PCI slots info.

Definition at line 849 of file [windrvr.h](#).

dwCards

`DWORD WD_PCI_SCAN_CARDS::dwCards`

Number of matching PCI cards.

Definition at line 845 of file [windrvr.h](#).

dwOptions

`DWORD WD_PCI_SCAN_CARDS::dwOptions`

Scan Options.

Scan options – `WD_PCI_SCAN_OPTIONS`

Definition at line 852 of file [windrvr.h](#).

searchId

`WD_PCI_ID WD_PCI_SCAN_CARDS::searchId`

Scan Parameters.

PCI vendor and/or device IDs to search for;

- `dwVendorId==0` – scan all PCI vendor IDs;
- `dwDeviceId==0` – scan all PCI device IDs

Definition at line 842 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_SLOT Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD [dwDomain](#)
Domain number.
- DWORD [dwBus](#)
Bus number.
- DWORD [dwSlot](#)
Slot number.
- DWORD [dwFunction](#)
Function number.

Detailed Description

Definition at line [824](#) of file [windrvr.h](#).

Field Documentation

dwBus

```
DWORD WD_PCI_SLOT::dwBus
```

Bus number.

Definition at line [828](#) of file [windrvr.h](#).

dwDomain

```
DWORD WD_PCI_SLOT::dwDomain
```

Domain number.

currently only applicable for Linux systems. Zero by default

Definition at line [826](#) of file [windrvr.h](#).

dwFunction

```
DWORD WD_PCI_SLOT::dwFunction
```

Function number.

Definition at line [830](#) of file [windrvr.h](#).

dwSlot

DWORD WD_PCI_SLOT::dwSlot

Slot number.

Definition at line 829 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_PCI_SRIOV Struct Reference

```
#include <windrvr.h>
```

Data Fields

- **WD_PCI_SLOT pciSlot**
PCI slot information.
- **DWORD dwNumVFs**
Number of Virtual Functions.

Detailed Description

Definition at line 893 of file [windrvr.h](#).

Field Documentation

dwNumVFs

DWORD WD_PCI_SRIOV::dwNumVFs

Number of Virtual Functions.

Definition at line 896 of file [windrvr.h](#).

pciSlot

[WD_PCI_SLOT](#) WD_PCI_SRIOV::pcislot

PCI slot information.

Definition at line 895 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_SLEEP Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD **dwMicroSeconds**
Sleep time in Micro Seconds (1/1,000,000 Second)
- DWORD **dwOptions**
can be: SLEEP_NON_BUSY (10000 uSec +)

Detailed Description

Definition at line 927 of file [windrvr.h](#).

Field Documentation

dwMicroSeconds

DWORD WD_SLEEP::dwMicroSeconds
Sleep time in Micro Seconds (1/1,000,000 Second)
Definition at line 929 of file [windrvr.h](#).

dwOptions

DWORD WD_SLEEP::dwOptions
can be: SLEEP_NON_BUSY (10000 uSec +)
Definition at line 931 of file [windrvr.h](#).
The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_TRANSFER Struct Reference

```
#include <windrvr.h>
```

Data Fields

- KPTR pPort
I/O port for transfer or kernel memory address.
- DWORD cmdTrans
Transfer command WD_TRANSFER_CMD.
- DWORD dwBytes
For string transfer.
- DWORD fAutoinc
Transfer from one port/address or use incremental range of addresses.
- DWORD dwOptions
Must be 0.

```
• union {
    BYTE Byte
        Use for 8 bit transfer.
    WORD Word
        Use for 16 bit transfer.
    UINT32 Dword
        Use for 32 bit transfer.
    UINT64 Qword
        Use for 64 bit transfer.
    PVOID pBuffer
        Use for string transfer.
} Data
```

Detailed Description

Definition at line 563 of file [windrvr.h](#).

Field Documentation

Byte

`BYTE WD_TRANSFER::Byte`

Use for 8 bit transfer.

Definition at line 575 of file [windrvr.h](#).

cmdTrans

`DWORD WD_TRANSFER::cmdTrans`

Transfer command WD_TRANSFER_CMD.

Definition at line 566 of file [windrvr.h](#).

```
union { ... } WD_TRANSFER::Data
```

dwBytes

`DWORD WD_TRANSFER::dwBytes`

For string transfer.

Definition at line 569 of file [windrvr.h](#).

dwOptions

`DWORD WD_TRANSFER::dwOptions`

Must be 0.

Definition at line 572 of file [windrvr.h](#).

Dword

`UINT32 WD_TRANSFER::Dword`

Use for 32 bit transfer.

Definition at line 577 of file [windrvr.h](#).

fAutoinc

`DWORD WD_TRANSFER::fAutoinc`

Transfer from one port/address or use incremental range of addresses.

Definition at line 570 of file [windrvr.h](#).

pBuffer

`PVOID WD_TRANSFER::pBuffer`

Use for string transfer.

Definition at line 579 of file [windrvr.h](#).

pPort

`KPTR WD_TRANSFER::pPort`

I/O port for transfer or kernel memory address.

Definition at line 565 of file [windrvr.h](#).

Qword

`UINT64 WD_TRANSFER::Qword`

Use for 64 bit transfer.

Definition at line 578 of file [windrvr.h](#).

Word

`WORD WD_TRANSFER::Word`

Use for 16 bit transfer.

Definition at line 576 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_USAGE Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD [applications_num](#)
- DWORD [devices_num](#)

Detailed Description

Definition at line 1515 of file [windrvr.h](#).

Field Documentation

applications_num

DWORD WD_USAGE::applications_num

Definition at line 1517 of file [windrvr.h](#).

devices_num

DWORD WD_USAGE::devices_num

Definition at line 1518 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WD_VERSION Struct Reference

```
#include <windrvr.h>
```

Data Fields

- DWORD [dwVer](#)
- CHAR [cVer \[WD_VERSION_STR_LENGTH\]](#)

Detailed Description

Definition at line 641 of file [windrvr.h](#).

Field Documentation

cVer

```
CHAR WD_VERSION::cVer[WD_VERSION_STR_LENGTH]
```

Definition at line 644 of file [windrvr.h](#).

dwVer

```
DWORD WD_VERSION::dwVer
```

Definition at line 643 of file [windrvr.h](#).

The documentation for this struct was generated from the following file:

- [windrvr.h](#)

WDC_ADDR_DESC Struct Reference

Address space information struct.

```
#include <wdc_defs.h>
```

Data Fields

- DWORD [dwAddrSpace](#)
Address space number.
- BOOL [fIsMemory](#)
TRUE: memory address space; FALSE: I/O.
- DWORD [dwItemIndex](#)
Index of address space in the pDev->cardReg.Card.Item array.
- DWORD [reserved](#)
Size of address space.
- UINT64 [qwBytes](#)
- KPTR [pAddr](#)
I/O / Memory kernel mapped address – for [WD_Transfer\(\)](#), [WD_MultiTransfer\(\)](#), or direct kernel access.

Detailed Description

Address space information struct.

Definition at line 27 of file [wdc_defs.h](#).

Field Documentation

dwAddrSpace

```
DWORD WDC_ADDR_DESC::dwAddrSpace
```

Definition at line 28 of file [wdc_defs.h](#).

dwItemIndex

DWORD WDC_ADDR_DESC::dwItemIndex

TRUE: memory address space; FALSE: I/O.

Definition at line 30 of file [wdc_defs.h](#).

fIsMemory

BOOL WDC_ADDR_DESC::fIsMemory

Address space number.

Definition at line 29 of file [wdc_defs.h](#).

pAddr

KPTR WDC_ADDR_DESC::pAddr

Size of address space.

Definition at line 34 of file [wdc_defs.h](#).

pUserDirectMemAddr

UPTR WDC_ADDR_DESC::pUserDirectMemAddr

I/O / Memory kernel mapped address – for [WD_Transfer\(\)](#), [WD_MultiTransfer\(\)](#), or direct kernel access.

Definition at line 37 of file [wdc_defs.h](#).

qwBytes

UINT64 WDC_ADDR_DESC::qwBytes

Definition at line 33 of file [wdc_defs.h](#).

reserved

DWORD WDC_ADDR_DESC::reserved

Index of address space in the pDev->cardReg.Card.Item array.

Definition at line 32 of file [wdc_defs.h](#).

The documentation for this struct was generated from the following file:

- [wdc_defs.h](#)

WDC_DEVICE Struct Reference

Device information struct.

```
#include <wdc_defs.h>
```

Data Fields

- **WD_PCI_ID id**
PCI device ID.
- **WD_PCI_SLOT slot**
PCI device slot location information.
- **DWORD dwNumAddrSpaces**
Total number of device's address spaces.
- **WDC_ADDR_DESC * pAddrDesc**
Array of device's address spaces information.
- **WD_CARD_REGISTER cardReg**
Device's resources information.
- **WD_KERNEL_PLUGIN kerPlug**
Kernel Plugin information.
- **WD_INTERRUPT Int**
Interrupt information.
- **HANDLE hIntThread**
Event information.
- **WD_EVENT Event**
Event information.
- **HANDLE hEvent**
Event information.
- **PVOID pCtx**

Detailed Description

Device information struct.

Definition at line 46 of file [wdc_defs.h](#).

Field Documentation

cardReg

`WD_CARD_REGISTER WDC_DEVICE::cardReg`

Array of device's address spaces information.

Definition at line 56 of file [wdc_defs.h](#).

dwNumAddrSpaces

`DWORD WDC_DEVICE::dwNumAddrSpaces`

PCI device slot location information.

Definition at line 50 of file [wdc_defs.h](#).

Event

`WD_EVENT WDC_DEVICE::Event`

Definition at line 64 of file [wdc_defs.h](#).

hEvent

```
HANDLE WDC_DEVICE::hEvent
```

Event information.

Definition at line 65 of file [wdc_defs.h](#).

hIntThread

```
HANDLE WDC_DEVICE::hIntThread
```

Interrupt information.

Definition at line 61 of file [wdc_defs.h](#).

id

```
WD_PCI_ID WDC_DEVICE::id
```

Definition at line 47 of file [wdc_defs.h](#).

Int

```
WD_INTERRUPT WDC_DEVICE::Int
```

Kernel Plugin information.

Definition at line 60 of file [wdc_defs.h](#).

kerPlug

```
WD_KERNEL_PLUGIN WDC_DEVICE::kerPlug
```

Device's resources information.

Definition at line 58 of file [wdc_defs.h](#).

pAddrDesc

```
WDC_ADDR_DESC* WDC_DEVICE::pAddrDesc
```

Total number of device's address spaces.

Definition at line 53 of file [wdc_defs.h](#).

pCtx

```
PVOID WDC_DEVICE::pCtx
```

Definition at line 68 of file [wdc_defs.h](#).

slot

`WD_PCI_SLOT WDC_DEVICE::slot`

PCI device ID.

Definition at line 48 of file [wdc_defs.h](#).

The documentation for this struct was generated from the following file:

- [wdc_defs.h](#)

WDC_INTERRUPT_PARAMS Struct Reference

```
#include <wdc_lib.h>
```

Data Fields

- `WD_TRANSFER * pTransCmds`
- `DWORD dwNumCmds`
- `DWORD dwOptions`
- `INT_HANDLER funcIntHandler`
- `PVOID pData`
- `BOOL fUseKP`

Detailed Description

Definition at line 1482 of file [wdc_lib.h](#).

Field Documentation

dwNumCmds

`DWORD WDC_INTERRUPT_PARAMS::dwNumCmds`

Definition at line 1484 of file [wdc_lib.h](#).

dwOptions

`DWORD WDC_INTERRUPT_PARAMS::dwOptions`

Definition at line 1485 of file [wdc_lib.h](#).

funcIntHandler

`INT_HANDLER WDC_INTERRUPT_PARAMS::funcIntHandler`

Definition at line 1486 of file [wdc_lib.h](#).

fUseKP

```
BOOL WDC_INTERRUPT_PARAMS::fUseKP
```

Definition at line 1488 of file [wdc_lib.h](#).

pData

```
PVOID WDC_INTERRUPT_PARAMS::pData
```

Definition at line 1487 of file [wdc_lib.h](#).

pTransCmds

```
WD_TRANSFER* WDC_INTERRUPT_PARAMS::pTransCmds
```

Definition at line 1483 of file [wdc_lib.h](#).

The documentation for this struct was generated from the following file:

- [wdc_lib.h](#)

WDC_PCI_SCAN_CAPS_RESULT Struct Reference

PCI capabilities scan results.

```
#include <wdc_lib.h>
```

Data Fields

- DWORD [dwNumCaps](#)
Number of matching PCI capabilities.
- [WD_PCI_CAP](#) [pciCaps](#) [[WD_PCI_MAX_CAPS](#)]
Array of matching PCI capabilities.

Detailed Description

PCI capabilities scan results.

Definition at line 46 of file [wdc_lib.h](#).

Field Documentation

dwNumCaps

```
DWORD WDC_PCI_SCAN_CAPS_RESULT::dwNumCaps
```

Number of matching PCI capabilities.

Definition at line 47 of file [wdc_lib.h](#).

pciCaps

`WD_PCI_CAP WDC_PCI_SCAN_CAPS_RESULT::pciCaps[WD_PCI_MAX_CAPS]`

Array of matching PCI capabilities.

Definition at line 48 of file [wdc_lib.h](#).

The documentation for this struct was generated from the following file:

- [wdc_lib.h](#)

WDC_PCI_SCAN_RESULT Struct Reference

PCI scan results.

```
#include <wdc_lib.h>
```

Data Fields

- **DWORD dwNumDevices**
Number of matching devices.
- **WD_PCI_ID devicId [WD_PCI_CARDS]**
Array of matching device IDs.
- **WD_PCI_SLOT deviceSlot [WD_PCI_CARDS]**
Array of matching device locations.

Detailed Description

PCI scan results.

Definition at line 37 of file [wdc_lib.h](#).

Field Documentation

devicId

`WD_PCI_ID WDC_PCI_SCAN_RESULT::deviceId[WD_PCI_CARDS]`

Array of matching device IDs.

Definition at line 39 of file [wdc_lib.h](#).

deviceSlot

`WD_PCI_SLOT WDC_PCI_SCAN_RESULT::deviceSlot[WD_PCI_CARDS]`

Array of matching device locations.

Definition at line 40 of file [wdc_lib.h](#).

dwNumDevices

DWORD WDC_PCI_SCAN_RESULT::dwNumDevices

Number of matching devices.

Definition at line 38 of file [wdc.lib.h](#).

The documentation for this struct was generated from the following file:

- [wdc.lib.h](#)

WDS_IPC_MSG_RX Struct Reference

IPC message received.

```
#include <wds_lib.h>
```

Data Fields

- DWORD dwSenderId
WinDriver IPC unique ID of the sending process.
- DWORD dwMsgID
A 32 bit unique number defined by the user application.
- UINT64 qwMsgData
Optional - 64 bit additional data from the sending user-application process.

Detailed Description

IPC message received.

Definition at line 43 of file [wds.lib.h](#).

Field Documentation

dwMsgID

DWORD WDS_IPC_MSG_RX::dwMsgID

A 32 bit unique number defined by the user application.

This number should be known to all user-applications that work under WinDriver IPC and share the same group ID

Definition at line 45 of file [wds.lib.h](#).

dwSenderId

DWORD WDS_IPC_MSG_RX::dwSenderId

WinDriver IPC unique ID of the sending process.

Definition at line 44 of file [wds.lib.h](#).

qwMsgData

`UINT64 WDS_IPC_MSG_RX::qwMsgData`

Optional - 64 bit additional data from the sending user-application process.

Definition at line 50 of file [wds_lib.h](#).

The documentation for this struct was generated from the following file:

- [wds_lib.h](#)

WDS_IPC_SCAN_RESULT Struct Reference

IPC scan processes results.

```
#include <wds_lib.h>
```

Data Fields

- `DWORD dwNumProcs`
Number of matching processes.
- `WD_IPC_PROCESS procInfo [WD_IPC_MAX_PROCS]`
Array of processes info.

Detailed Description

IPC scan processes results.

Definition at line 37 of file [wds_lib.h](#).

Field Documentation

dwNumProcs

`DWORD WDS_IPC_SCAN_RESULT::dwNumProcs`

Number of matching processes.

Definition at line 38 of file [wds_lib.h](#).

procInfo

`WD_IPC_PROCESS WDS_IPC_SCAN_RESULT::procInfo [WD_IPC_MAX_PROCS]`

Array of processes info.

Definition at line 39 of file [wds_lib.h](#).

The documentation for this struct was generated from the following file:

- [wds_lib.h](#)

WDU_ALTERNATE_SETTING Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- **WDU_INTERFACE_DESCRIPTOR** Descriptor
Interface descriptor information structure.
- **WDU_ENDPOINT_DESCRIPTOR * pEndpointDescriptors**
Pointer to the beginning of an array of endpoint descriptor information structures for the alternate setting's endpoints.
- **WDU_PIPE_INFO * pPipes**
Pointer to the beginning of an array of pipe information structures for the alternate setting's pipes.

Detailed Description

Definition at line 209 of file [windrvr_usb.h](#).

Field Documentation

Descriptor

`WDU_INTERFACE_DESCRIPTOR WDU_ALTERNATE_SETTING::Descriptor`

Interface descriptor information structure.

Definition at line 211 of file [windrvr_usb.h](#).

pEndpointDescriptors

`WDU_ENDPOINT_DESCRIPTOR* WDU_ALTERNATE_SETTING::pEndpointDescriptors`

Pointer to the beginning of an array of endpoint descriptor information structures for the alternate setting's endpoints.

Definition at line 214 of file [windrvr_usb.h](#).

pPipes

`WDU_PIPE_INFO* WDU_ALTERNATE_SETTING::pPipes`

Pointer to the beginning of an array of pipe information structures for the alternate setting's pipes.

Definition at line 221 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_CONFIGURATION Struct Reference

`#include <windrvr_usb.h>`

Data Fields

- **WDU_CONFIGURATION_DESCRIPTOR** Descriptor
Configuration descriptor information structure.
- **DWORD dwNumInterfaces**

Number of interfaces supported by this configuration.

- **WDU_INTERFACE * pInterfaces**

Pointer to the beginning of an array of interface information structures for the configuration's interfaces.

Detailed Description

Definition at line 246 of file [windrvr_usb.h](#).

Field Documentation

Descriptor

`WDU_CONFIGURATION_DESCRIPTOR WDU_CONFIGURATION::Descriptor`

Configuration descriptor information structure.

Definition at line 248 of file [windrvr_usb.h](#).

dwNumInterfaces

`DWORD WDU_CONFIGURATION::dwNumInterfaces`

Number of interfaces supported by this configuration.

Definition at line 250 of file [windrvr_usb.h](#).

pInterfaces

`WDU_INTERFACE* WDU_CONFIGURATION::pInterfaces`

Pointer to the beginning of an array of interface information structures for the configuration's interfaces.

Definition at line 252 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_CONFIGURATION_DESCRIPTOR Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- **UCHAR bLength**
Size, in bytes, of the descriptor.
- **UCHAR bDescriptorType**
Configuration descriptor (0x02)
- **USHORT wTotalLength**
Total length, in bytes, of data returned.
- **UCHAR bNumInterfaces**

- **UCHAR bConfigurationValue**
Configuration number.
- **UCHAR iConfiguration**
Index of string descriptor that describes this configuration.
- **UCHAR bmAttributes**
Power settings for this configuration:
 - **UCHAR MaxPower**

Detailed Description

Definition at line 171 of file [windrvr_usb.h](#).

Field Documentation

bConfigurationValue

`UCHAR WDU_CONFIGURATION_DESCRIPTOR::bConfigurationValue`

Configuration number.

Definition at line 177 of file [windrvr_usb.h](#).

bDescriptorType

`UCHAR WDU_CONFIGURATION_DESCRIPTOR::bDescriptorType`

Configuration descriptor (0x02)

Definition at line 174 of file [windrvr_usb.h](#).

bLength

`UCHAR WDU_CONFIGURATION_DESCRIPTOR::bLength`

Size, in bytes, of the descriptor.

Definition at line 173 of file [windrvr_usb.h](#).

bmAttributes

`UCHAR WDU_CONFIGURATION_DESCRIPTOR::bmAttributes`

Power settings for this configuration:

- self-powered
- remote wakeup (allows device to wake up the host)

Definition at line 180 of file [windrvr_usb.h](#).

bNumInterfaces

UCHAR WDU_CONFIGURATION_DESCRIPTOR::bNumInterfaces

Number of interfaces.

Definition at line 176 of file [windrvr_usb.h](#).

iConfiguration

UCHAR WDU_CONFIGURATION_DESCRIPTOR::iConfiguration

Index of string descriptor that describes this configuration.

Definition at line 178 of file [windrvr_usb.h](#).

MaxPower

UCHAR WDU_CONFIGURATION_DESCRIPTOR::MaxPower

Definition at line 184 of file [windrvr_usb.h](#).

wTotalLength

USHORT WDU_CONFIGURATION_DESCRIPTOR::wTotalLength

Total length, in bytes, of data returned.

Definition at line 175 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_DEVICE Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- **WDU_DEVICE_DESCRIPTOR Descriptor**
CDevice descriptor information structure.
- **WDU_PIPE_INFO Pipe0**
Pipe information structure for the device's default control pipe (pipe 0)
- **WDU_CONFIGURATION * pConfigs**
Pointer to the beginning of an array of configuration information structures describing the device's configurations.
- **WDU_CONFIGURATION * pActiveConfig**
Pointer to the device's active configuration information structure, stored in the pConfigs array.
- **WDU_INTERFACE * pActiveInterface [WD_USB_MAX_INTERFACES]**
Array of pointers to interface information structures; the non-NULL elements in the array represent the device's active interfaces.

Detailed Description

Definition at line 258 of file [windrvr_usb.h](#).

Field Documentation

Descriptor

`WDU_DEVICE_DESCRIPTOR WDU_DEVICE::Descriptor`

CDevice descriptor information structure.

Definition at line 259 of file [windrvr_usb.h](#).

pActiveConfig

`WDU_CONFIGURATION* WDU_DEVICE::pActiveConfig`

Pointer to the device's active configuration information structure, stored in the pConfigs array.

Definition at line 267 of file [windrvr_usb.h](#).

pActiveInterface

`WDU_INTERFACE* WDU_DEVICE::pActiveInterface[WD_USB_MAX_INTERFACES]`

Array of pointers to interface information structures; the non-NULL elements in the array represent the device's active interfaces.

On Windows, the number of active interfaces is the number of interfaces supported by the active configuration, as stored in the pActiveConfig->dwNumInterfaces field. On Linux, the number of active interfaces is currently always 1, because the WDU_ATTACH_CALLBACK device-attach callback is called for each device interface.

Definition at line 272 of file [windrvr_usb.h](#).

pConfigs

`WDU_CONFIGURATION* WDU_DEVICE::pConfigs`

Pointer to the beginning of an array of configuration information structures describing the device's configurations.

Definition at line 263 of file [windrvr_usb.h](#).

Pipe0

`WDU_PIPE_INFO WDU_DEVICE::Pipe0`

Pipe information structure for the device's default control pipe (pipe 0)

Definition at line 261 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_DEVICE_DESCRIPTOR Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- UCHAR **bLength**
Size, in bytes, of the descriptor (18 bytes)
- UCHAR **bDescriptorType**
Device descriptor (0x01)
- USHORT **bcdUSB**
Number of the USB specification with which the device complies.
- UCHAR **bDeviceClass**
The device's class.
- UCHAR **bDeviceSubClass**
The device's sub-class.
- UCHAR **bDeviceProtocol**
The device's protocol.
- UCHAR **bMaxPacketSize0**
Maximum size of transferred packets.
- USHORT **idVendor**
Vendor ID, as assigned by USB-IF.
- USHORT **idProduct**
Product ID, as assigned by the product manufacturer.
- USHORT **bcdDevice**
Device release numbe.
- UCHAR **iManufacturer**
Index of manufacturer string descriptor.
- UCHAR **iProduct**
Index of product string descriptor.
- UCHAR **iSerialNumber**
Index of serial number string descriptor.
- UCHAR **bNumConfigurations**
Number of possible configurations.

Detailed Description

Definition at line 188 of file [windrvr_usb.h](#).

Field Documentation

bcdDevice

```
USHORT WDU_DEVICE_DESCRIPTOR::bcdDevice
```

Device release numbe.

Definition at line 202 of file [windrvr_usb.h](#).

bcdUSB

USHORT WDU_DEVICE_DESCRIPTOR::bcdUSB

Number of the USB specification with which the device complies.

Definition at line 192 of file [windrvr_usb.h](#).

bDescriptorType

UCHAR WDU_DEVICE_DESCRIPTOR::bDescriptorType

Device descriptor (0x01)

Definition at line 191 of file [windrvr_usb.h](#).

bDeviceClass

UCHAR WDU_DEVICE_DESCRIPTOR::bDeviceClass

The device's class.

Definition at line 194 of file [windrvr_usb.h](#).

bDeviceProtocol

UCHAR WDU_DEVICE_DESCRIPTOR::bDeviceProtocol

The device's protocol.

Definition at line 196 of file [windrvr_usb.h](#).

bDeviceSubClass

UCHAR WDU_DEVICE_DESCRIPTOR::bDeviceSubClass

The device's sub-class.

Definition at line 195 of file [windrvr_usb.h](#).

bLength

UCHAR WDU_DEVICE_DESCRIPTOR::bLength

Size, in bytes, of the descriptor (18 bytes)

Definition at line 190 of file [windrvr_usb.h](#).

bMaxPacketSize0

UCHAR WDU_DEVICE_DESCRIPTOR::bMaxPacketSize0

Maximum size of transferred packets.

Definition at line 197 of file [windrvr_usb.h](#).

bNumConfigurations

UCHAR WDU_DEVICE_DESCRIPTOR::bNumConfigurations

Number of possible configurations.

Definition at line 206 of file [windrvr_usb.h](#).

idProduct

USHORT WDU_DEVICE_DESCRIPTOR::idProduct

Product ID, as assigned by the product manufacturer.

Definition at line 200 of file [windrvr_usb.h](#).

idVendor

USHORT WDU_DEVICE_DESCRIPTOR::idVendor

Vendor ID, as assigned by USB-IF.

Definition at line 199 of file [windrvr_usb.h](#).

iManufacturer

UCHAR WDU_DEVICE_DESCRIPTOR::iManufacturer

Index of manufacturer string descriptor.

Definition at line 203 of file [windrvr_usb.h](#).

iProduct

UCHAR WDU_DEVICE_DESCRIPTOR::iProduct

Index of product string descriptor.

Definition at line 204 of file [windrvr_usb.h](#).

iSerialNumber

UCHAR WDU_DEVICE_DESCRIPTOR::iSerialNumber

Index of serial number string descriptor.

Definition at line 205 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_ENDPOINT_DESCRIPTOR Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- UCHAR **bLength**
Size, in bytes, of the descriptor (7 bytes)
- UCHAR **bDescriptorType**
Endpoint descriptor (0x05)
- UCHAR **bEndpointAddress**
Endpoint address: Use bits 0–3 for endpoint number, set bits 4–6 to zero (0), and set bit 7 to zero (0) for outbound data and to one (1) for inbound data (ignored for control endpoints).
- UCHAR **bmAttributes**
Specifies the transfer type for this endpoint (control, interrupt, isochronous or bulk).
- USHORT **wMaxPacketSize**
Maximum size of packets this endpoint can send or receive.
- UCHAR **bInterval**
Interval, in frame counts, for polling endpoint data transfers.

Detailed Description

Definition at line 151 of file [windrvr_usb.h](#).

Field Documentation

bDescriptorType

UCHAR WDU_ENDPOINT_DESCRIPTOR::bDescriptorType

Endpoint descriptor (0x05)

Definition at line 154 of file [windrvr_usb.h](#).

bEndpointAddress

UCHAR WDU_ENDPOINT_DESCRIPTOR::bEndpointAddress

Endpoint address: Use bits 0–3 for endpoint number, set bits 4–6 to zero (0), and set bit 7 to zero (0) for outbound data and to one (1) for inbound data (ignored for control endpoints).

Definition at line 155 of file [windrvr_usb.h](#).

bInterval

UCHAR WDU_ENDPOINT_DESCRIPTOR::bInterval

Interval, in frame counts, for polling endpoint data transfers.

Ignored for bulk and control endpoints. Must equal 1 for isochronous endpoints. May range from 1 to 255 for interrupt endpoints.

Definition at line 165 of file [windrvr_usb.h](#).

bLength

UCHAR WDU_ENDPOINT_DESCRIPTOR::bLength

Size, in bytes, of the descriptor (7 bytes)

Definition at line 153 of file [windrvr_usb.h](#).

bmAttributes

UCHAR WDU_ENDPOINT_DESCRIPTOR::bmAttributes

Specifies the transfer type for this endpoint (control, interrupt, isochronous or bulk).

See the USB specification for further information.

Definition at line 160 of file [windrvr_usb.h](#).

wMaxPacketSize

USHORT WDU_ENDPOINT_DESCRIPTOR::wMaxPacketSize

Maximum size of packets this endpoint can send or receive.

Definition at line 163 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_EVENT_TABLE Struct Reference

```
#include <wdu_lib.h>
```

Data Fields

- [WDU_ATTACH_CALLBACK](#) pfDeviceAttach
- [WDU_DETACH_CALLBACK](#) pfDeviceDetach
- [WDU_POWER_CHANGE_CALLBACK](#) pfPowerChange
- PVOID pUserData

Detailed Description

Definition at line 63 of file [wdu_lib.h](#).

Field Documentation

pfDeviceAttach

[WDU_ATTACH_CALLBACK](#) WDU_EVENT_TABLE::pfDeviceAttach

Definition at line 65 of file [wdu_lib.h](#).

pfDeviceDetach

`WDU_DETACH_CALLBACK WDU_EVENT_TABLE::pfDeviceDetach`

Definition at line 66 of file [wdu_lib.h](#).

pfPowerChange

`WDU_POWER_CHANGE_CALLBACK WDU_EVENT_TABLE::pfPowerChange`

Definition at line 67 of file [wdu_lib.h](#).

pUserData

`PVOID WDU_EVENT_TABLE::pUserData`

Definition at line 68 of file [wdu_lib.h](#).

The documentation for this struct was generated from the following file:

- [wdu_lib.h](#)

WDU_GET_DESCRIPTOR Struct Reference

`#include <windrvr_usb.h>`

Data Fields

- `DWORD dwUniqueId`
- `UCHAR bType`
- `UCHAR bIndex`
- `USHORT wLength`
- `PVOID pBuffer`
- `USHORT wLanguage`

Detailed Description

Definition at line 441 of file [windrvr_usb.h](#).

Field Documentation

bIndex

`UCHAR WDU_GET_DESCRIPTOR::bIndex`

Definition at line 445 of file [windrvr_usb.h](#).

bType

UCHAR WDU_GET_DESCRIPTOR::bType

Definition at line 444 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_GET_DESCRIPTOR::dwUniqueId

Definition at line 443 of file [windrvr_usb.h](#).

pBuffer

PVOID WDU_GET_DESCRIPTOR::pBuffer

Definition at line 447 of file [windrvr_usb.h](#).

wLanguage

USHORT WDU_GET_DESCRIPTOR::wLanguage

Definition at line 448 of file [windrvr_usb.h](#).

wLength

USHORT WDU_GET_DESCRIPTOR::wLength

Definition at line 446 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_GET_DEVICE_DATA Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- PVOID pBuf
- DWORD dwBytes
- DWORD dwOptions

Detailed Description

Definition at line 308 of file [windrvr_usb.h](#).

Field Documentation

dwBytes

DWORD WDU_GET_DEVICE_DATA::dwBytes

Definition at line 314 of file [windrvr_usb.h](#).

dwOptions

DWORD WDU_GET_DEVICE_DATA::dwOptions

Definition at line 315 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_GET_DEVICE_DATA::dwUniqueId

Definition at line 310 of file [windrvr_usb.h](#).

pBuf

PVOID WDU_GET_DEVICE_DATA::pBuf

Definition at line 312 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_HALT_TRANSFER Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwPipeNum
- DWORD dwOptions

Detailed Description

Definition at line 391 of file [windrvr_usb.h](#).

Field Documentation

dwOptions

DWORD WDU_HALT_TRANSFER::dwOptions

Definition at line 395 of file [windrvr_usb.h](#).

dwPipeNum

DWORD WDU_HALT_TRANSFER::dwPipeNum

Definition at line 394 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_HALT_TRANSFER::dwUniqueId

Definition at line 393 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_INTERFACE Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- [WDU_ALTERNATE_SETTING](#) * pAlternateSettings
Pointer to the beginning of an array of alternate setting information structures for the interface's alternate settings.
- DWORD dwNumAltSettings
Pointer to the beginning of an array of endpoint descriptor information structures for the alternate setting's endpoints.
- [WDU_ALTERNATE_SETTING](#) * pActiveAltSetting
Pointer to the beginning of an array of pipe information structures for the alternate setting's pipes.

Detailed Description

Definition at line 227 of file [windrvr_usb.h](#).

Field Documentation

dwNumAltSettings

DWORD WDU_INTERFACE::dwNumAltSettings

Pointer to the beginning of an array of endpoint descriptor information structures for the alternate setting's endpoints.

Definition at line 235 of file [windrvr_usb.h](#).

pActiveAltSetting

[WDU_ALTERNATE_SETTING](#)* WDU_INTERFACE::pActiveAltSetting

Pointer to the beginning of an array of pipe information structures for the alternate setting's pipes.

Definition at line 239 of file [windrvr_usb.h](#).

pAlternateSettings

`WDU_ALTERNATE_SETTING* WDU_INTERFACE::pAlternateSettings`

Pointer to the beginning of an array of alternate setting information structures for the interface's alternate settings.

Definition at line 229 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_INTERFACE_DESCRIPTOR Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- UCHAR **bLength**
Size, in bytes, of the descriptor (9 bytes)
- UCHAR **bDescriptorType**
Interface descriptor (0x04)
- UCHAR **blInterfaceNumber**
Interface number.
- UCHAR **bAlternateSetting**
Alternate setting number.
- UCHAR **bNumEndpoints**
Number of endpoints used by this interface.
- UCHAR **blInterfaceClass**
The interface's class code, as assigned by USB-IF.
- UCHAR **blInterfaceSubClass**
The interface's sub-class code, as assigned by USB-IF.
- UCHAR **blInterfaceProtocol**
The interface's protocol code, as assigned by USB-IF.
- UCHAR **iInterface**
Index of string descriptor that describes this interface.

Detailed Description

Definition at line 134 of file [windrvr_usb.h](#).

Field Documentation

bAlternateSetting

`UCHAR WDU_INTERFACE_DESCRIPTOR::bAlternateSetting`

Alternate setting number.

Definition at line 139 of file [windrvr_usb.h](#).

bDescriptorType

UCHAR WDU_INTERFACE_DESCRIPTOR::bDescriptorType

Interface descriptor (0x04)

Definition at line 137 of file [windrvr_usb.h](#).

bInterfaceClass

UCHAR WDU_INTERFACE_DESCRIPTOR::bInterfaceClass

The interface's class code, as assigned by USB-IF.

Definition at line 141 of file [windrvr_usb.h](#).

bInterfaceNumber

UCHAR WDU_INTERFACE_DESCRIPTOR::bInterfaceNumber

Interface number.

Definition at line 138 of file [windrvr_usb.h](#).

bInterfaceProtocol

UCHAR WDU_INTERFACE_DESCRIPTOR::bInterfaceProtocol

The interface's protocol code, as assigned by USB-IF.

Definition at line 145 of file [windrvr_usb.h](#).

bInterfaceSubClass

UCHAR WDU_INTERFACE_DESCRIPTOR::bInterfaceSubClass

The interface's sub-class code, as assigned by USB-IF.

Definition at line 143 of file [windrvr_usb.h](#).

bLength

UCHAR WDU_INTERFACE_DESCRIPTOR::bLength

Size, in bytes, of the descriptor (9 bytes)

Definition at line 136 of file [windrvr_usb.h](#).

bNumEndpoints

UCHAR WDU_INTERFACE_DESCRIPTOR::bNumEndpoints

Number of endpoints used by this interface.

Definition at line 140 of file [windrvr_usb.h](#).

iInterface

UCHAR WDU_INTERFACE_DESCRIPTOR::iInterface

Index of string descriptor that describes this interface.

Definition at line 147 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_MATCH_TABLE Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- USHORT wVendorId
Required USB Vendor ID to detect, as assigned by USB-IF ()*
- USHORT wProductId
Required USB Product ID to detect, as assigned by the product manufacturer ()*
- UCHAR bDeviceClass
The device's class code, as assigned by USB-IF ()*
- UCHAR bDeviceSubClass
The device's sub-class code, as assigned by USB-IF ()*
- UCHAR bInterfaceClass
The interface's class code, as assigned by USB-IF ()*
- UCHAR bInterfaceSubClass
The interface's sub-class code, as assigned by USB-IF ()*
- UCHAR bInterfaceProtocol
The interface's protocol code, as assigned by USB-IF ()*

Detailed Description

Definition at line 290 of file [windrvr_usb.h](#).

Field Documentation

bDeviceClass

UCHAR WDU_MATCH_TABLE::bDeviceClass

The device's class code, as assigned by USB-IF (*)

Definition at line 296 of file [windrvr_usb.h](#).

bDeviceSubClass

UCHAR WDU_MATCH_TABLE::bDeviceSubClass

The device's sub-class code, as assigned by USB-IF (*)

Definition at line 298 of file [windrvr_usb.h](#).

bInterfaceClass

UCHAR WDU_MATCH_TABLE::bInterfaceClass

The interface's class code, as assigned by USB-IF (*)

Definition at line 300 of file [windrvr_usb.h](#).

bInterfaceProtocol

UCHAR WDU_MATCH_TABLE::bInterfaceProtocol

The interface's protocol code, as assigned by USB-IF (*)

Definition at line 304 of file [windrvr_usb.h](#).

bInterfaceSubClass

UCHAR WDU_MATCH_TABLE::bInterfaceSubClass

The interface's sub-class code, as assigned by USB-IF (*)

Definition at line 302 of file [windrvr_usb.h](#).

wProductId

USHORT WDU_MATCH_TABLE::wProductId

Required USB Product ID to detect, as assigned by the product manufacturer (*)

Definition at line 294 of file [windrvr_usb.h](#).

wVendorId

USHORT WDU_MATCH_TABLE::wVendorId

Required USB Vendor ID to detect, as assigned by USB-IF (*)

Definition at line 292 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_PIPE_INFO Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD **dwNumber**
Pipe number; zero for the default control pipe.
- DWORD **dwMaximumPacketSize**
Maximum size of packets that can be transferred using this pipe.
- DWORD **type**
Transfer type for this pipe.
- DWORD **direction**
Direction of the transfer:
- DWORD **dwInterval**
Interval in milliseconds.

Detailed Description

Definition at line 120 of file [windrvr_usb.h](#).

Field Documentation

direction

`DWORD WDU_PIPE_INFO::direction`

Direction of the transfer:

- `WDU_DIR_IN` or `WDU_DIR_OUT` for isochronous, bulk or interrupt pipes.
- `WDU_DIR_IN_OUT` for control pipes.

Definition at line 126 of file [windrvr_usb.h](#).

dwInterval

`DWORD WDU_PIPE_INFO::dwInterval`

Interval in milliseconds.

Relevant only to interrupt pipes.

Definition at line 130 of file [windrvr_usb.h](#).

dwMaximumPacketSize

`DWORD WDU_PIPE_INFO::dwMaximumPacketSize`

Maximum size of packets that can be transferred using this pipe.

Definition at line 123 of file [windrvr_usb.h](#).

dwNumber

DWORD WDU_PIPE_INFO::dwNumber

Pipe number; zero for the default control pipe.

Definition at line 122 of file [windrvr_usb.h](#).

type

DWORD WDU_PIPE_INFO::type

Transfer type for this pipe.

Definition at line 125 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_RESET_DEVICE Struct Reference

#include <windrvr_usb.h>

Data Fields

- DWORD dwUniqueID
- DWORD dwOptions

Detailed Description

Definition at line 410 of file [windrvr_usb.h](#).

Field Documentation

dwOptions

DWORD WDU_RESET_DEVICE::dwOptions

Definition at line 413 of file [windrvr_usb.h](#).

dwUniqueID

DWORD WDU_RESET_DEVICE::dwUniqueID

Definition at line 412 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_RESET_PIPE Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwPipeNum
- DWORD dwOptions

Detailed Description

Definition at line 374 of file [windrvr_usb.h](#).

Field Documentation

dwOptions

DWORD WDU_RESET_PIPE::dwOptions

Definition at line 378 of file [windrvr_usb.h](#).

dwPipeNum

DWORD WDU_RESET_PIPE::dwPipeNum

Definition at line 377 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_RESET_PIPE::dwUniqueId

Definition at line 376 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_SELECTIVE_SUSPEND Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwOptions

Detailed Description

Definition at line 404 of file [windrvr_usb.h](#).

Field Documentation

dwOptions

DWORD WDU_SELECTIVE_SUSPEND::dwOptions

Definition at line 407 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_SELECTIVE_SUSPEND::dwUniqueId

Definition at line 406 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_SET_INTERFACE Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwInterfaceNum
- DWORD dwAlternateSetting
- DWORD dwOptions

Detailed Description

Definition at line 366 of file [windrvr_usb.h](#).

Field Documentation

dwAlternateSetting

DWORD WDU_SET_INTERFACE::dwAlternateSetting

Definition at line 370 of file [windrvr_usb.h](#).

dwInterfaceNum

DWORD WDU_SET_INTERFACE::dwInterfaceNum

Definition at line 369 of file [windrvr_usb.h](#).

dwOptions

DWORD WDU_SET_INTERFACE::dwOptions

Definition at line 371 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_SET_INTERFACE::dwUniqueId

Definition at line 368 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_STREAM Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwOptions
- DWORD dwPipeNum
- DWORD dwBufferSize
- DWORD dwRxSize
- BOOL fBlocking
- DWORD dwRxTxTimeout
- DWORD dwReserved

Detailed Description

Definition at line 451 of file [windrvr_usb.h](#).

Field Documentation

dwBufferSize

DWORD WDU_STREAM::dwBufferSize

Definition at line 456 of file [windrvr_usb.h](#).

dwOptions

DWORD WDU_STREAM::dwOptions

Definition at line 454 of file [windrvr_usb.h](#).

dwPipeNum

DWORD WDU_STREAM::dwPipeNum

Definition at line 455 of file [windrvr_usb.h](#).

dwReserved

DWORD WDU_STREAM::dwReserved

Definition at line 460 of file [windrvr_usb.h](#).

dwRxSize

DWORD WDU_STREAM::dwRxSize

Definition at line 457 of file [windrvr_usb.h](#).

dwRxTxTimeout

DWORD WDU_STREAM::dwRxTxTimeout

Definition at line 459 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_STREAM::dwUniqueId

Definition at line 453 of file [windrvr_usb.h](#).

fBlocking

BOOL WDU_STREAM::fBlocking

Definition at line 458 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_STREAM_STATUS Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueId
- DWORD dwOptions
- BOOL fIsRunning
- DWORD dwLastError
- DWORD dwBytesInBuffer
- DWORD dwReserved

Detailed Description

Definition at line 463 of file [windrvr_usb.h](#).

Field Documentation

dwBytesInBuffer

DWORD WDU_STREAM_STATUS::dwBytesInBuffer

Definition at line 469 of file [windrvr_usb.h](#).

dwLastError

DWORD WDU_STREAM_STATUS::dwLastError

Definition at line 468 of file [windrvr_usb.h](#).

dwOptions

DWORD WDU_STREAM_STATUS::dwOptions

Definition at line 466 of file [windrvr_usb.h](#).

dwReserved

DWORD WDU_STREAM_STATUS::dwReserved

Definition at line 470 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_STREAM_STATUS::dwUniqueId

Definition at line 465 of file [windrvr_usb.h](#).

fIsRunning

BOOL WDU_STREAM_STATUS::fIsRunning

Definition at line 467 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_TRANSFER Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD **dwUniqueID**
- DWORD **dwPipeNum**

Pipe number on device.
- DWORD **fRead**

TRUE for read (IN) transfers; FALSE for write (OUT) transfers.
- DWORD **dwOptions**

USB_TRANSFER options: USB_ISOCH_FULL_PACKETS_ONLY - For isochronous transfers only.
- PVOID **pBuffer**

Pointer to buffer to read/write.
- DWORD **dwBufferSize**

Amount of bytes to transfer.
- DWORD **dwBytesTransferred**

Returns the number of bytes actually read/written.
- UCHAR **SetupPacket [8]**

Setup packet for control pipe transfer.
- DWORD **dwTimeout**

Timeout for the transfer in milliseconds.

Detailed Description

Definition at line 416 of file [windrvr_usb.h](#).

Field Documentation

dwBufferSize

DWORD WDU_TRANSFER::dwBufferSize

Amount of bytes to transfer.

Definition at line 432 of file [windrvr_usb.h](#).

dwBytesTransferred

DWORD WDU_TRANSFER::dwBytesTransferred

Returns the number of bytes actually read/written.

Definition at line 433 of file [windrvr_usb.h](#).

dwOptions

DWORD WDU_TRANSFER::dwOptions

USB_TRANSFER options: USB_ISOCH_FULL_PACKETS_ONLY - For isochronous transfers only.

If set, only full packets will be transmitted and the transfer function will return when the amount of bytes left to transfer is less than the maximum packet size for the pipe (the function will return without transmitting the remaining bytes).

Definition at line 422 of file [windrvr_usb.h](#).

dwPipeNum

DWORD WDU_TRANSFER::dwPipeNum

Pipe number on device.

Definition at line 419 of file [windrvr_usb.h](#).

dwTimeout

DWORD WDU_TRANSFER::dwTimeout

Timeout for the transfer in milliseconds.

Set to 0 for infinite wait.

Definition at line 436 of file [windrvr_usb.h](#).

dwUniqueId

DWORD WDU_TRANSFER::dwUniqueId

Definition at line 418 of file [windrvr_usb.h](#).

fRead

DWORD WDU_TRANSFER::fRead

TRUE for read (IN) transfers; FALSE for write (OUT) transfers.

Definition at line 420 of file [windrvr_usb.h](#).

pBuffer

PVOID WDU_TRANSFER::pBuffer

Pointer to buffer to read/write.

Definition at line 430 of file [windrvr_usb.h](#).

SetupPacket

UCHAR WDU_TRANSFER::SetupPacket[8]

Setup packet for control pipe transfer.

Definition at line 435 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

WDU_WAKEUP Struct Reference

```
#include <windrvr_usb.h>
```

Data Fields

- DWORD dwUniqueID
- DWORD dwOptions

Detailed Description

Definition at line 398 of file [windrvr_usb.h](#).

Field Documentation

dwOptions

DWORD WDU_WAKEUP::dwOptions

Definition at line 401 of file [windrvr_usb.h](#).

dwUniqueID

DWORD WDU_WAKEUP::dwUniqueID

Definition at line 400 of file [windrvr_usb.h](#).

The documentation for this struct was generated from the following file:

- [windrvr_usb.h](#)

Chapter 21

File Documentation

bits.h File Reference

Enumerations

- enum **BIT** {
 BIT0 = 0x00000001 , BIT1 = 0x00000002 , BIT2 = 0x00000004 , BIT3 = 0x00000008 ,
 BIT4 = 0x00000010 , BIT5 = 0x00000020 , BIT6 = 0x00000040 , BIT7 = 0x00000080 ,
 BIT8 = 0x00000100 , BIT9 = 0x00000200 , BIT10 = 0x00000400 , BIT11 = 0x00000800 ,
 BIT12 = 0x00001000 , BIT13 = 0x00002000 , BIT14 = 0x00004000 , BIT15 = 0x00008000 ,
 BIT16 = 0x00010000 , BIT17 = 0x00020000 , BIT18 = 0x00040000 , BIT19 = 0x00080000 ,
 BIT20 = 0x00100000 , BIT21 = 0x00200000 , BIT22 = 0x00400000 , BIT23 = 0x00800000 ,
 BIT24 = 0x01000000 , BIT25 = 0x02000000 , BIT26 = 0x04000000 , BIT27 = 0x08000000 ,
 BIT28 = 0x10000000 , BIT29 = 0x20000000 , BIT30 = 0x40000000 , BIT31 = (int)0x80000000 }

Enumeration Type Documentation

BIT

enum **BIT**

Enumerator

BIT0
BIT1
BIT2
BIT3
BIT4
BIT5
BIT6
BIT7
BIT8
BIT9
BIT10
BIT11
BIT12
BIT13
BIT14
BIT15
BIT16
BIT17
BIT18
BIT19
BIT20
BIT21
BIT22
BIT23

Enumerator

BIT24	
BIT25	
BIT26	
BIT27	
BIT28	
BIT29	
BIT30	
BIT31	

Definition at line 10 of file [bits.h](#).

bits.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 /*
00004  * File - bits.h
00005 */
00006
00007 #ifndef _BITS_H_
00008 #define _BITS_H_
00009
00010 typedef enum
00011 {
00012     BIT0  = 0x00000001,
00013     BIT1  = 0x00000002,
00014     BIT2  = 0x00000004,
00015     BIT3  = 0x00000008,
00016     BIT4  = 0x00000010,
00017     BIT5  = 0x00000020,
00018     BIT6  = 0x00000040,
00019     BIT7  = 0x00000080,
00020     BIT8  = 0x00000100,
00021     BIT9  = 0x00000200,
00022     BIT10 = 0x00000400,
00023     BIT11 = 0x00000800,
00024     BIT12 = 0x00001000,
00025     BIT13 = 0x00002000,
00026     BIT14 = 0x00004000,
00027     BIT15 = 0x00008000,
00028     BIT16 = 0x00010000,
00029     BIT17 = 0x00020000,
00030     BIT18 = 0x00040000,
00031     BIT19 = 0x00080000,
00032     BIT20 = 0x00100000,
00033     BIT21 = 0x00200000,
00034     BIT22 = 0x00400000,
00035     BIT23 = 0x00800000,
00036     BIT24 = 0x01000000,
00037     BIT25 = 0x02000000,
00038     BIT26 = 0x04000000,
00039     BIT27 = 0x08000000,
00040     BIT28 = 0x10000000,
00041     BIT29 = 0x20000000,
00042     BIT30 = 0x40000000,
00043     BIT31 = (int)0x80000000
00044 } BIT;
00045
00046 #endif /* _BITS_H_ */
```

cstring.h File Reference

```
#include <stdarg.h>
```

Data Structures

- class [CCString](#)

Typedefs

- [typedef const char * PCSTR](#)

Functions

- [const CCString operator+ \(const CCString &str1, const CCString &str2\)](#)
- [int strcmp \(PCSTR str1, PCSTR str2\)](#)

Typeface Documentation

PCSTR

```
typedef const char* PCSTR
```

Definition at line 8 of file [cstring.h](#).

Function Documentation

operator+()

```
const CCString operator+ (
    const CCString & str1,
    const CCString & str2 )
```

strcmp()

```
int strcmp (
    PCSTR str1,
    PCSTR str2 )
```

cstring.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _CSTRING_H_
00004 #define _CSTRING_H_
00005
00006 #include <stdarg.h>
00007
00008 typedef const char* PCSTR;
00009 class CCString
00010 {
00011 public:
00012     // Constructors
00013     CCString();
00014     CCString(const CCString& stringSrc);
00015     CCString(PCSTR pcwStr);
```

```
00016     virtual ~CCString();
00017
00018     operator PCSTR() const;
00019     operator char *() const;
00020
00021     // MFC compatibility functions
00022     // overloaded assignment
00023     const int operator==(const CCString& s);
00024     const int operator==(const char *s);
00025     const int operator!=(const CCString& s);
00026     const int operator!=(const char *s);
00027     const CCString& operator=(const CCString& s);
00028     const CCString& operator=(const PCSTR s);
00029     const CCString& operator=(PCSTR s);
00030     const CCString& operator+=(const PCSTR s);
00031     const CCString& cat_printf(const PCSTR format, ...);
00032     char& operator[](int i);
00033     int Compare(PCSTR s);
00034     int CompareNoCase(PCSTR s);
00035     const int operator!=(char *s);
00036     void MakeUpper();
00037     void MakeLower();
00038
00039     // sub-string extraction
00040     CCString Mid(int nFirst, int nCount);
00041     CCString Mid(int nFirst);
00042
00043     CCString StrRemove(PCSTR str);
00044     CCString StrReplace(PCSTR str, PCSTR new_str);
00045
00046     int Length() const;
00047     bool contains(PCSTR substr);
00048     bool is_empty();
00049     char *m_str; // The String itself
00050
00051     void Format(const PCSTR format, ...);
00052     int IsAllocOK();
00053     int GetBuffer(unsigned long size);
00054
00055     // ANSI C compatibility functions
00056     int strcmp(const PCSTR s);
00057     int stricmp(const PCSTR s);
00058     void sprintf(const PCSTR format, ...);
00059     void toupper();
00060     void tolower();
00061     CCString tolower_copy(void) const;
00062     CCString toupper_copy(void) const;
00063     int find_first(char c) const;
00064     int find_last(char c) const;
00065     CCString substr(int start, int end) const;
00066     CCString trim(int index) const;
00067
00068 protected:
00069     void Init();
00070     void vsprintf(const PCSTR format, va_list ap);
00071     int m_buf_size;
00072 };
00073
00074 const CCString operator+(const CCString &str1, const CCString &str2);
00075
00076 #if !defined(WIN32)
00077     int stricmp(PCSTR str1, PCSTR str2);
00078#endif
00079
00080#endif /* _CSTRING_H_ */
00081
```

kplib.h File Reference

Macros

- `#define __KERNEL__`
- `#define COPY_FROM_USER_OR_KERNEL(dst, src, n, fKernelMode)`
Macro for copying data from the user mode to the Kernel Plugin.
- `#define COPY_TO_USER_OR_KERNEL(dst, src, n, fKernelMode)`
Macro copying data from the Kernel Plugin to user mode.
- `#define FALSE 0`
- `#define TRUE 1`

- #define NULL 0UL

Typedefs

- typedef struct _KP_SPINLOCK KP_SPINLOCK
Kernel PlugIn spinlock object structure.
- typedef volatile int KP_INTERLOCKED
a Kernel PlugIn interlocked operations counter

Functions

- KP_SPINLOCK * kp_spinlock_init (void)
Initializes a new Kernel PlugIn spinlock object.
- void kp_spinlock_wait (KP_SPINLOCK *spinlock)
Waits on a Kernel PlugIn spinlock object.
- void kp_spinlock_release (KP_SPINLOCK *spinlock)
Releases a Kernel PlugIn spinlock object.
- void kp_spinlock_uninit (KP_SPINLOCK *spinlock)
Uninitializes a Kernel PlugIn spinlock object.
- void kp_interlocked_init (KP_INTERLOCKED *target)
Initializes a Kernel PlugIn interlocked counter.
- void kp_interlocked_uninit (KP_INTERLOCKED *target)
Uninitializes a Kernel PlugIn interlocked counter.
- int kp_interlocked_increment (KP_INTERLOCKED *target)
Increments the value of a Kernel PlugIn interlocked counter by one.
- int kp_interlocked_decrement (KP_INTERLOCKED *target)
Decrements the value of a Kernel PlugIn interlocked counter by one.
- int kp_interlocked_add (KP_INTERLOCKED *target, int val)
Adds a specified value to the current value of a Kernel PlugIn interlocked counter.
- int kp_interlocked_read (KP_INTERLOCKED *target)
Reads to the value of a Kernel PlugIn interlocked counter.
- void kp_interlocked_set (KP_INTERLOCKED *target, int val)
Sets the value of a Kernel PlugIn interlocked counter to the specified value.
- int kp_interlocked_exchange (KP_INTERLOCKED *target, int val)
Sets the value of a Kernel PlugIn interlocked counter to the specified value and returns the previous value of the counter.
- int __cdecl KDBG (DWORD dwLevel, DWORD dwSection, const char *format,...)
- char *__cdecl strcpy (char *s1, const char *s2)
- void *__cdecl malloc (unsigned long size)
- void __cdecl free (void *buf)

Macro Definition Documentation

__KERNEL__

```
#define __KERNEL__
```

Definition at line 7 of file kpstdlib.h.

COPY_FROM_USER_OR_KERNEL

```
#define COPY_FROM_USER_OR_KERNEL( dst, src, n, fKernelMode )
```

Value:

```
{ \\\n    if (OS_needs_copy_from_user(fKernelMode)) \\\n        COPY_FROM_USER(dst, src, n); \\\n    else \\\n        memcpy(dst, src, n); \\\n}
```

Macro for copying data from the user mode to the Kernel PlugIn.

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. Note that if you wish to access the user-mode data from within the Kernel PlugIn interrupt handler functions, you should first copy the data into some variable in the Kernel PlugIn before the execution of the kernel-mode interrupt handler routines. To safely share a data buffer between the user-mode and Kernel PlugIn routines (e.g., KP_IntAtIrql() and KP_IntAtDpc()), 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.

Definition at line 239 of file [kpstdlib.h](#).

COPY_TO_USER_OR_KERNEL

```
#define COPY_TO_USER_OR_KERNEL( dst, src, n, fKernelMode )
```

Value:

```
{ \\\n    if (OS_needs_copy_from_user(fKernelMode)) \\\n        COPY_TO_USER(dst, src, n); \\\n    else \\\n        memcpy(dst, src, n); \\\n}
```

Macro copying data from the Kernel PlugIn to user mode.

See [COPY_FROM_USER_OR_KERNEL\(\)](#) for more info.

Definition at line 251 of file [kpstdlib.h](#).

FALSE

```
#define FALSE 0
```

Definition at line 260 of file [kpstdlib.h](#).

NULL

```
#define NULL 0UL
```

Definition at line [268](#) of file [kpstdlib.h](#).

TRUE

```
#define TRUE 1
```

Definition at line [264](#) of file [kpstdlib.h](#).

Typedef Documentation

KP_INTERLOCKED

```
typedef volatile int KP_INTERLOCKED
```

a Kernel Plugin interlocked operations counter

Definition at line [73](#) of file [kpstdlib.h](#).

KP_SPINLOCK

```
typedef struct _KP_SPINLOCK KP_SPINLOCK
```

Kernel Plugin spinlock object structure.

Definition at line [25](#) of file [kpstdlib.h](#).

Function Documentation

free()

```
void __cdecl free (
    void * buf )
```

KDBG()

```
int __cdecl KDBG (
    DWORD dwLevel,
    DWORD dwSection,
    const char * format,
    ... )
```

kp_interlocked_add()

```
int kp_interlocked_add (
    KP_INTERLOCKED * target,
    int val )
```

Adds a specified value to the current value of a Kernel PlugIn interlocked counter.

Parameters

in, out	<i>target</i>	Pointer to the Kernel PlugIn interlocked counter to which to add
in	<i>val</i>	The value to add to the interlocked counter (target)

Returns

Returns the new value of the interlocked counter (target).

kp_interlocked_decrement()

```
int kp_interlocked_decrement (
    KP_INTERLOCKED * target )
```

Decrements the value of a Kernel PlugIn interlocked counter by one.

Parameters

in, out	<i>target</i>	Pointer to the Kernel PlugIn interlocked counter to decrement
---------	---------------	---

Returns

Returns the new value of the interlocked counter (target).

kp_interlocked_exchange()

```
int kp_interlocked_exchange (
    KP_INTERLOCKED * target,
    int val )
```

Sets the value of a Kernel PlugIn interlocked counter to the specified value and returns the previous value of the counter.

Parameters

in, out	<i>target</i>	Pointer to the Kernel PlugIn interlocked counter to exchange
in	<i>val</i>	The new value to set for the interlocked counter (target)

Returns

Returns the previous value of the interlocked counter (target).

kp_interlocked_increment()

```
int kp_interlocked_increment (
    KP_INTERLOCKED * target )
```

Increments the value of a Kernel PlugIn interlocked counter by one.

Parameters

in, out	target	Pointer to the Kernel PlugIn interlocked counter to increment
---------	--------	---

Returns

Returns the new value of the interlocked counter (target).

kp_interlocked_init()

```
void kp_interlocked_init (
    KP_INTERLOCKED * target )
```

Initializes a Kernel PlugIn interlocked counter.

Parameters

in, out	target	Pointer to the Kernel PlugIn interlocked counter to initialize
---------	--------	--

Returns

None

kp_interlocked_read()

```
int kp_interlocked_read (
    KP_INTERLOCKED * target )
```

Reads to the value of a Kernel PlugIn interlocked counter.

Parameters

in	target	Pointer to the Kernel PlugIn interlocked counter to read
----	--------	--

Returns

Returns the value of the interlocked counter (target).

kp_interlocked_set()

```
void kp_interlocked_set (
    KP_INTERLOCKED * target,
    int val )
```

Sets the value of a Kernel PlugIn interlocked counter to the specified value.

Parameters

in, out	<i>target</i>	Pointer to the Kernel PlugIn interlocked counter to set
in	<i>val</i>	The value to set for the interlocked counter (target)

Returns

None

kp_interlocked_uninit()

```
void kp_interlocked_uninit (
    KP_INTERLOCKED * target )
```

Uninitializes a Kernel PlugIn interlocked counter.

Parameters

in, out	<i>target</i>	Pointer to the Kernel PlugIn interlocked counter to uninitialized
---------	---------------	---

Returns

None

kp_spinlock_init()

```
KP_SPINLOCK * kp_spinlock_init (
    void )
```

Initializes a new Kernel PlugIn spinlock object.

Returns

If successful, returns a pointer to the new Kernel PlugIn spinlock object, otherwise returns NULL.

kp_spinlock_release()

```
void kp_spinlock_release (
    KP_SPINLOCK * spinlock )
```

Releases a Kernel PlugIn spinlock object.

Parameters

in	<i>spinlock</i>	Pointer to the Kernel PlugIn spinlock object on which to release
----	-----------------	--

Returns

None

kp_spinlock_uninit()

```
void kp_spinlock_uninit (
    KP_SPINLOCK * spinlock )
```

Uninitializes a Kernel PlugIn spinlock object.

Parameters

in	<i>spinlock</i>	Pointer to the Kernel PlugIn spinlock object on which to uninitialized
----	-----------------	--

Returns

None

kp_spinlock_wait()

```
void kp_spinlock_wait (
    KP_SPINLOCK * spinlock )
```

Waits on a Kernel PlugIn spinlock object.

Parameters

in	<i>spinlock</i>	Pointer to the Kernel PlugIn spinlock object on which to wait
----	-----------------	---

Returns

None

malloc()

```
void *__cdecl malloc (
    unsigned long size )
```

strcpy()

```
char *__cdecl strcpy (
    char * s1,
    const char * s2 )
```

kpstdlib.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _KPSTDLIB_H_
00004 #define _KPSTDLIB_H_
00005
00006 #ifndef __KERNEL__
00007     #define __KERNEL__
00008 #endif
00009
00010 #if !defined(UNIX) && defined(LINUX)
00011     #define UNIX
00012 #endif
00013
00014 #if defined(UNIX)
00015     #include "windrvr.h" // for use of KDBG DWORD parameter.
00016 #endif
00017
00018 #ifdef __cplusplus
00019 extern "C" {
00020 #endif
00021
00022 /* Spinlocks and interlocked operations */
00023
00025 typedef struct _KP_SPINLOCK KP_SPINLOCK;
00034 KP_SPINLOCK *kp_spinlock_init(void);
00035
00046 void kp_spinlock_wait(KP_SPINLOCK *spinlock);
00047
00058 void kp_spinlock_release(KP_SPINLOCK *spinlock);
00059
00070 void kp_spinlock_uninit(KP_SPINLOCK *spinlock);
00071
00073 typedef volatile int KP_INTERLOCKED;
00074
00085 void kp_interlocked_init(KP_INTERLOCKED *target);
00086
00097 void kp_interlocked_uninit(KP_INTERLOCKED *target);
00098
00109 int kp_interlocked_increment(KP_INTERLOCKED *target);
00110
00121 int kp_interlocked_decrement(KP_INTERLOCKED *target);
00122
00136 int kp_interlocked_add(KP_INTERLOCKED *target, int val);
00137
00148 int kp_interlocked_read(KP_INTERLOCKED *target);
00149
00162 void kp_interlocked_set(KP_INTERLOCKED *target, int val);
00163
00178 int kp_interlocked_exchange(KP_INTERLOCKED *target, int val);
00179
00180 #if defined(WINNT) || defined(WIN32)
00181     #if defined(_WIN64) && !defined(KERNEL_64BIT)
00182         #define KERNEL_64BIT
00183     #endif
00184     typedef unsigned long ULONG;
00185     typedef unsigned short USHORT;
00186     typedef unsigned char UCHAR;
00187     typedef long LONG;
00188     typedef short SHORT;
00189     typedef char CHAR;
00190     typedef ULONG DWORD;
00191     typedef DWORD *PDWORD;
00192     typedef unsigned char *PBYTE;
00193     typedef USHORT WORD;
00194     typedef void *PVOID;
00195     typedef char *PCHAR;
00196     typedef PVOID HANDLE;
00197     typedef ULONG BOOL;
00198     #ifndef WINAPI
00199         #define WINAPI
00200     #endif
00201 #elif defined(UNIX)
00202     #ifndef __cdecl
00203         #define __cdecl
00204     #endif
00205 #endif
00206
00207 #if defined(WINNT) || defined(WIN32)
00208     #define OS_needs_copy_from_user(fKernelMode) FALSE
00209     #define COPY_FROM_USER(dst,src,n) memcpy(dst,src,n)
00210     #define COPY_TO_USER(dst,src,n) memcpy(dst,src,n)
00211 #elif defined(LINUX)
00212     #define OS_needs_copy_from_user(fKernelMode) (!fKernelMode)
```

```
00213     #define COPY_FROM_USER(dst,src,n) LINUX_copy_from_user(dst,src,n)
00214     #define COPY_TO_USER(dst,src,n) LINUX_copy_to_user(dst,src,n)
00215 #endif
00216
00239 #define COPY_FROM_USER_OR_KERNEL(dst, src, n, fKernelMode) \
00240 { \
00241     if (OS_needs_copy_from_user(fKernelMode)) \
00242         COPY_FROM_USER(dst, src, n); \
00243     else \
00244         memcpy(dst, src, n); \
00245 }
00246
00251 #define COPY_TO_USER_OR_KERNEL(dst, src, n, fKernelMode) \
00252 { \
00253     if (OS_needs_copy_from_user(fKernelMode)) \
00254         COPY_TO_USER(dst, src, n); \
00255     else \
00256         memcpy(dst, src, n); \
00257 }
00258
00259 #ifndef FALSE
00260     #define FALSE 0
00261 #endif
00262
00263 #ifndef TRUE
00264     #define TRUE 1
00265 #endif
00266
00267 #ifndef NULL
00268     #define NULL OUL
00269 #endif
00270
00271 int __cdecl KDBG(DWORD dwLevel, DWORD dwSection, const char *format, ...);
00272
00273 #if defined(WIN32)
00274     #if defined(KERNEL_64BIT)
00275         #include <stdarg.h>
00276     #else
00277         #define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
00278         // Define varargs ANSI style
00279         typedef char * va_list;
00280         #define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
00281         #define va_arg(ap,t) ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
00282         #define va_end(ap) ( ap = (va_list)0 )
00283
00284     #endif
00285
00286     int __cdecl _snprintf(char *buffer, unsigned long Limit, const char *format,
00287     ...);
00288     int __cdecl _vsnprintf(char *buffer, unsigned long Limit, const char
00289     *format, va_list Next);
00290 #endif
00291
00292 char* __cdecl strcpy(char *s1, const char *s2);
00293 void* __cdecl malloc(unsigned long size);
00294 void __cdecl free(void *buf);
00295
00296 #if defined(LINUX)
00297     #include <linux_wrappers.h>
00298     #define memset LINUX_memset
00299     #define strncmp LINUX_strncmp
00300     #define strcpy LINUX_strcpy
00301     #define strcmp LINUX_strcmp
00302     #define strncpy LINUX_strncpy
00303     #define strcat LINUX_strcat
00304     #define strncat LINUX_strncat
00305     #define strlen LINUX_strlen
00306     #define memcpy LINUX_memcpy
00307     #define memcmp LINUX_memcmp
00308     #define sprint LINUX_sprintf
00309     #define vsprintf LINUX_vsprintf
00310     #define snprintf LINUX_snprintf
00311     #define vsnprintf LINUX_vsnprintf
00312 #elif defined(WINNT)
00313     #if !defined(size_t)
00314         #if defined(KERNEL_64BIT)
00315             typedef unsigned __int64 size_t;
00316         #else
00317             typedef unsigned int size_t;
00318         #endif
00319     #endif
00320     void* __cdecl memcpy(void *dest, const void *src, size_t count);
00321     void* __cdecl memset(void *dest, int c, size_t count);
00322 #if !defined(_STRNCPY)
00323     char* _strncpy(char* s1, const char* s2, size_t limit);
00324 #endif
00325     #define sprintf _snprintf
```

```
00326     #define vsnprintf _vsnprintf
00327     #define strncpy _strncpy
00328 #endif
00329
00330 #ifdef __cplusplus
00331 }
00332#endif
00333
00334 #endif /* _KPSTDLIB_H_ */
00335
```

pci_regs.h File Reference

Macros

- `#define PCI_HEADER_TYPE 0x0e`
8 bits
- `#define PCI_HEADER_TYPE_NORMAL 0`
- `#define PCI_HEADER_TYPE_BRIDGE 1`
- `#define PCI_HEADER_TYPE_CARDBUS 2`
- `#define PCI_SR_CAP_LIST_BIT 0x00000010`
- `#define PCI_CAP_LIST_ID 0`
Capability ID.
- `#define PCI_CAP_ID_PM 0x01`
Power Management.
- `#define PCI_CAP_ID_AGP 0x02`
Accelerated Graphics Port.
- `#define PCI_CAP_ID_VPD 0x03`
Vital Product Data.
- `#define PCI_CAP_ID_SLOTID 0x04`
Slot Identification.
- `#define PCI_CAP_ID_MSI 0x05`
Message Signalled Interrupts.
- `#define PCI_CAP_ID_CHSWP 0x06`
CompactPCI HotSwap.
- `#define PCI_CAP_ID_PCIX 0x07`
PCI-X.
- `#define PCI_CAP_ID_HT 0x08`
HyperTransport.
- `#define PCI_CAP_ID_VNDR 0x09`
Vendor-Specific.
- `#define PCI_CAP_ID_DBG 0x0A`
Debug port.
- `#define PCI_CAP_ID_CCRC 0x0B`
CompactPCI Central Resource Control.
- `#define PCI_CAP_ID_SHPC 0x0C`
PCI Standard Hot-Plug Controller.
- `#define PCI_CAP_ID_SVID 0x0D`
Bridge subsystem vendor/device ID.
- `#define PCI_CAP_ID_AGP3 0x0E`
AGP Target PCI-PCI bridge.
- `#define PCI_CAP_ID_SECDEV 0x0F`
Secure Device.
- `#define PCI_CAP_ID_EXP 0x10`

- #define PCI_CAP_ID_MSIX 0x11
 - MSI-X.
- #define PCI_CAP_ID_SATA 0x12
 - SATA Data/Index Conf.
- #define PCI_CAP_ID_AF 0x13
 - PCI Advanced Features.
- #define PCI_CAP_LIST_NEXT 1
 - Next capability in the list.
- #define PCI_EXT_CAP_ID(header) (header & 0x0000ffff)
- #define PCI_EXT_CAP_VER(header) ((header >> 16) & 0xf)
- #define PCI_EXT_CAP_NEXT(header) ((header >> 20) & 0xffc)
- #define PCI_EXT_CAP_ID_ERR 0x0001
 - Advanced Error Reporting.
- #define PCI_EXT_CAP_ID_VC 0x0002
 - Virtual Channel Capability.
- #define PCI_EXT_CAP_ID_DSN 0x0003
 - Device Serial Number.
- #define PCI_EXT_CAP_ID_PWR 0x0004
 - Power Budgeting.
- #define PCI_EXT_CAP_ID_RCLD 0x0005
 - Root Complex Link Declaration.
- #define PCI_EXT_CAP_ID_RCILC 0x0006
 - Root Complex Internal Link Control.
- #define PCI_EXT_CAP_ID_RCEC 0x0007
 - Root Complex Event Collector.
- #define PCI_EXT_CAP_ID_MFVC 0x0008
 - Multi-Function VC Capability.
- #define PCI_EXT_CAP_ID_VC9 0x0009
 - same as _VC
- #define PCI_EXT_CAP_ID_RCRB 0x000A
 - Root Complex RB?
- #define PCI_EXT_CAP_ID_VNDR 0x000B
 - Vendor-Specific.
- #define PCI_EXT_CAP_ID_CAC 0x000C
 - Config Access - obsolete.
- #define PCI_EXT_CAP_ID_ACS 0x000D
 - Access Control Services.
- #define PCI_EXT_CAP_ID_ARI 0x000E
 - Alternate Routing ID.
- #define PCI_EXT_CAP_ID_ATS 0x000F
 - Address Translation Services.
- #define PCI_EXT_CAP_ID_SRIOV 0x0010
 - Single Root I/O Virtualization.
- #define PCI_EXT_CAP_ID_MRIOV 0x0011
 - Multi Root I/O Virtualization.
- #define PCI_EXT_CAP_ID_MCAST 0x0012
 - Multicast.
- #define PCI_EXT_CAP_ID_PRI 0x0013
 - Page Request Interface.
- #define PCI_EXT_CAP_ID_AMD_XXX 0x0014

- Reserved for AMD.*
- #define PCI_EXT_CAP_ID_REBAR 0x0015
Resizable BAR.
 - #define PCI_EXT_CAP_ID_DPA 0x0016
Dynamic Power Allocation.
 - #define PCI_EXT_CAP_ID_TPH 0x0017
TPH Requester.
 - #define PCI_EXT_CAP_ID_LTR 0x0018
Latency Tolerance Reporting.
 - #define PCI_EXT_CAP_ID_SECPCI 0x0019
Secondary PCIe Capability.
 - #define PCI_EXT_CAP_ID_PMUX 0x001A
Protocol Multiplexing.
 - #define PCI_EXT_CAP_ID_PASID 0x001B
Process Address Space ID.
 - #define PCI_EXT_CAP_ID_LNR 0x001C
LN Requester (LNR)
 - #define PCI_EXT_CAP_ID_DPC 0x001D
Downstream Port Containment (DPC)
 - #define PCI_EXT_CAP_ID_L1PMS 0x001E
L1 PM Substates.
 - #define PCI_EXT_CAP_ID_PTM 0x001F
Precision Time Measurement (PTM)
 - #define PCI_EXT_CAP_ID_MPHY 0x0020
PCI Express over M-PHY (M-PCIe)
 - #define PCI_EXT_CAP_ID_FRSQ 0x0021
FRS Queueing.
 - #define PCI_EXT_CAP_ID_RTR 0x0022
Readiness Time Reporting.
 - #define GET_CAPABILITY_STR(cap_id)
 - #define GET_EXTENDED_CAPABILITY_STR(cap_id)
 - #define PCI_EXP_DEVCAP_PHANTOM_SHIFT 3
 - #define PCI_STATUS_DEVSEL_SHIFT 9
 - #define PCI_EXP_DEVCTL_READRQ_SHIFT 12
 - #define PCI_EXP_SLTCAP_SPLV_SHIFT 7
 - #define PCI_EXP_FLAGS_TYPE_SHIFT 9
 - #define PCI_EXP_DEVCAP_L1_SHIFT 9
 - #define PCI_EXP_DEVCAP_PWD_SCL_SHIFT 26
 - #define PCI_EXP_DEVCAP_PWR_VAL_SHIFT 18
 - #define PCI_EXP_DEVCTL_PAYLOAD_SHIFT 5
 - #define PCI_EXP_LNKCAP_MLW_SHIFT 4
 - #define PCI_EXP_LNKCAP_ASPEMS_SHIFT 10
 - #define PCI_EXP_LNKCAP_L0SEL_SHIFT 12
 - #define PCI_EXP_LNKCAP_L1EL_SHIFT 15
 - #define PCI_EXP_SLTCAP_SPLS_SHIFT 15
 - #define PCI_EXP_SLTCAP_AIC_SHIFT 6
 - #define PCI_EXP_SLTCTL_PIC_SHIFT 8
 - #define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT 22
 - #define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT 22
 - #define PCI_COMMAND 0x04
16 bits
 - #define PCI_COMMAND_IO 0x1

- `#define PCI_COMMAND_MEMORY 0x2`
Enable response in I/O space.
- `#define PCI_COMMAND_MASTER 0x4`
Enable response in Memory space.
- `#define PCI_COMMAND_SPECIAL 0x8`
Enable bus mastering.
- `#define PCI_COMMAND_INVALIDATE 0x10`
Enable response to special cycles.
- `#define PCI_COMMAND_VGA_PALETTE 0x20`
Use memory write and invalidate.
- `#define PCI_COMMAND_PARITY 0x40`
Enable palette snooping.
- `#define PCI_COMMAND_WAIT 0x80`
Enable parity checking.
- `#define PCI_COMMAND_SERR 0x100`
Enable address/data stepping.
- `#define PCI_COMMAND_FAST_BACK 0x200`
Enable SERR.
- `#define PCI_COMMAND_INTX_DISABLE 0x400`
Enable back-to-back writes.
- `#define PCI_STATUS 0x06`
INTx Emulation Disable.
- `#define PCI_STATUS_INTERRUPT 0x08`
16 bits
- `#define PCI_STATUS_CAP_LIST 0x10`
Interrupt status.
- `#define PCI_STATUS_66MHZ 0x20`
Support Capability List.
- `#define PCI_STATUS_UDF 0x40`
Support 66 Mhz PCI 2.1 bus.
- `#define PCI_STATUS_FAST_BACK 0x80`
Support User Definable Features.
- `#define PCI_STATUS_PARITY 0x100`
Accept fast-back to back.
- `#define PCI_STATUS_DEVSEL_MASK 0x600`
Detected parity error.
- `#define PCI_STATUS_DEVSEL_FAST 0x000`
DEVSEL timing.
- `#define PCI_STATUS_DEVSEL_MEDIUM 0x200`
- `#define PCI_STATUS_DEVSEL_SLOW 0x400`
- `#define PCI_STATUS_SIG_TARGET_ABORT 0x800`
Set on target abort.
- `#define PCI_STATUS_REC_TARGET_ABORT 0x1000`
Master ack of abort.
- `#define PCI_STATUS_REC_MASTER_ABORT 0x2000`
Set on master abort.
- `#define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000`
Set when we drive SERR.
- `#define PCI_STATUS_DETECTED_PARITY 0x8000`
Set on parity error.
- `#define PCI_EXP_FLAGS 2`

- **Capabilities register.**
- #define **PCI_EXP_FLAGS_VERS** 0x000f
 - *Capability version.*
- #define **PCI_EXP_FLAGS_TYPE** 0x00f0
 - *Device/Port type.*
- #define **PCI_EXP_TYPE_ENDPOINT** 0x0
 - *Express Endpoint.*
- #define **PCI_EXP_TYPE_LEG_END** 0x1
 - *Legacy Endpoint.*
- #define **PCI_EXP_TYPE_ROOT_PORT** 0x4
 - *Root Port.*
- #define **PCI_EXP_TYPE_UPSTREAM** 0x5
 - *Upstream Port.*
- #define **PCI_EXP_TYPE_DOWNSTREAM** 0x6
 - *Downstream Port.*
- #define **PCI_EXP_TYPE_PCI_BRIDGE** 0x7
 - *PCIe to PCI/PCI-X Bridge.*
- #define **PCI_EXP_TYPE_PCIE_BRIDGE** 0x8
 - *PCI/PCI-X to PCIe Bridge.*
- #define **PCI_EXP_TYPE_RC_END** 0x9
 - *Root Complex Integrated Endpoint.*
- #define **PCI_EXP_TYPE_RC_EC** 0xa
 - *Root Complex Event Collector.*
- #define **PCI_EXP_FLAGS_SLOT** 0x0100
 - *Slot implemented.*
- #define **PCI_EXP_FLAGS_IRQ** 0x3e00
 - *Interrupt message number.*
- #define **PCI_EXP_DEVCAP** 4
 - *Device capabilities.*
- #define **PCI_EXP_DEVCAP_PAYLOAD** 0x00000007
 - *Max_Payload_Size.*
- #define **PCI_EXP_DEVCAP_PHANTOM** 0x00000018
 - *Phantom functions.*
- #define **PCI_EXP_DEVCAP_EXT_TAG** 0x00000020
 - *Extended tags.*
- #define **PCI_EXP_DEVCAP_L0S** 0x000001c0
 - *L0s Acceptable Latency.*
- #define **PCI_EXP_DEVCAP_L1** 0x00000e00
 - *L1 Acceptable Latency.*
- #define **PCI_EXP_DEVCAP_ATN_BUT** 0x00001000
 - *Attention Button Present.*
- #define **PCI_EXP_DEVCAP_ATN_IND** 0x00002000
 - *Attention Indicator Present.*
- #define **PCI_EXP_DEVCAP_PWR_IND** 0x00004000
 - *Power Indicator Present.*
- #define **PCI_EXP_DEVCAP_RBER** 0x00008000
 - *Role-Based Error Reporting.*
- #define **PCI_EXP_DEVCAP_PWR_VAL** 0x03fc0000
 - *Slot Power Limit Value.*
- #define **PCI_EXP_DEVCAP_PWR_SCL** 0x0c000000
 - *Slot Power Limit Scale.*

- #define PCI_EXP_DEVCTL_FLR 0x10000000
 - Function Level Reset.*
- #define PCI_EXP_DEVCTL 8
 - Device Control.*
- #define PCI_EXP_DEVCTL_CERE 0x0001
 - Correctable Error Reporting En.*
- #define PCI_EXP_DEVCTL_NFERE 0x0002
 - Non-Fatal Error Reporting Enable.*
- #define PCI_EXP_DEVCTL_FERE 0x0004
 - Fatal Error Reporting Enable.*
- #define PCI_EXP_DEVCTL_URRE 0x0008
 - Unsupported Request Reporting En.*
- #define PCI_EXP_DEVCTL_RELAX_EN 0x0010
 - Enable relaxed ordering.*
- #define PCI_EXP_DEVCTL_PAYLOAD 0x00e0
 - Max_Payload_Size.*
- #define PCI_EXP_DEVCTL_EXT_TAG 0x0100
 - Extended Tag Field Enable.*
- #define PCI_EXP_DEVCTL_PHANTOM 0x0200
 - Phantom Functions Enable.*
- #define PCI_EXP_DEVCTL_AUX_PME 0x0400
 - Auxiliary Power PM Enable.*
- #define PCI_EXP_DEVCTL_NOSNOOP_EN 0x0800
 - Enable No Snoop.*
- #define PCI_EXP_DEVCTL_READRQ 0x7000
 - Max_Read_Request_Size.*
- #define PCI_EXP_DEVCTL_READRQ_128B 0x0000
 - 128 Bytes*
- #define PCI_EXP_DEVCTL_READRQ_256B 0x1000
 - 256 Bytes*
- #define PCI_EXP_DEVCTL_READRQ_512B 0x2000
 - 512 Bytes*
- #define PCI_EXP_DEVCTL_READRQ_1024B 0x3000
 - 1024 Bytes*
- #define PCI_EXP_DEVCTL_BCR_FLR 0x8000
 - Bridge Configuration Retry / FLR.*
- #define PCI_EXP_DEVSTA_10
 - Device Status.*
- #define PCI_EXP_DEVSTA_CED 0x0001
 - Correctable Error Detected.*
- #define PCI_EXP_DEVSTA_NFED 0x0002
 - Non-Fatal Error Detected.*
- #define PCI_EXP_DEVSTA_FED 0x0004
 - Fatal Error Detected.*
- #define PCI_EXP_DEVSTA_URD 0x0008
 - Unsupported Request Detected.*
- #define PCI_EXP_DEVSTA_AUXPD 0x0010
 - AUX Power Detected.*
- #define PCI_EXP_DEVSTA_TRPND 0x0020
 - Transactions Pending.*
- #define PCI_EXP_LNKCAP 12

- **#define PCI_EXP_LNKCAP_SLS** 0x0000000f
 - Supported Link Speeds.*
- **#define PCI_EXP_LNKCAP_SLS_2_5GB** 0x00000001
 - LNKCAP2 SLS Vector bit 0.*
- **#define PCI_EXP_LNKCAP_SLS_5_0GB** 0x00000002
 - LNKCAP2 SLS Vector bit 1.*
- **#define PCI_EXP_LNKCAP_MLW** 0x000003f0
 - Maximum Link Width.*
- **#define PCI_EXP_LNKCAP_ASPMS** 0x00000c00
 - ASPM Support.*
- **#define PCI_EXP_LNKCAP_L0SEL** 0x00007000
 - L0s Exit Latency.*
- **#define PCI_EXP_LNKCAP_L1EL** 0x00038000
 - L1 Exit Latency.*
- **#define PCI_EXP_LNKCAP_CLKPM** 0x00040000
 - Clock Power Management.*
- **#define PCI_EXP_LNKCAP_SDERC** 0x00080000
 - Surprise Down Error Reporting Capable.*
- **#define PCI_EXP_LNKCAP_DLLLARC** 0x00100000
 - Data Link Layer Link Active Reporting Capable.*
- **#define PCI_EXP_LNKCAP_LBNC** 0x00200000
 - Link Bandwidth Notification Capability.*
- **#define PCI_EXP_LNKCAP_PN** 0xff000000
 - Port Number.*
- **#define PCI_EXP_LNKCTL** 16
 - Link Control.*
- **#define PCI_EXP_LNKCTL_ASPMC** 0x0003
 - ASPM Control.*
- **#define PCI_EXP_LNKCTL_ASPM_L0S** 0x0001
 - L0s Enable.*
- **#define PCI_EXP_LNKCTL_ASPM_L1** 0x0002
 - L1 Enable.*
- **#define PCI_EXP_LNKCTL_RCB** 0x0008
 - Read Completion Boundary.*
- **#define PCI_EXP_LNKCTL_LD** 0x0010
 - Link Disable.*
- **#define PCI_EXP_LNKCTL_RL** 0x0020
 - Retrain Link.*
- **#define PCI_EXP_LNKCTL_CCC** 0x0040
 - Common Clock Configuration.*
- **#define PCI_EXP_LNKCTL_ES** 0x0080
 - Extended Synch.*
- **#define PCI_EXP_LNKCTL_CLKREQ_EN** 0x0100
 - Enable clkreq.*
- **#define PCI_EXP_LNKCTL_HAWD** 0x0200
 - Hardware Autonomous Width Disable.*
- **#define PCI_EXP_LNKCTL_LBMIE** 0x0400
 - Link Bandwidth Management Interrupt Enable.*
- **#define PCI_EXP_LNKCTL_LABIE** 0x0800
 - Link Autonomous Bandwidth Interrupt Enable.*

- #define PCI_EXP_LNKSTA 18
Link Status.
- #define PCI_EXP_LNKSTA_CLS 0x000f
Current Link Speed.
- #define PCI_EXP_LNKSTA_CLS_2_5GB 0x0001
Current Link Speed 2.5GT/s.
- #define PCI_EXP_LNKSTA_CLS_5_0GB 0x0002
Current Link Speed 5.0GT/s.
- #define PCI_EXP_LNKSTA_CLS_8_0GB 0x0003
Current Link Speed 8.0GT/s.
- #define PCI_EXP_LNKSTA_NLW 0x03f0
Negotiated Link Width.
- #define PCI_EXP_LNKSTA_NLW_X1 0x0010
Current Link Width x1.
- #define PCI_EXP_LNKSTA_NLW_X2 0x0020
Current Link Width x2.
- #define PCI_EXP_LNKSTA_NLW_X4 0x0040
Current Link Width x4.
- #define PCI_EXP_LNKSTA_NLW_X8 0x0080
Current Link Width x8.
- #define PCI_EXP_LNKSTA_NLW_SHIFT 4
start of NLW mask in link status
- #define PCI_EXP_LNKSTA_LT 0x0800
Link Training.
- #define PCI_EXP_LNKSTA_SLC 0x1000
Slot Clock Configuration.
- #define PCI_EXP_LNKSTA_DLLLA 0x2000
Data Link Layer Link Active.
- #define PCI_EXP_LNKSTA_LBMS 0x4000
Link Bandwidth Management Status.
- #define PCI_EXP_LNKSTA_LABS 0x8000
Link Autonomous Bandwidth Status.
- #define PCI_CAP_EXP_ENDPOINT_SIZEOF_V1 20
v1 endpoints end here
- #define PCI_EXP_SLTCAP 20
Slot Capabilities.
- #define PCI_EXP_SLTCAP_ABPP 0x00000001
Attention Button Present.
- #define PCI_EXP_SLTCAP_PCP 0x00000002
Power Controller Present.
- #define PCI_EXP_SLTCAP_MRLSP 0x00000004
MRL Sensor Present.
- #define PCI_EXP_SLTCAP_AIP 0x00000008
Attention Indicator Present.
- #define PCI_EXP_SLTCAP_PIP 0x00000010
Power Indicator Present.
- #define PCI_EXP_SLTCAP_HPS 0x00000020
Hot-Plug Surprise.
- #define PCI_EXP_SLTCAP_HPC 0x00000040
Hot-Plug Capable.
- #define PCI_EXP_SLTCAP_SPLV 0x00007f80

- #define PCI_EXP_SLTCAP_SPLS 0x00018000
 - Slot Power Limit Value.*
- #define PCI_EXP_SLTCAP_EIP 0x00020000
 - Slot Power Limit Scale.*
- #define PCI_EXP_SLTCAP_NCCS 0x00040000
 - No Command Completed Support.*
- #define PCI_EXP_SLTCAP_PSN 0xffff80000
 - Physical Slot Number.*
- #define PCI_EXP_SLTCTL 24
 - Slot Control.*
- #define PCI_EXP_SLTCTL_ABPE 0x0001
 - Attention Button Pressed Enable.*
- #define PCI_EXP_SLTCTL_PFDE 0x0002
 - Power Fault Detected Enable.*
- #define PCI_EXP_SLTCTL_MRLSCE 0x0004
 - MRL Sensor Changed Enable.*
- #define PCI_EXP_SLTCTL_PDCE 0x0008
 - Presence Detect Changed Enable.*
- #define PCI_EXP_SLTCTL_CCIE 0x0010
 - Command Completed Interrupt Enable.*
- #define PCI_EXP_SLTCTL_HPIE 0x0020
 - Hot-Plug Interrupt Enable.*
- #define PCI_EXP_SLTCTL_AIC 0x00c0
 - Attention Indicator Control.*
- #define PCI_EXP_SLTCTL_ATTN_IND_ON 0x0040
 - Attention Indicator on.*
- #define PCI_EXP_SLTCTL_ATTN_IND_BLINK 0x0080
 - Attention Indicator blinking.*
- #define PCI_EXP_SLTCTL_ATTN_IND_OFF 0x00c0
 - Attention Indicator off.*
- #define PCI_EXP_SLTCTL_PIC 0x0300
 - Power Indicator Control.*
- #define PCI_EXP_SLTCTL_PWR_IND_ON 0x0100
 - Power Indicator on.*
- #define PCI_EXP_SLTCTL_PWR_IND_BLINK 0x0200
 - Power Indicator blinking.*
- #define PCI_EXP_SLTCTL_PWR_IND_OFF 0x0300
 - Power Indicator off.*
- #define PCI_EXP_SLTCTL_PCC 0x0400
 - Power Controller Control.*
- #define PCI_EXP_SLTCTL_PWR_ON 0x0000
 - Power On.*
- #define PCI_EXP_SLTCTL_PWR_OFF 0x0400
 - Power Off.*
- #define PCI_EXP_SLTCTL_EIC 0x0800
 - Electromechanical Interlock Control.*
- #define PCI_EXP_SLTCTL_DLLSCE 0x1000
 - Data Link Layer State Changed Enable.*
- #define PCI_EXP_SLTSTA 26
 - Slot Status.*

- #define PCI_EXP_SLTSTA_AB_P 0x0001
Attention Button Pressed.
- #define PCI_EXP_SLTSTA_PFD 0x0002
Power Fault Detected.
- #define PCI_EXP_SLTSTA_MRLSC 0x0004
MRL Sensor Changed.
- #define PCI_EXP_SLTSTA_PDC 0x0008
Presence Detect Changed.
- #define PCI_EXP_SLTSTA_CC 0x0010
Command Completed.
- #define PCI_EXP_SLTSTA_MRLSS 0x0020
MRL Sensor State.
- #define PCI_EXP_SLTSTA_PDS 0x0040
Presence Detect State.
- #define PCI_EXP_SLTSTA_EIS 0x0080
Electromechanical Interlock Status.
- #define PCI_EXP_SLTSTA_DLLSC 0x0100
Data Link Layer State Changed.
- #define PCI_EXP_RTCTL 28
Root Control.
- #define PCI_EXP_RTCTL_SECEE 0x0001
System Error on Correctable Error.
- #define PCI_EXP_RTCTL_SEFEE 0x0002
System Error on Non-Fatal Error.
- #define PCI_EXP_RTCTL_SEFEE 0x0004
System Error on Fatal Error.
- #define PCI_EXP_RTCTL_PMEIE 0x0008
PME Interrupt Enable.
- #define PCI_EXP_RTCTL_CRSSVE 0x0010
CRS Software Visibility Enable.
- #define PCI_EXP_RTCAP 30
Root Capabilities.
- #define PCI_EXP_RTCAP_CRSVIS 0x0001
CRS Software Visibility capability.
- #define PCI_EXP_RTSTA 32
Root Status.
- #define PCI_EXP_RTSTA_PME 0x00010000
PME status.
- #define PCI_EXP_RTSTA_PENDING 0x00020000
PME pending.
- #define PCI_EXP_DEVCAP2 36
Device Capabilities 2.
- #define PCI_EXP_DEVCAP2_RANGE_A 0x1
Completion Timeout Range A.
- #define PCI_EXP_DEVCAP2_RANGE_B 0x2
Completion Timeout Range B.
- #define PCI_EXP_DEVCAP2_RANGE_C 0x4
Completion Timeout Range C.
- #define PCI_EXP_DEVCAP2_RANGE_D 0x8
Completion Timeout Range D.
- #define PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP 0xF

- `#define PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP` 0x0000010
 - Completion Timeout Ranges Supported.*
- `#define PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP` 0x0000010
 - Completion Timeout Disable Supported.*
- `#define PCI_EXP_DEVCAP2_ARI` 0x00000020
 - Alternative Routing-ID.*
- `#define PCI_EXP_DEVCAP2_ATOMIC_ROUTE` 0x00000040
 - Atomic Op routing.*
- `#define PCI_EXP_DEVCAP2_ATOMIC_COMP32` 0x000080
 - 32-bit AtomicOp Completer Supported*
- `#define PCI_EXP_DEVCAP2_ATOMIC_COMP64` 0x00000100
 - Atomic 64-bit compare.*
- `#define PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP` 0x000200
 - 128-bit CAS Completer Supported*
- `#define PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR` 0x000400
 - No RO Enabled PR-PR Passing.*
- `#define PCI_EXP_DEVCAP2_LTR` 0x00000800
 - Latency tolerance reporting.*
- `#define PCI_EXP_DEVCAP2_TPH_COMP_SUPP` 0x001000
 - TPH Completer Supported.*
- `#define PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP` 0x002000
 - Extended TPH Completer Supported.*
- `#define PCI_EXP_DEVCAP2_OBFF_MASK` 0x000c0000
 - OBFF support mechanism.*
- `#define PCI_EXP_DEVCAP2_OBFF_MSG` 0x00040000
 - New message signaling.*
- `#define PCI_EXP_DEVCAP2_OBFF_WAKE` 0x00080000
 - Re-use WAKE# for OBFF.*
- `#define PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP` 0x100000
 - Extended Fmt Field Supported.*
- `#define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP` 0x200000
 - End-End TLP Prefix Supported.*
- `#define PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES` 0xC00000
 - Max End-End TLP Prefixes.*
- `#define PCI_EXP_DEVCTL2` 40
 - Device Control 2.*
- `#define PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE` 0x0010
 - End-End TLP Prefix Blocking.*
- `#define PCI_EXP_DEVCTL2_COMP_TIMEOUT` 0x000f
 - Completion Timeout Value.*
- `#define PCI_EXP_DEVCTL2_ARI` 0x0020
 - Alternative Routing-ID.*
- `#define PCI_EXP_DEVCTL2_ATOMIC_REQ` 0x0040
 - Set Atomic requests.*
- `#define PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK` 0x0080
 - Block atomic egress.*
- `#define PCI_EXP_DEVCTL2_IDO_REQ_EN` 0x0100
 - Allow IDO for requests.*
- `#define PCI_EXP_DEVCTL2_IDO_CMP_EN` 0x0200
 - Allow IDO for completions.*
- `#define PCI_EXP_DEVCTL2_LTR_EN` 0x0400
 - Enable LTR mechanism.*

- #define PCI_EXP_DEVCTL2_OBFF_MSGA_EN 0x2000
 - Enable OBFF Message type A.*
- #define PCI_EXP_DEVCTL2_OBFF_MSGB_EN 0x4000
 - Enable OBFF Message type B.*
- #define PCI_EXP_DEVCTL2_OBFF_WAKE_EN 0x6000
 - OBFF using WAKE# signaling.*
- #define PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK 0x8000
 - End-End TLP Prefix Blocking.*
- #define PCI_EXP_DEVSTA2 42
 - Device Status 2.*
- #define PCI_CAP_EXP_ENDPOINT_SIZEOF_V2 44
 - v2 endpoints end here*
- #define PCI_EXP_LNKCAP2 44
 - Link Capabilities 2.*
- #define PCI_EXP_LNKCAP2_SLS_2_5GB 0x00000002
 - Supported Speed 2.5GT/s.*
- #define PCI_EXP_LNKCAP2_SLS_5_0GB 0x00000004
 - Supported Speed 5.0GT/s.*
- #define PCI_EXP_LNKCAP2_SLS_8_0GB 0x00000008
 - Supported Speed 8.0GT/s.*
- #define PCI_EXP_LNKCAP2_CROSSLINK 0x00000100
 - Crosslink supported.*
- #define PCI_EXP_LNKCTL2 48
 - Link Control 2.*
- #define PCI_EXP_LNKCTL2_LNK_SPEED_2_5 0x00000001
 - Link Speed 2.5 GT/s.*
- #define PCI_EXP_LNKCTL2_LNK_SPEED_5_0 0x00000002
 - Link Speed 5 GT/s.*
- #define PCI_EXP_LNKCTL2_LNK_SPEED_8_0 0x00000003
 - Link Speed 8 GT/s.*
- #define PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK 0x0000000f
 - Target Lnk Speed Mask.*
- #define PCI_EXP_LNKCTL2_ENTER_COMP 0x00000010
 - Enter Compliance.*
- #define PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS 0x00000020
 - Hardware Autonomous Speed Disable.*
- #define PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH 0x00000040
 - Selectable De-emphasis.*
- #define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK 0x00000380
 - Transmit Margin.*
- #define PCI_EXP_LNKCTL2_ENTER_MOD_COMP 0x00000400
 - Enter Modified Compliance.*
- #define PCI_EXP_LNKCTL2_COMP_SOS 0x00000800
 - Compliance SOS.*
- #define PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL 0x00001000
 - De-emphasis level polling.*
- #define PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL 0x0000f000
 - Transmitter Preset in polling.*
- #define PCI_EXP_LNKSTA2 50/** Link Status 2 */
 - Current De-emphasis level (at 5 GT/s speed only)*

- #define PCI_EXP_LNKSTA2_EQUALIZ_COMP 0x00000002
Equalization Complete.
- #define PCI_EXP_LNKSTA2_EQUALIZ_PH1 0x00000004
Equalization Ph.1 Successful.
- #define PCI_EXP_LNKSTA2_EQUALIZ_PH2 0x00000008
Equalization Ph.2 Successful.
- #define PCI_EXP_LNKSTA2_EQUALIZ_PH3 0x00000010
Equalization Ph.3 Successful.
- #define PCI_EXP_LNKSTA2_LINE_EQ_REQ 0x00000020
Link Equalization Request.
- #define PCI_EXP_SLTCAP2 52
Slot Capabilities 2.
- #define PCI_EXP_SLTCTL2 56
Slot Control 2.
- #define PCI_EXP_SLTSTA2 58
Slot Status 2.

Enumerations

- enum WDC_PCI_HEADER_TYPE {
 HEADER_TYPE_NORMAL = 0x01 , HEADER_TYPE_BRIDGE = 0x02 , HEADER_TYPE_CARDBUS = 0x04 , HEADER_TYPE_NRML_BRIDGE = HEADER_TYPE_NORMAL | HEADER_TYPE_BRIDGE , HEADER_TYPE_NRML_CARDBUS = HEADER_TYPE_NORMAL | HEADER_TYPE_CARDBUS , HEADER_TYPE_BRIDGE_CARDBUS = HEADER_TYPE_BRIDGE | HEADER_TYPE_CARDBUS , HEADER_TYPE_ALL }
 - enum PCI_CONFIG_REGS_OFFSET {
 PCI_VID = 0x00 , PCI_DID = 0x02 , PCI_CR = 0x04 , PCI_SR = 0x06 , PCI_REV = 0x08 , PCI_CCR = 0x09 , PCI_CCSC = 0x0a , PCI_CCBC = 0x0b , PCI_CLSR = 0x0c , PCI_LTR = 0x0d , PCI_HDR = 0x0e , PCI_BISTR = 0x0f , PCI_BAR0 = 0x10 , PCI_BAR1 = 0x14 , PCI_BAR2 = 0x18 , PCI_BAR3 = 0x1c , PCI_BAR4 = 0x20 , PCI_BAR5 = 0x24 , PCI_CIS = 0x28 , PCI_SVID = 0x2c , PCI_SDID = 0x2e , PCI_EROM = 0x30 , PCI_CAP = 0x34 , PCI_ILR = 0x3c , PCI_IPR = 0x3d , PCI_MGR = 0x3e , PCI_MLR = 0x3f }
 - enum PCIE_CONFIG_REGS_OFFSET {
 PCIE_CAP_ID = 0x0 , NEXT_CAP_PTR = 0x1 , CAP_REG = 0x2 , DEV_CAPS = 0x4 , DEV_CTL = 0x8 , DEV_STS = 0xa , LNK_CAPS = 0xc , LNK_CTL = 0x10 , LNK_STS = 0x12 , SLOT_CAPS = 0x14 , SLOT_CTL = 0x18 , SLOT_STS = 0x1a , ROOT_CAPS = 0x1c , ROOT_CTL = 0x1e , ROOT_STS = 0x20 , DEV_CAPS2 = 0x24 , DEV_CTL2 = 0x28 , DEV_STS2 = 0x2a , LNK_CAPS2 = 0x2c , LNK_CTL2 = 0x30 , LNK_STS2 = 0x32 , SLOT_CAPS2 = 0x34 , SLOT_CTL2 = 0x38 , SLOT_STS2 = 0x3a }
 - enum AD_PCI_BAR {
 AD_PCI_BAR0 = 0 , AD_PCI_BAR1 = 1 , AD_PCI_BAR2 = 2 , AD_PCI_BAR3 = 3 , AD_PCI_BAR4 = 4 , AD_PCI_BAR5 = 5 , AD_PCI_BARS = 6 }
- PCI base address spaces (BARs)*

Macro Definition Documentation

GET_CAPABILITY_STR

```
#define GET_CAPABILITY_STR(
    cap_id )
```

Value:

```
(cap_id) == 0x00 ? "Null Capability" : \
(cap_id) == PCI_CAP_ID_PM ? "Power Management" : \
(cap_id) == PCI_CAP_ID_AGP ? "Accelerated Graphics Port" : \
(cap_id) == PCI_CAP_ID_VPD ? "Vital Product Data" : \
(cap_id) == PCI_CAP_ID_SLOTID ? "Slot Identification" : \
(cap_id) == PCI_CAP_ID_MSI ? "Message Signalled Interrupts (MSI)" : \
(cap_id) == PCI_CAP_ID_CHSWP ? "CompactPCI HotSwap" : \
(cap_id) == PCI_CAP_ID_PCIX ? "PCI-X" : \
(cap_id) == PCI_CAP_ID_HT ? "HyperTransport" : \
(cap_id) == PCI_CAP_ID_VNDR ? "Vendor-Specific" : \
(cap_id) == PCI_CAP_ID_DBG ? "Debug port" : \
(cap_id) == PCI_CAP_ID_CCRC ? "CompactPCI Central Resource Control" : \
(cap_id) == PCI_CAP_ID_SHPC ? "PCI Standard Hot-Plug Controller" : \
(cap_id) == PCI_CAP_ID_SSVID ? "Bridge subsystem vendor/device ID" : \
(cap_id) == PCI_CAP_ID_AGP3 ? "AGP Target PCI-PCI bridge" : \
(cap_id) == PCI_CAP_ID_SECDEV ? "Secure Device" : \
(cap_id) == PCI_CAP_ID_EXP ? "PCI Express" : \
(cap_id) == PCI_CAP_ID_MSIX ? "Extended Message Signalled Interrupts (MSI-X)" : \
(cap_id) == PCI_CAP_ID_SATA ? "SATA Data/Index Conf." : \
(cap_id) == PCI_CAP_ID_AF ? "PCI Advanced Features" : \
"Unknown"
```

Definition at line 173 of file [pci_regs.h](#).

GET_EXTENDED_CAPABILITY_STR

```
#define GET_EXTENDED_CAPABILITY_STR(
    cap_id )
```

Definition at line 196 of file [pci_regs.h](#).

PCI_CAP_EXP_ENDPOINT_SIZEOF_V1

```
#define PCI_CAP_EXP_ENDPOINT_SIZEOF_V1 20
v1 endpoints end here
```

Definition at line 385 of file [pci_regs.h](#).

PCI_CAP_EXP_ENDPOINT_SIZEOF_V2

```
#define PCI_CAP_EXP_ENDPOINT_SIZEOF_V2 44
v2 endpoints end here
```

Definition at line 491 of file [pci_regs.h](#).

PCI_CAP_ID_AF

```
#define PCI_CAP_ID_AF 0x13
```

PCI Advanced Features.

Definition at line 130 of file [pci_regs.h](#).

PCI_CAP_ID_AGP

```
#define PCI_CAP_ID_AGP 0x02
```

Accelerated Graphics Port.

Definition at line 113 of file [pci_regs.h](#).

PCI_CAP_ID_AGP3

#define PCI_CAP_ID_AGP3 0x0E

AGP Target PCI-PCI bridge.

Definition at line 125 of file [pci_regs.h](#).

PCI_CAP_ID_CCRC

#define PCI_CAP_ID_CCRC 0x0B

CompactPCI Central Resource Control.

Definition at line 122 of file [pci_regs.h](#).

PCI_CAP_ID_CHSWP

#define PCI_CAP_ID_CHSWP 0x06

CompactPCI HotSwap.

Definition at line 117 of file [pci_regs.h](#).

PCI_CAP_ID_DBG

#define PCI_CAP_ID_DBG 0x0A

Debug port.

Definition at line 121 of file [pci_regs.h](#).

PCI_CAP_ID_EXP

#define PCI_CAP_ID_EXP 0x10

PCI Express.

Definition at line 127 of file [pci_regs.h](#).

PCI_CAP_ID_HT

#define PCI_CAP_ID_HT 0x08

HyperTransport.

Definition at line 119 of file [pci_regs.h](#).

PCI_CAP_ID_MSI

#define PCI_CAP_ID_MSI 0x05

Message Signalled Interrupts.

Definition at line 116 of file [pci_regs.h](#).

PCI_CAP_ID_MSIX

```
#define PCI_CAP_ID_MSIX 0x11
```

MSI-X.

Definition at line 128 of file [pci_regs.h](#).

PCI_CAP_ID_PCIX

```
#define PCI_CAP_ID_PCIX 0x07
```

PCI-X.

Definition at line 118 of file [pci_regs.h](#).

PCI_CAP_ID_PM

```
#define PCI_CAP_ID_PM 0x01
```

Power Management.

Definition at line 112 of file [pci_regs.h](#).

PCI_CAP_ID_SATA

```
#define PCI_CAP_ID_SATA 0x12
```

SATA Data/Index Conf.

Definition at line 129 of file [pci_regs.h](#).

PCI_CAP_ID_SECDEV

```
#define PCI_CAP_ID_SECDEV 0x0F
```

Secure Device.

Definition at line 126 of file [pci_regs.h](#).

PCI_CAP_ID_SHPC

```
#define PCI_CAP_ID_SHPC 0x0C
```

PCI Standard Hot-Plug Controller.

Definition at line 123 of file [pci_regs.h](#).

PCI_CAP_ID_SLOTID

```
#define PCI_CAP_ID_SLOTID 0x04
```

Slot Identification.

Definition at line 115 of file [pci_regs.h](#).

PCI_CAP_ID_SVID

```
#define PCI_CAP_ID_SVID 0x0D
```

Bridge subsystem vendor/device ID.

Definition at line 124 of file [pci_regs.h](#).

PCI_CAP_ID_VNDR

```
#define PCI_CAP_ID_VNDR 0x09
```

Vendor-Specific.

Definition at line 120 of file [pci_regs.h](#).

PCI_CAP_ID_VPD

```
#define PCI_CAP_ID_VPD 0x03
```

Vital Product Data.

Definition at line 114 of file [pci_regs.h](#).

PCI_CAP_LIST_ID

```
#define PCI_CAP_LIST_ID 0
```

Capability ID.

Definition at line 111 of file [pci_regs.h](#).

PCI_CAP_LIST_NEXT

```
#define PCI_CAP_LIST_NEXT 1
```

Next capability in the list.

Definition at line 131 of file [pci_regs.h](#).

PCI_COMMAND

```
#define PCI_COMMAND 0x04
```

16 bits

Definition at line 253 of file [pci_regs.h](#).

PCI_COMMAND_FAST_BACK

```
#define PCI_COMMAND_FAST_BACK 0x200
```

Enable back-to-back writes.

Definition at line [263](#) of file [pci_regs.h](#).

PCI_COMMAND_INTX_DISABLE

```
#define PCI_COMMAND_INTX_DISABLE 0x400
```

INTx Emulation Disable.

Definition at line [264](#) of file [pci_regs.h](#).

PCI_COMMAND_INVALIDATE

```
#define PCI_COMMAND_INVALIDATE 0x10
```

Use memory write and invalidate.

Definition at line [258](#) of file [pci_regs.h](#).

PCI_COMMAND_IO

```
#define PCI_COMMAND_IO 0x1
```

Enable response in I/O space.

Definition at line [254](#) of file [pci_regs.h](#).

PCI_COMMAND_MASTER

```
#define PCI_COMMAND_MASTER 0x4
```

Enable bus mastering.

Definition at line [256](#) of file [pci_regs.h](#).

PCI_COMMAND_MEMORY

```
#define PCI_COMMAND_MEMORY 0x2
```

Enable response in Memory space.

Definition at line [255](#) of file [pci_regs.h](#).

PCI_COMMAND_PARITY

```
#define PCI_COMMAND_PARITY 0x40
```

Enable parity checking.

Definition at line [260](#) of file [pci_regs.h](#).

PCI_COMMAND_SERR

```
#define PCI_COMMAND_SERR 0x100
```

Enable SERR.

Definition at line [262](#) of file [pci_regs.h](#).

PCI_COMMAND_SPECIAL

```
#define PCI_COMMAND_SPECIAL 0x8
```

Enable response to special cycles.

Definition at line [257](#) of file [pci_regs.h](#).

PCI_COMMAND_VGA_PALETTE

```
#define PCI_COMMAND_VGA_PALETTE 0x20
```

Enable palette snooping.

Definition at line [259](#) of file [pci_regs.h](#).

PCI_COMMAND_WAIT

```
#define PCI_COMMAND_WAIT 0x80
```

Enable address/data stepping.

Definition at line [261](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP

```
#define PCI_EXP_DEVCAP 4
```

Device capabilities.

Definition at line [301](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2

```
#define PCI_EXP_DEVCAP2 36
```

Device Capabilities 2.

Definition at line [442](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP

```
#define PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP 0x000200
```

128-bit CAS Completer Supported

Definition at line [457](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_ARI

```
#define PCI_EXP_DEVCAP2_ARI 0x00000020
```

Alternative Routing-ID.

Definition at line 451 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_ATOMIC_COMP32

```
#define PCI_EXP_DEVCAP2_ATOMIC_COMP32 0x0000080
```

32-bit AtomicOp Completer Supported

Definition at line 454 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_ATOMIC_COMP64

```
#define PCI_EXP_DEVCAP2_ATOMIC_COMP64 0x00000100
```

Atomic 64-bit compare.

Definition at line 455 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_ATOMIC_ROUTE

```
#define PCI_EXP_DEVCAP2_ATOMIC_ROUTE 0x00000040
```

Atomic Op routing.

Definition at line 452 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP

```
#define PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP 0x0000010
```

Completion Timeout Disable Supported.

Definition at line 450 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP

```
#define PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP 0xF
```

Completion Timeout Ranges Supported.

Definition at line 448 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP

```
#define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP 0x200000
```

End-End TLP Prefix Supported.

Definition at line 471 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT

```
#define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT 22
```

Definition at line [250](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP

```
#define PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP 0x100000
```

Extended Fmt Field Supported.

Definition at line [469](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP

```
#define PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP 0x002000
```

Extended TPH Completer Supported.

Definition at line [464](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_LTR

```
#define PCI_EXP_DEVCAP2_LTR 0x00000800
```

Latency tolerance reporting.

Definition at line [460](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES

```
#define PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES 0xC00000
```

Max End-End TLP Prefixes.

Definition at line [473](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR

```
#define PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR 0x000400
```

No RO Enabled PR-PR Passing.

Definition at line [459](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_OBFF_MASK

```
#define PCI_EXP_DEVCAP2_OBFF_MASK 0x000c0000
```

OBFF support mechanism.

Definition at line [465](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_OBFF_MSG

```
#define PCI_EXP_DEVCAP2_OBFF_MSG 0x00040000
```

New message signaling.

Definition at line 466 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_OBFF_WAKE

```
#define PCI_EXP_DEVCAP2_OBFF_WAKE 0x00080000
```

Re-use WAKE# for OBFF.

Definition at line 467 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_RANGE_A

```
#define PCI_EXP_DEVCAP2_RANGE_A 0x1
```

Completion Timeout Range A.

Definition at line 443 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_RANGE_B

```
#define PCI_EXP_DEVCAP2_RANGE_B 0x2
```

Completion Timeout Range B.

Definition at line 444 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_RANGE_C

```
#define PCI_EXP_DEVCAP2_RANGE_C 0x4
```

Completion Timeout Range C.

Definition at line 445 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_RANGE_D

```
#define PCI_EXP_DEVCAP2_RANGE_D 0x8
```

Completion Timeout Range D.

Definition at line 446 of file [pci_regs.h](#).

PCI_EXP_DEVCAP2_TPH_COMP_SUPP

```
#define PCI_EXP_DEVCAP2_TPH_COMP_SUPP 0x001000
```

TPH Completer Supported.

Definition at line 462 of file [pci_regs.h](#).

PCI_EXP_DEVCAP_ATN_BUT

```
#define PCI_EXP_DEVCAP_ATN_BUT 0x000001000
```

Attention Button Present.

Definition at line [307](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_ATN_IND

```
#define PCI_EXP_DEVCAP_ATN_IND 0x000002000
```

Attention Indicator Present.

Definition at line [308](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_EXT_TAG

```
#define PCI_EXP_DEVCAP_EXT_TAG 0x000000020
```

Extended tags.

Definition at line [304](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_FLR

```
#define PCI_EXP_DEVCAP_FLR 0x100000000
```

Function Level Reset.

Definition at line [313](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_L0S

```
#define PCI_EXP_DEVCAP_L0S 0x0000001c0
```

L0s Acceptable Latency.

Definition at line [305](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_L1

```
#define PCI_EXP_DEVCAP_L1 0x00000e00
```

L1 Acceptable Latency.

Definition at line [306](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_L1_SHIFT

```
#define PCI_EXP_DEVCAP_L1_SHIFT 9
```

Definition at line [239](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PAYLOAD

```
#define PCI_EXP_DEVCAP_PAYLOAD 0x00000007
```

Max_Payload_Size.

Definition at line [302](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PHANTOM

```
#define PCI_EXP_DEVCAP_PHANTOM 0x00000018
```

Phantom functions.

Definition at line [303](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PHANTOM_SHIFT

```
#define PCI_EXP_DEVCAP_PHANTOM_SHIFT 3
```

Definition at line [234](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PWD_SCL_SHIFT

```
#define PCI_EXP_DEVCAP_PWD_SCL_SHIFT 26
```

Definition at line [240](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PWR_IND

```
#define PCI_EXP_DEVCAP_PWR_IND 0x00004000
```

Power Indicator Present.

Definition at line [309](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PWR_SCL

```
#define PCI_EXP_DEVCAP_PWR_SCL 0x0c000000
```

Slot Power Limit Scale.

Definition at line [312](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PWR_VAL

```
#define PCI_EXP_DEVCAP_PWR_VAL 0x03fc0000
```

Slot Power Limit Value.

Definition at line [311](#) of file [pci_regs.h](#).

PCI_EXP_DEVCAP_PWR_VAL_SHIFT

```
#define PCI_EXP_DEVCAP_PWR_VAL_SHIFT 18
```

Definition at line 241 of file [pci_regs.h](#).

PCI_EXP_DEVCAP_RBER

```
#define PCI_EXP_DEVCAP_RBER 0x00008000
```

Role-Based Error Reporting.

Definition at line 310 of file [pci_regs.h](#).

PCI_EXP_DEVCTL

```
#define PCI_EXP_DEVCTL 8
```

Device Control.

Definition at line 314 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2

```
#define PCI_EXP_DEVCTL2 40
```

Device Control 2.

Definition at line 475 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_ARI

```
#define PCI_EXP_DEVCTL2_ARI 0x0020
```

Alternative Routing-ID.

Definition at line 479 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK

```
#define PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK 0x0080
```

Block atomic egress.

Definition at line 481 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_ATOMIC_REQ

```
#define PCI_EXP_DEVCTL2_ATOMIC_REQ 0x0040
```

Set Atomic requests.

Definition at line 480 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_COMP_TIMEOUT

```
#define PCI_EXP_DEVCTL2_COMP_TIMEOUT 0x000f
```

Completion Timeout Value.

Definition at line 478 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE

```
#define PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE 0x0010
```

End-End TLP Prefix Blocking.

Definition at line 477 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK

```
#define PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK 0x8000
```

End-End TLP Prefix Blocking.

Definition at line 489 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_IDO_CMP_EN

```
#define PCI_EXP_DEVCTL2_IDO_CMP_EN 0x0200
```

Allow IDO for completions.

Definition at line 483 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_IDO_REQ_EN

```
#define PCI_EXP_DEVCTL2_IDO_REQ_EN 0x0100
```

Allow IDO for requests.

Definition at line 482 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_LTR_EN

```
#define PCI_EXP_DEVCTL2_LTR_EN 0x0400
```

Enable LTR mechanism.

Definition at line 484 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_OBFF_MSGA_EN

```
#define PCI_EXP_DEVCTL2_OBFF_MSGA_EN 0x2000
```

Enable OBFF Message type A.

Definition at line 485 of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_OBFF_MSGB_EN

```
#define PCI_EXP_DEVCTL2_OBFF_MSGB_EN 0x4000
```

Enable OBFF Message type B.

Definition at line [486](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL2_OBFF_WAKE_EN

```
#define PCI_EXP_DEVCTL2_OBFF_WAKE_EN 0x6000
```

OBFF using WAKE# signaling.

Definition at line [487](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_AUX_PME

```
#define PCI_EXP_DEVCTL_AUX_PME 0x0400
```

Auxiliary Power PM Enable.

Definition at line [323](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_BCR_FLR

```
#define PCI_EXP_DEVCTL_BCR_FLR 0x8000
```

Bridge Configuration Retry / FLR.

Definition at line [330](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_CERE

```
#define PCI_EXP_DEVCTL_CERE 0x0001
```

Correctable Error Reporting En.

Definition at line [315](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_EXT_TAG

```
#define PCI_EXP_DEVCTL_EXT_TAG 0x0100
```

Extended Tag Field Enable.

Definition at line [321](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_FERE

```
#define PCI_EXP_DEVCTL_FERE 0x0004
```

Fatal Error Reporting Enable.

Definition at line [317](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_NFERE

```
#define PCI_EXP_DEVCTL_NFERE 0x0002
```

Non-Fatal Error Reporting Enable.

Definition at line [316](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_NOSNOOP_EN

```
#define PCI_EXP_DEVCTL_NOSNOOP_EN 0x0800
```

Enable No Snoop.

Definition at line [324](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_PAYLOAD

```
#define PCI_EXP_DEVCTL_PAYLOAD 0x00e0
```

Max_Payload_Size.

Definition at line [320](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_PAYLOAD_SHIFT

```
#define PCI_EXP_DEVCTL_PAYLOAD_SHIFT 5
```

Definition at line [242](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_PHANTOM

```
#define PCI_EXP_DEVCTL_PHANTOM 0x0200
```

Phantom Functions Enable.

Definition at line [322](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ

```
#define PCI_EXP_DEVCTL_READRQ 0x7000
```

Max_Read_Request_Size.

Definition at line [325](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ_1024B

```
#define PCI_EXP_DEVCTL_READRQ_1024B 0x3000
```

1024 Bytes

Definition at line [329](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ_128B

```
#define PCI_EXP_DEVCTL_READRQ_128B 0x0000
```

128 Bytes

Definition at line [326](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ_256B

```
#define PCI_EXP_DEVCTL_READRQ_256B 0x1000
```

256 Bytes

Definition at line [327](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ_512B

```
#define PCI_EXP_DEVCTL_READRQ_512B 0x2000
```

512 Bytes

Definition at line [328](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_READRQ_SHIFT

```
#define PCI_EXP_DEVCTL_READRQ_SHIFT 12
```

Definition at line [236](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_RELAX_EN

```
#define PCI_EXP_DEVCTL_RELAX_EN 0x0010
```

Enable relaxed ordering.

Definition at line [319](#) of file [pci_regs.h](#).

PCI_EXP_DEVCTL_URRE

```
#define PCI_EXP_DEVCTL_URRE 0x0008
```

Unsupported Request Reporting En.

Definition at line [318](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA

```
#define PCI_EXP_DEVSTA 10
```

Device Status.

Definition at line [331](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA2

```
#define PCI_EXP_DEVSTA2 42
```

Device Status 2.

Definition at line [490](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_AUXPD

```
#define PCI_EXP_DEVSTA_AUXPD 0x0010
```

AUX Power Detected.

Definition at line [336](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_CED

```
#define PCI_EXP_DEVSTA_CED 0x0001
```

Correctable Error Detected.

Definition at line [332](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_FED

```
#define PCI_EXP_DEVSTA_FED 0x0004
```

Fatal Error Detected.

Definition at line [334](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_NFED

```
#define PCI_EXP_DEVSTA_NFED 0x0002
```

Non-Fatal Error Detected.

Definition at line [333](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_TRPND

```
#define PCI_EXP_DEVSTA_TRPND 0x0020
```

Transactions Pending.

Definition at line [337](#) of file [pci_regs.h](#).

PCI_EXP_DEVSTA_URD

```
#define PCI_EXP_DEVSTA_URD 0x0008
```

Unsupported Request Detected.

Definition at line [335](#) of file [pci_regs.h](#).

PCI_EXP_FLAGS

```
#define PCI_EXP_FLAGS 2
```

Capabilities register.

Definition at line 287 of file [pci_regs.h](#).

PCI_EXP_FLAGS_IRQ

```
#define PCI_EXP_FLAGS_IRQ 0x3e00
```

Interrupt message number.

Definition at line 300 of file [pci_regs.h](#).

PCI_EXP_FLAGS_SLOT

```
#define PCI_EXP_FLAGS_SLOT 0x0100
```

Slot implemented.

Definition at line 299 of file [pci_regs.h](#).

PCI_EXP_FLAGS_TYPE

```
#define PCI_EXP_FLAGS_TYPE 0x00f0
```

Device/Port type.

Definition at line 289 of file [pci_regs.h](#).

PCI_EXP_FLAGS_TYPE_SHIFT

```
#define PCI_EXP_FLAGS_TYPE_SHIFT 9
```

Definition at line 238 of file [pci_regs.h](#).

PCI_EXP_FLAGS_VERS

```
#define PCI_EXP_FLAGS_VERS 0x000f
```

Capability version.

Definition at line 288 of file [pci_regs.h](#).

PCI_EXP_LNKCAP

```
#define PCI_EXP_LNKCAP 12
```

Link Capabilities.

Definition at line 338 of file [pci_regs.h](#).

PCI_EXP_LNKCAP2

```
#define PCI_EXP_LNKCAP2 44
```

Link Capabilities 2.

Definition at line [492](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP2_CROSSLINK

```
#define PCI_EXP_LNKCAP2_CROSSLINK 0x00000100
```

Crosslink supported.

Definition at line [496](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP2_SLS_2_5GB

```
#define PCI_EXP_LNKCAP2_SLS_2_5GB 0x00000002
```

Supported Speed 2.5GT/s.

Definition at line [493](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP2_SLS_5_0GB

```
#define PCI_EXP_LNKCAP2_SLS_5_0GB 0x00000004
```

Supported Speed 5.0GT/s.

Definition at line [494](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP2_SLS_8_0GB

```
#define PCI_EXP_LNKCAP2_SLS_8_0GB 0x00000008
```

Supported Speed 8.0GT/s.

Definition at line [495](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_ASPMS

```
#define PCI_EXP_LNKCAP_ASPMS 0x00000c00
```

ASPM Support.

Definition at line [343](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_ASPMS_SHIFT

```
#define PCI_EXP_LNKCAP_ASPMS_SHIFT 10
```

Definition at line [244](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_CLKPM

```
#define PCI_EXP_LNKCAP_CLKPM 0x00040000
```

Clock Power Management.

Definition at line [346](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_DLLLARC

```
#define PCI_EXP_LNKCAP_DLLLARC 0x00100000
```

Data Link Layer Link Active Reporting Capable.

Definition at line [350](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_L0SEL

```
#define PCI_EXP_LNKCAP_L0SEL 0x00007000
```

L0s Exit Latency.

Definition at line [344](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_L0SEL_SHIFT

```
#define PCI_EXP_LNKCAP_L0SEL_SHIFT 12
```

Definition at line [245](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_L1EL

```
#define PCI_EXP_LNKCAP_L1EL 0x00038000
```

L1 Exit Latency.

Definition at line [345](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_L1EL_SHIFT

```
#define PCI_EXP_LNKCAP_L1EL_SHIFT 15
```

Definition at line [246](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_LBNC

```
#define PCI_EXP_LNKCAP_LBNC 0x00200000
```

Link Bandwidth Notification Capability.

Definition at line [352](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_MLW

```
#define PCI_EXP_LNKCAP_MLW 0x0000003f0
```

Maximum Link Width.

Definition at line [342](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_MLW_SHIFT

```
#define PCI_EXP_LNKCAP_MLW_SHIFT 4
```

Definition at line [243](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_PN

```
#define PCI_EXP_LNKCAP_PN 0xff000000
```

Port Number.

Definition at line [353](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_SDERC

```
#define PCI_EXP_LNKCAP_SDERC 0x00080000
```

Surprise Down Error Reporting Capable.

Definition at line [348](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_SLS

```
#define PCI_EXP_LNKCAP_SLS 0x0000000f
```

Supported Link Speeds.

Definition at line [339](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_SLS_2_5GB

```
#define PCI_EXP_LNKCAP_SLS_2_5GB 0x00000001
```

LNKCAP2 SLS Vector bit 0.

Definition at line [340](#) of file [pci_regs.h](#).

PCI_EXP_LNKCAP_SLS_5_0GB

```
#define PCI_EXP_LNKCAP_SLS_5_0GB 0x00000002
```

LNKCAP2 SLS Vector bit 1.

Definition at line [341](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL

```
#define PCI_EXP_LNKCTL 16
```

Link Control.

Definition at line [354](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2

```
#define PCI_EXP_LNKCTL2 48
```

Link Control 2.

Definition at line [497](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_COMP_SOS

```
#define PCI_EXP_LNKCTL2_COMP_SOS 0x00000800
```

Compliance SOS.

Definition at line [511](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL

```
#define PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL 0x00001000
```

De-emphasis level polling.

Definition at line [513](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_ENTER_COMP

```
#define PCI_EXP_LNKCTL2_ENTER_COMP 0x00000010
```

Enter Compliance.

Definition at line [503](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_ENTER_MOD_COMP

```
#define PCI_EXP_LNKCTL2_ENTER_MOD_COMP 0x00000400
```

Enter Modified Compliance.

Definition at line [510](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS

```
#define PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS 0x00000020
```

Hardware Autonomous Speed Disable.

Definition at line [505](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_LNK_SPEED_2_5

```
#define PCI_EXP_LNKCTL2_LNK_SPEED_2_5 0x00000001
```

Link Speed 2.5 GT/s.

Definition at line [498](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_LNK_SPEED_5_0

```
#define PCI_EXP_LNKCTL2_LNK_SPEED_5_0 0x00000002
```

Link Speed 5 GT/s.

Definition at line [499](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_LNK_SPEED_8_0

```
#define PCI_EXP_LNKCTL2_LNK_SPEED_8_0 0x00000003
```

Link Speed 8 GT/s.

Definition at line [500](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH

```
#define PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH 0x00000040
```

Selectable De-emphasis.

Definition at line [507](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK

```
#define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK 0x00000380
```

Transmit Margin.

Definition at line [508](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT

```
#define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT 22
```

Definition at line [251](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL

```
#define PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL 0x0000f000
```

Transmitter Preset in polling.

Definition at line [515](#) of file [pci_regs.h](#).

PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK

```
#define PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK 0x0000000f
```

Target Lnk Speed Mask.

Definition at line 502 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_ASPM_L0S

```
#define PCI_EXP_LNKCTL_ASPM_L0S 0x0001
```

L0s Enable.

Definition at line 356 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_ASPM_L1

```
#define PCI_EXP_LNKCTL_ASPM_L1 0x0002
```

L1 Enable.

Definition at line 357 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_ASPMC

```
#define PCI_EXP_LNKCTL_ASPMC 0x0003
```

ASPM Control.

Definition at line 355 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_CCC

```
#define PCI_EXP_LNKCTL_CCC 0x0040
```

Common Clock Configuration.

Definition at line 361 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_CLKREQ_EN

```
#define PCI_EXP_LNKCTL_CLKREQ_EN 0x0100
```

Enable clkreq.

Definition at line 363 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_ES

```
#define PCI_EXP_LNKCTL_ES 0x0080
```

Extended Synch.

Definition at line 362 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_HAWD

```
#define PCI_EXP_LNKCTL_HAWD 0x0200
```

Hardware Autonomous Width Disable.

Definition at line 364 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_LABIE

```
#define PCI_EXP_LNKCTL_LABIE 0x0800
```

Link Autonomous Bandwidth Interrupt Enable.

Definition at line 368 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_LBMIE

```
#define PCI_EXP_LNKCTL_LBMIE 0x0400
```

Link Bandwidth Management Interrupt Enable.

Definition at line 366 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_LD

```
#define PCI_EXP_LNKCTL_LD 0x0010
```

Link Disable.

Definition at line 359 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_RCB

```
#define PCI_EXP_LNKCTL_RCB 0x0008
```

Read Completion Boundary.

Definition at line 358 of file [pci_regs.h](#).

PCI_EXP_LNKCTL_RL

```
#define PCI_EXP_LNKCTL_RL 0x0020
```

Retrain Link.

Definition at line 360 of file [pci_regs.h](#).

PCI_EXP_LNKSTA

```
#define PCI_EXP_LNKSTA 18
```

Link Status.

Definition at line 369 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2

```
#define PCI_EXP_LNKSTA2 50/** Link Status 2 */
```

Definition at line 516 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_CDL

```
#define PCI_EXP_LNKSTA2_CDL 0x00000001
```

Current De-emphasis level (at 5 GT/s speed only)

Definition at line 518 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_EQUALIZ_COMP

```
#define PCI_EXP_LNKSTA2_EQUALIZ_COMP 0x00000002
```

Equalization Complete.

Definition at line 519 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_EQUALIZ_PH1

```
#define PCI_EXP_LNKSTA2_EQUALIZ_PH1 0x00000004
```

Equalization Ph.1 Successful.

Definition at line 520 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_EQUALIZ_PH2

```
#define PCI_EXP_LNKSTA2_EQUALIZ_PH2 0x00000008
```

Equalization Ph.2 Successful.

Definition at line 521 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_EQUALIZ_PH3

```
#define PCI_EXP_LNKSTA2_EQUALIZ_PH3 0x00000010
```

Equalization Ph.3 Successful.

Definition at line 522 of file [pci_regs.h](#).

PCI_EXP_LNKSTA2_LINE_EQ_REQ

```
#define PCI_EXP_LNKSTA2_LINE_EQ_REQ 0x00000020
```

Link Equalization Request.

Definition at line 523 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_CLS

```
#define PCI_EXP_LNKSTA_CLS 0x000f
```

Current Link Speed.

Definition at line 370 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_CLS_2_5GB

```
#define PCI_EXP_LNKSTA_CLS_2_5GB 0x0001
```

Current Link Speed 2.5GT/s.

Definition at line 371 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_CLS_5_0GB

```
#define PCI_EXP_LNKSTA_CLS_5_0GB 0x0002
```

Current Link Speed 5.0GT/s.

Definition at line 372 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_CLS_8_0GB

```
#define PCI_EXP_LNKSTA_CLS_8_0GB 0x0003
```

Current Link Speed 8.0GT/s.

Definition at line 373 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_DLLLA

```
#define PCI_EXP_LNKSTA_DLLLA 0x2000
```

Data Link Layer Link Active.

Definition at line 382 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_LABS

```
#define PCI_EXP_LNKSTA_LABS 0x8000
```

Link Autonomous Bandwidth Status.

Definition at line 384 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_LBMS

```
#define PCI_EXP_LNKSTA_LBMS 0x4000
```

Link Bandwidth Management Status.

Definition at line 383 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_LT

```
#define PCI_EXP_LNKSTA_LT 0x0800
```

Link Training.

Definition at line 380 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW

```
#define PCI_EXP_LNKSTA_NLW 0x03f0
```

Negotiated Link Width.

Definition at line 374 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW_SHIFT

```
#define PCI_EXP_LNKSTA_NLW_SHIFT 4
```

start of NLW mask in link status

Definition at line 379 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW_X1

```
#define PCI_EXP_LNKSTA_NLW_X1 0x0010
```

Current Link Width x1.

Definition at line 375 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW_X2

```
#define PCI_EXP_LNKSTA_NLW_X2 0x0020
```

Current Link Width x2.

Definition at line 376 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW_X4

```
#define PCI_EXP_LNKSTA_NLW_X4 0x0040
```

Current Link Width x4.

Definition at line 377 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_NLW_X8

```
#define PCI_EXP_LNKSTA_NLW_X8 0x0080
```

Current Link Width x8.

Definition at line 378 of file [pci_regs.h](#).

PCI_EXP_LNKSTA_SLC

```
#define PCI_EXP_LNKSTA_SLC 0x1000
```

Slot Clock Configuration.

Definition at line [381](#) of file [pci_regs.h](#).

PCI_EXP_RTCAP

```
#define PCI_EXP_RTCAP 30
```

Root Capabilities.

Definition at line [436](#) of file [pci_regs.h](#).

PCI_EXP_RTCAP_CRSVIS

```
#define PCI_EXP_RTCAP_CRSVIS 0x0001
```

CRS Software Visibility capability.

Definition at line [437](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL

```
#define PCI_EXP_RTCTL 28
```

Root Control.

Definition at line [430](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL_CRSSVE

```
#define PCI_EXP_RTCTL_CRSSVE 0x0010
```

CRS Software Visibility Enable.

Definition at line [435](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL_PMEIE

```
#define PCI_EXP_RTCTL_PMEIE 0x0008
```

PME Interrupt Enable.

Definition at line [434](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL_SECEE

```
#define PCI_EXP_RTCTL_SECEE 0x0001
```

System Error on Correctable Error.

Definition at line [431](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL_SEFEE

```
#define PCI_EXP_RTCTL_SEFEE 0x0004
```

System Error on Fatal Error.

Definition at line [433](#) of file [pci_regs.h](#).

PCI_EXP_RTCTL_SENFEE

```
#define PCI_EXP_RTCTL_SENFEE 0x0002
```

System Error on Non-Fatal Error.

Definition at line [432](#) of file [pci_regs.h](#).

PCI_EXP_RTSTA

```
#define PCI_EXP_RTSTA 32
```

Root Status.

Definition at line [438](#) of file [pci_regs.h](#).

PCI_EXP_RTSTA_PENDING

```
#define PCI_EXP_RTSTA_PENDING 0x00020000
```

PME pending.

Definition at line [440](#) of file [pci_regs.h](#).

PCI_EXP_RTSTA_PME

```
#define PCI_EXP_RTSTA_PME 0x00010000
```

PME status.

Definition at line [439](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP

```
#define PCI_EXP_SLTCAP 20
```

Slot Capabilities.

Definition at line [386](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP2

```
#define PCI_EXP_SLTCAP2 52
```

Slot Capabilities 2.

Definition at line [524](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_ABP

```
#define PCI_EXP_SLTCAP_ABP 0x00000001
```

Attention Button Present.

Definition at line [387](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_AIC_SHIFT

```
#define PCI_EXP_SLTCAP_AIC_SHIFT 6
```

Definition at line [248](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_AIP

```
#define PCI_EXP_SLTCAP_AIP 0x00000008
```

Attention Indicator Present.

Definition at line [390](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_EIP

```
#define PCI_EXP_SLTCAP_EIP 0x00020000
```

Electromechanical Interlock Present.

Definition at line [396](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_HPC

```
#define PCI_EXP_SLTCAP_HPC 0x00000040
```

Hot-Plug Capable.

Definition at line [393](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_HPS

```
#define PCI_EXP_SLTCAP_HPS 0x00000020
```

Hot-Plug Surprise.

Definition at line [392](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_MRLSP

```
#define PCI_EXP_SLTCAP_MRLSP 0x00000004
```

MRL Sensor Present.

Definition at line [389](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_NCCS

```
#define PCI_EXP_SLTCAP_NCCS 0x00040000
```

No Command Completed Support.

Definition at line [397](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_PCP

```
#define PCI_EXP_SLTCAP_PCP 0x00000002
```

Power Controller Present.

Definition at line [388](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_PIP

```
#define PCI_EXP_SLTCAP_PIP 0x00000010
```

Power Indicator Present.

Definition at line [391](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_PSN

```
#define PCI_EXP_SLTCAP_PSN 0xffff80000
```

Physical Slot Number.

Definition at line [398](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_SPLS

```
#define PCI_EXP_SLTCAP_SPLS 0x00018000
```

Slot Power Limit Scale.

Definition at line [395](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_SPLS_SHIFT

```
#define PCI_EXP_SLTCAP_SPLS_SHIFT 15
```

Definition at line [247](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_SPLV

```
#define PCI_EXP_SLTCAP_SPLV 0x00007f80
```

Slot Power Limit Value.

Definition at line [394](#) of file [pci_regs.h](#).

PCI_EXP_SLTCAP_SPLV_SHIFT

```
#define PCI_EXP_SLTCAP_SPLV_SHIFT 7
```

Definition at line 237 of file [pci_regs.h](#).

PCI_EXP_SLTCTL

```
#define PCI_EXP_SLTCTL 24
```

Slot Control.

Definition at line 399 of file [pci_regs.h](#).

PCI_EXP_SLTCTL2

```
#define PCI_EXP_SLTCTL2 56
```

Slot Control 2.

Definition at line 525 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_ABPE

```
#define PCI_EXP_SLTCTL_ABPE 0x0001
```

Attention Button Pressed Enable.

Definition at line 400 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_AIC

```
#define PCI_EXP_SLTCTL_AIC 0x00c0
```

Attention Indicator Control.

Definition at line 406 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_ATTN_IND_BLINK

```
#define PCI_EXP_SLTCTL_ATTN_IND_BLINK 0x0080
```

Attention Indicator blinking.

Definition at line 408 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_ATTN_IND_OFF

```
#define PCI_EXP_SLTCTL_ATTN_IND_OFF 0x00c0
```

Attention Indicator off.

Definition at line 409 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_ATTN_IND_ON

```
#define PCI_EXP_SLTCTL_ATTN_IND_ON 0x0040
```

Attention Indicator on.

Definition at line [407](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_CCIE

```
#define PCI_EXP_SLTCTL_CCIE 0x0010
```

Command Completed Interrupt Enable.

Definition at line [404](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_DLLSCE

```
#define PCI_EXP_SLTCTL_DLLSCE 0x1000
```

Data Link Layer State Changed Enable.

Definition at line [419](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_EIC

```
#define PCI_EXP_SLTCTL_EIC 0x0800
```

Electromechanical Interlock Control.

Definition at line [417](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_HPIE

```
#define PCI_EXP_SLTCTL_HPIE 0x0020
```

Hot-Plug Interrupt Enable.

Definition at line [405](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_MRLSCE

```
#define PCI_EXP_SLTCTL_MRLSCE 0x0004
```

MRL Sensor Changed Enable.

Definition at line [402](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PCC

```
#define PCI_EXP_SLTCTL_PCC 0x0400
```

Power Controller Control.

Definition at line [414](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PDCE

```
#define PCI_EXP_SLTCTL_PDCE 0x0008
```

Presence Detect Changed Enable.

Definition at line [403](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PFDE

```
#define PCI_EXP_SLTCTL_PFDE 0x0002
```

Power Fault Detected Enable.

Definition at line [401](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PIC

```
#define PCI_EXP_SLTCTL_PIC 0x0300
```

Power Indicator Control.

Definition at line [410](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PIC_SHIFT

```
#define PCI_EXP_SLTCTL_PIC_SHIFT 8
```

Definition at line [249](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PWR_IND_BLINK

```
#define PCI_EXP_SLTCTL_PWR_IND_BLINK 0x0200
```

Power Indicator blinking.

Definition at line [412](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PWR_IND_OFF

```
#define PCI_EXP_SLTCTL_PWR_IND_OFF 0x0300
```

Power Indicator off.

Definition at line [413](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PWR_IND_ON

```
#define PCI_EXP_SLTCTL_PWR_IND_ON 0x0100
```

Power Indicator on.

Definition at line [411](#) of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PWR_OFF

```
#define PCI_EXP_SLTCTL_PWR_OFF 0x0400
```

Power Off.

Definition at line 416 of file [pci_regs.h](#).

PCI_EXP_SLTCTL_PWR_ON

```
#define PCI_EXP_SLTCTL_PWR_ON 0x0000
```

Power On.

Definition at line 415 of file [pci_regs.h](#).

PCI_EXP_SLTSTA

```
#define PCI_EXP_SLTSTA 26
```

Slot Status.

Definition at line 420 of file [pci_regs.h](#).

PCI_EXP_SLTSTA2

```
#define PCI_EXP_SLTSTA2 58
```

Slot Status 2.

Definition at line 526 of file [pci_regs.h](#).

PCI_EXP_SLTSTA_ABP

```
#define PCI_EXP_SLTSTA_ABP 0x0001
```

Attention Button Pressed.

Definition at line 421 of file [pci_regs.h](#).

PCI_EXP_SLTSTA_CC

```
#define PCI_EXP_SLTSTA_CC 0x0010
```

Command Completed.

Definition at line 425 of file [pci_regs.h](#).

PCI_EXP_SLTSTA_DLLSC

```
#define PCI_EXP_SLTSTA_DLLSC 0x0100
```

Data Link Layer State Changed.

Definition at line 429 of file [pci_regs.h](#).

PCI_EXP_SLTSTA_EIS

```
#define PCI_EXP_SLTSTA_EIS 0x0080
```

Electromechanical Interlock Status.

Definition at line [428](#) of file [pci_regs.h](#).

PCI_EXP_SLTSTA_MRLSC

```
#define PCI_EXP_SLTSTA_MRLSC 0x0004
```

MRL Sensor Changed.

Definition at line [423](#) of file [pci_regs.h](#).

PCI_EXP_SLTSTA_MRLSS

```
#define PCI_EXP_SLTSTA_MRLSS 0x0020
```

MRL Sensor State.

Definition at line [426](#) of file [pci_regs.h](#).

PCI_EXP_SLTSTA_PDC

```
#define PCI_EXP_SLTSTA_PDC 0x0008
```

Presence Detect Changed.

Definition at line [424](#) of file [pci_regs.h](#).

PCI_EXP_SLTSTA_PDS

```
#define PCI_EXP_SLTSTA_PDS 0x0040
```

Presence Detect State.

Definition at line [427](#) of file [pci_regs.h](#).

PCI_EXP_SLTSTA_PFD

```
#define PCI_EXP_SLTSTA_PFD 0x0002
```

Power Fault Detected.

Definition at line [422](#) of file [pci_regs.h](#).

PCI_EXP_TYPE_DOWNSTREAM

```
#define PCI_EXP_TYPE_DOWNSTREAM 0x6
```

Downstream Port.

Definition at line [294](#) of file [pci_regs.h](#).

PCI_EXP_TYPE_ENDPOINT

```
#define PCI_EXP_TYPE_ENDPOINT 0x0
```

Express Endpoint.

Definition at line 290 of file [pci_regs.h](#).

PCI_EXP_TYPE_LEG_END

```
#define PCI_EXP_TYPE_LEG_END 0x1
```

Legacy Endpoint.

Definition at line 291 of file [pci_regs.h](#).

PCI_EXP_TYPE_PCI_BRIDGE

```
#define PCI_EXP_TYPE_PCI_BRIDGE 0x7
```

PCIe to PCI/PCI-X Bridge.

Definition at line 295 of file [pci_regs.h](#).

PCI_EXP_TYPE_PCIE_BRIDGE

```
#define PCI_EXP_TYPE_PCIE_BRIDGE 0x8
```

PCI/PCI-X to PCIe Bridge.

Definition at line 296 of file [pci_regs.h](#).

PCI_EXP_TYPE_RC_EC

```
#define PCI_EXP_TYPE_RC_EC 0xa
```

Root Complex Event Collector.

Definition at line 298 of file [pci_regs.h](#).

PCI_EXP_TYPE_RC_END

```
#define PCI_EXP_TYPE_RC_END 0x9
```

Root Complex Integrated Endpoint.

Definition at line 297 of file [pci_regs.h](#).

PCI_EXP_TYPE_ROOT_PORT

```
#define PCI_EXP_TYPE_ROOT_PORT 0x4
```

Root Port.

Definition at line 292 of file [pci_regs.h](#).

PCI_EXP_TYPE_UPSTREAM

```
#define PCI_EXP_TYPE_UPSTREAM 0x5
```

Upstream Port.

Definition at line [293](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID

```
#define PCI_EXT_CAP_ID ( header ) (header & 0x0000ffff)
```

Definition at line [134](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_ACS

```
#define PCI_EXT_CAP_ID_ACS 0x000D
```

Access Control Services.

Definition at line [150](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_AMD_XXX

```
#define PCI_EXT_CAP_ID_AMD_XXX 0x0014
```

Reserved for AMD.

Definition at line [157](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_ARI

```
#define PCI_EXT_CAP_ID_ARI 0x000E
```

Alternate Routing ID.

Definition at line [151](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_ATS

```
#define PCI_EXT_CAP_ID_ATS 0x000F
```

Address Translation Services.

Definition at line [152](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_CAC

```
#define PCI_EXT_CAP_ID_CAC 0x000C
```

Config Access - obsolete.

Definition at line [149](#) of file [pci_regs.h](#).

PCI_EXT_CAP_ID_DPA

```
#define PCI_EXT_CAP_ID_DPA 0x0016
```

Dynamic Power Allocation.

Definition at line 159 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_DPC

```
#define PCI_EXT_CAP_ID_DPC 0x001D
```

Downstream Port Containment (DPC)

Definition at line 166 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_DSN

```
#define PCI_EXT_CAP_ID_DSN 0x0003
```

Device Serial Number.

Definition at line 140 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_ERR

```
#define PCI_EXT_CAP_ID_ERR 0x0001
```

Advanced Error Reporting.

Definition at line 138 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_FRSQ

```
#define PCI_EXT_CAP_ID_FRSQ 0x0021
```

FRS Queueing.

Definition at line 170 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_L1PMS

```
#define PCI_EXT_CAP_ID_L1PMS 0x001E
```

L1 PM Substates.

Definition at line 167 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_LNR

```
#define PCI_EXT_CAP_ID_LNR 0x001C
```

LN Requester (LNR)

Definition at line 165 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_LTR

```
#define PCI_EXT_CAP_ID_LTR 0x0018
```

Latency Tolerance Reporting.

Definition at line 161 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_MCAST

```
#define PCI_EXT_CAP_ID_MCAST 0x0012
```

Multicast.

Definition at line 155 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_MFVC

```
#define PCI_EXT_CAP_ID_MFVC 0x0008
```

Multi-Function VC Capability.

Definition at line 145 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_MPHY

```
#define PCI_EXT_CAP_ID_MPHY 0x0020
```

PCI Express over M-PHY (M-PCIe)

Definition at line 169 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_MRIOV

```
#define PCI_EXT_CAP_ID_MRIOV 0x0011
```

Multi Root I/O Virtualization.

Definition at line 154 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_PASID

```
#define PCI_EXT_CAP_ID_PASID 0x001B
```

Process Address Space ID.

Definition at line 164 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_PMUX

```
#define PCI_EXT_CAP_ID_PMUX 0x001A
```

Protocol Multiplexing.

Definition at line 163 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_PRI

```
#define PCI_EXT_CAP_ID_PRI 0x0013
```

Page Request Interface.

Definition at line 156 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_PTM

```
#define PCI_EXT_CAP_ID_PTM 0x001F
```

Precision Time Measurement (PTM)

Definition at line 168 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_PWR

```
#define PCI_EXT_CAP_ID_PWR 0x0004
```

Power Budgeting.

Definition at line 141 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_RCEC

```
#define PCI_EXT_CAP_ID_RCEC 0x0007
```

Root Complex Event Collector.

Definition at line 144 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_RCILC

```
#define PCI_EXT_CAP_ID_RCILC 0x0006
```

Root Complex Internal Link Control.

Definition at line 143 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_RCLD

```
#define PCI_EXT_CAP_ID_RCLD 0x0005
```

Root Complex Link Declaration.

Definition at line 142 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_RCRB

```
#define PCI_EXT_CAP_ID_RCRB 0x000A
```

Root Complex RB?

Definition at line 147 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_REBAR

```
#define PCI_EXT_CAP_ID_REBAR 0x0015
```

Resizable BAR.

Definition at line 158 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_RTR

```
#define PCI_EXT_CAP_ID_RTR 0x0022
```

Readiness Time Reporting.

Definition at line 171 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_SECPCI

```
#define PCI_EXT_CAP_ID_SECPCI 0x0019
```

Secondary PCIe Capability.

Definition at line 162 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_SRIOV

```
#define PCI_EXT_CAP_ID_SRIOV 0x0010
```

Single Root I/O Virtualization.

Definition at line 153 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_TPH

```
#define PCI_EXT_CAP_ID_TPH 0x0017
```

TPH Requester.

Definition at line 160 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_VC

```
#define PCI_EXT_CAP_ID_VC 0x0002
```

Virtual Channel Capability.

Definition at line 139 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_VC9

```
#define PCI_EXT_CAP_ID_VC9 0x0009
```

same as _VC

Definition at line 146 of file [pci_regs.h](#).

PCI_EXT_CAP_ID_VNDR

```
#define PCI_EXT_CAP_ID_VNDR 0x000B
```

Vendor-Specific.

Definition at line 148 of file [pci_regs.h](#).

PCI_EXT_CAP_NEXT

```
#define PCI_EXT_CAP_NEXT( header ) ((header >> 20) & 0xffc)
```

Definition at line 136 of file [pci_regs.h](#).

PCI_EXT_CAP_VER

```
#define PCI_EXT_CAP_VER( header ) ((header >> 16) & 0xf)
```

Definition at line 135 of file [pci_regs.h](#).

PCI_HEADER_TYPE

```
#define PCI_HEADER_TYPE 0x0e
```

8 bits

Definition at line 88 of file [pci_regs.h](#).

PCI_HEADER_TYPE_BRIDGE

```
#define PCI_HEADER_TYPE_BRIDGE 1
```

Definition at line 90 of file [pci_regs.h](#).

PCI_HEADER_TYPE_CARDBUS

```
#define PCI_HEADER_TYPE_CARDBUS 2
```

Definition at line 91 of file [pci_regs.h](#).

PCI_HEADER_TYPE_NORMAL

```
#define PCI_HEADER_TYPE_NORMAL 0
```

Definition at line 89 of file [pci_regs.h](#).

PCI_SR_CAP_LIST_BIT

```
#define PCI_SR_CAP_LIST_BIT 0x00000010
```

Definition at line 93 of file [pci_regs.h](#).

PCI_STATUS

```
#define PCI_STATUS 0x06
```

16 bits

Definition at line 266 of file [pci_regs.h](#).

PCI_STATUS_66MHZ

```
#define PCI_STATUS_66MHZ 0x20
```

Support 66 Mhz PCI 2.1 bus.

Definition at line 269 of file [pci_regs.h](#).

PCI_STATUS_CAP_LIST

```
#define PCI_STATUS_CAP_LIST 0x10
```

Support Capability List.

Definition at line 268 of file [pci_regs.h](#).

PCI_STATUS_DETECTED_PARITY

```
#define PCI_STATUS_DETECTED_PARITY 0x8000
```

Set on parity error.

Definition at line 282 of file [pci_regs.h](#).

PCI_STATUS_DEVSEL_FAST

```
#define PCI_STATUS_DEVSEL_FAST 0x000
```

Definition at line 275 of file [pci_regs.h](#).

PCI_STATUS_DEVSEL_MASK

```
#define PCI_STATUS_DEVSEL_MASK 0x600
```

DEVSEL timing.

Definition at line 274 of file [pci_regs.h](#).

PCI_STATUS_DEVSEL_MEDIUM

```
#define PCI_STATUS_DEVSEL_MEDIUM 0x200
```

Definition at line 276 of file [pci_regs.h](#).

PCI_STATUS_DEVSEL_SHIFT

```
#define PCI_STATUS_DEVSEL_SHIFT 9
```

Definition at line [235](#) of file [pci_regs.h](#).

PCI_STATUS_DEVSEL_SLOW

```
#define PCI_STATUS_DEVSEL_SLOW 0x400
```

Definition at line [277](#) of file [pci_regs.h](#).

PCI_STATUS_FAST_BACK

```
#define PCI_STATUS_FAST_BACK 0x80
```

Accept fast-back to back.

Definition at line [272](#) of file [pci_regs.h](#).

PCI_STATUS_INTERRUPT

```
#define PCI_STATUS_INTERRUPT 0x08
```

Interrupt status.

Definition at line [267](#) of file [pci_regs.h](#).

PCI_STATUS_PARITY

```
#define PCI_STATUS_PARITY 0x100
```

Detected parity error.

Definition at line [273](#) of file [pci_regs.h](#).

PCI_STATUS_REC_MASTER_ABORT

```
#define PCI_STATUS_REC_MASTER_ABORT 0x2000
```

Set on master abort.

Definition at line [280](#) of file [pci_regs.h](#).

PCI_STATUS_REC_TARGET_ABORT

```
#define PCI_STATUS_REC_TARGET_ABORT 0x1000
```

Master ack of abort.

Definition at line [279](#) of file [pci_regs.h](#).

PCI_STATUS_SIG_SYSTEM_ERROR

```
#define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000
```

Set when we drive SERR.

Definition at line [281](#) of file [pci_regs.h](#).

PCI_STATUS_SIG_TARGET_ABORT

```
#define PCI_STATUS_SIG_TARGET_ABORT 0x800
```

Set on target abort.

Definition at line [278](#) of file [pci_regs.h](#).

PCI_STATUS_UDF

```
#define PCI_STATUS_UDF 0x40
```

Support User Definable Features.

Definition at line [270](#) of file [pci_regs.h](#).

Enumeration Type Documentation

AD_PCI_BAR

```
enum AD_PCI_BAR
```

PCI base address spaces (BARs)

Enumerator

AD_PCI_BAR0	
AD_PCI_BAR1	
AD_PCI_BAR2	
AD_PCI_BAR3	
AD_PCI_BAR4	
AD_PCI_BAR5	
AD_PCI_BARS	

Definition at line [96](#) of file [pci_regs.h](#).

PCI_CONFIG_REGS_OFFSET

```
enum PCI_CONFIG_REGS_OFFSET
```

Enumerator

PCI_VID	Vendor ID.
PCI_DID	Device ID.

Enumerator

PCI_CR	Command register.
PCI_SR	Status register.
PCI_REV	Revision ID.
PCI_CCR	Class code.
PCI_CCSC	Sub class code.
PCI_CCBC	Base class code.
PCI_CLSR	Cache line size.
PCI_LTR	Latency timer.
PCI_HDR	Header type.
PCI_BISTR	Built-in self test.
PCI_BAR0	Base address register.
PCI_BAR1	Base address register.
PCI_BAR2	Base address register.
PCI_BAR3	Base address register.
PCI_BAR4	Base address register.
PCI_BAR5	Base address register.
PCI_CIS	CardBus CIS pointer.
PCI_SVID	Sub-system vendor ID.
PCI_SDID	Sub-system device ID.
PCI_EROM	Expansion ROM base address.
PCI_CAP	New capability pointer.
PCI_ILR	Interrupt line.
PCI_IPR	Interrupt pin.
PCI_MGR	Minimum required burst period.
PCI_MLR	Maximum latency - How often device must gain PCI bus access.

Definition at line 29 of file [pci_regs.h](#).

PCIE_CONFIG_REGS_OFFSET

```
enum PCIE_CONFIG_REGS_OFFSET
```

Enumerator

PCIE_CAP_ID
NEXT_CAP_PTR
CAP_REG
DEV_CAPS
DEV_CTL
DEV_STS
LNK_CAPS
LNK_CTL
LNK_STS
SLOT_CAPS
SLOT_CTL
SLOT_STS
ROOT_CAPS

Enumerator

ROOT_CTL	
ROOT_STS	
DEV_CAPS2	
DEV_CTL2	
DEV_STS2	
LNK_CAPS2	
LNK_CTL2	
LNK_STS2	
SLOT_CAPS2	
SLOT_CTL2	
SLOT_STS2	

Definition at line 60 of file [pci_regs.h](#).

WDC_PCI_HEADER_TYPE

```
enum WDC_PCI_HEADER_TYPE
```

Enumerator

HEADER_TYPE_NORMAL	
HEADER_TYPE_BRIDGE	
HEADER_TYPE_CARDBUS	
HEADER_TYPE_NRML_BRIDGE	
HEADER_TYPE_NRML_CARDBUS	
HEADER_TYPE_BRIDGE_CARDBUS	
HEADER_TYPE_ALL	

Definition at line 14 of file [pci_regs.h](#).

pci_regs.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _PCI_REGS_H_
00004 #define _PCI_REGS_H_
00005
00006 /*****
00007 * File - pci_regs.h - PCI configuration space and address spaces definitions *
00008 *****/
00009
00010 /* -----
00011 * Register type definitions
00012 * ----- */
00013
00014 typedef enum
00015 {
00016     HEADER_TYPE_NORMAL = 0x01,
00017     HEADER_TYPE_BRIDGE = 0x02,
00018     HEADER_TYPE_CARDBUS = 0x04,
00019     HEADER_TYPE_NRML_BRIDGE = HEADER_TYPE_NORMAL | HEADER_TYPE_BRIDGE,
00020     HEADER_TYPE_NRML_CARDBUS = HEADER_TYPE_NORMAL | HEADER_TYPE_CARDBUS,
00021     HEADER_TYPE_BRIDGE_CARDBUS = HEADER_TYPE_BRIDGE | HEADER_TYPE_CARDBUS,
00022     HEADER_TYPE_ALL =
00023         HEADER_TYPE_NORMAL | HEADER_TYPE_BRIDGE | HEADER_TYPE_CARDBUS
00024 } WDC_PCI_HEADER_TYPE;
00025
```

```
00026 /* -----  
00027   PCI configuration registers offsets  
00028 ----- */  
00029 typedef enum {  
00030     PCI_VID      = 0x00,  
00031     PCI_DID      = 0x02,  
00032     PCI_CR       = 0x04,  
00033     PCI_SR       = 0x06,  
00034     PCI_REV      = 0x08,  
00035     PCI_CCR      = 0x09,  
00036     PCI_CCSC     = 0x0a,  
00037     PCI_CCBC     = 0x0b,  
00038     PCI_CLSR     = 0x0c,  
00039     PCI_LTR      = 0x0d,  
00040     PCI_HDR      = 0x0e,  
00041     PCI_BISTR    = 0x0f,  
00042     PCI_BAR0     = 0x10,  
00043     PCI_BAR1     = 0x14,  
00044     PCI_BAR2     = 0x18,  
00045     PCI_BAR3     = 0x1c,  
00046     PCI_BAR4     = 0x20,  
00047     PCI_BAR5     = 0x24,  
00048     PCI_CIS      = 0x28,  
00049     PCI_SVID     = 0x2c,  
00050     PCI_SDID     = 0x2e,  
00051     PCI_EROM     = 0x30,  
00052     PCI_CAP      = 0x34,  
00053     PCI_ILR      = 0x3c,  
00054     PCI_IPR      = 0x3d,  
00055     PCI_MGR      = 0x3e,  
00056     PCI_MLR      = 0x3f  
00058 } PCI_CONFIG_REGS_OFFSET;  
00059  
00060 typedef enum  
00061 {  
00062     PCIE_CAP_ID    = 0x0,  
00063     NEXT_CAP_PTR   = 0x1,  
00064     CAP_REG        = 0x2,  
00065     DEV_CAPS       = 0x4,  
00066     DEV_CTL        = 0x8,  
00067     DEV_STS        = 0xa,  
00068     LNK_CAPS       = 0xc,  
00069     LNK_CTL        = 0x10,  
00070     LNK_STS        = 0x12,  
00071     SLOT_CAPS     = 0x14,  
00072     SLOT_CTL       = 0x18,  
00073     SLOT_STS       = 0x1a,  
00074     ROOT_CAPS      = 0x1c,  
00075     ROOT_CTL        = 0x1e,  
00076     ROOT_STS        = 0x20,  
00077     DEV_CAPS2      = 0x24,  
00078     DEV_CTL2        = 0x28,  
00079     DEV_STS2        = 0x2a,  
00080     LNK_CAPS2      = 0x2c,  
00081     LNK_CTL2        = 0x30,  
00082     LNK_STS2        = 0x32,  
00083     SLOT_CAPS2     = 0x34,  
00084     SLOT_CTL2        = 0x38,  
00085     SLOT_STS2        = 0x3a  
00086 } PCIE_CONFIG_REGS_OFFSET;  
00087  
00088 #define PCI_HEADER_TYPE      0x0e  
00089 #define PCI_HEADER_TYPE_NORMAL 0  
00090 #define PCI_HEADER_TYPE_BRIDGE 1  
00091 #define PCI_HEADER_TYPE_CARDBUS 2  
00092  
00093 #define PCI_SR_CAP_LIST_BIT 0x00000010  
00094  
00095 typedef enum {  
00096     AD_PCI_BAR0 = 0,  
00097     AD_PCI_BAR1 = 1,  
00098     AD_PCI_BAR2 = 2,  
00099     AD_PCI_BAR3 = 3,  
00100     AD_PCI_BAR4 = 4,  
00101     AD_PCI_BAR5 = 5,  
00102     AD_PCI_BARS = 6  
00103 } AD_PCI_BAR;  
00104  
00105 /* PCI basic and extended capability lists last updated from-  
00106 *      PCI Code and ID Assignment Specification Revision 1.5 */  
00107 /* Some of the following strings and macros are taken from  
00108 * include/uapi/linux/pci_regs.h in Linux Kernel source */  
00109 /* Capability lists */  
00110 #define PCI_CAP_LIST_ID    0  
00111 #define PCI_CAP_ID_PM     0x01  
00112 #define PCI_CAP_ID_AGP    0x02  
00113 #define PCI_CAP_ID_VPD    0x03
```

```
00115 #define PCI_CAP_ID_SLOTID 0x04
00116 #define PCI_CAP_ID_MSI 0x05
00117 #define PCI_CAP_ID_CHSWP 0x06
00118 #define PCI_CAP_ID_PCIX 0x07
00119 #define PCI_CAP_ID_HT 0x08
00120 #define PCI_CAP_ID_VNDR 0x09
00121 #define PCI_CAP_ID_DBG 0x0A
00122 #define PCI_CAP_ID_CCRC 0x0B
00123 #define PCI_CAP_ID_SHPC 0x0C
00124 #define PCI_CAP_ID_SSVID 0x0D
00125 #define PCI_CAP_ID_AGP3 0x0E
00126 #define PCI_CAP_ID_SECDEV 0x0F
00127 #define PCI_CAP_ID_EXP 0x10
00128 #define PCI_CAP_ID_MSIX 0x11
00129 #define PCI_CAP_ID_SATA 0x12
00130 #define PCI_CAP_ID_AF 0x13
00131 #define PCI_CAP_LIST_NEXT 1
00133 /* Extended Capabilities (PCI-X 2.0 and Express) */
00134 #define PCI_EXT_CAP_ID(header) ((header & 0x0000ffff))
00135 #define PCI_EXT_CAP_VER(header) (((header >> 16) & 0xf)
00136 #define PCI_EXT_CAP_NEXT(header) (((header >> 20) & 0xffc)
00137
00138 #define PCI_EXT_CAP_ID_ERR 0x0001
00139 #define PCI_EXT_CAP_ID_VC 0x0002
00140 #define PCI_EXT_CAP_ID_DSN 0x0003
00141 #define PCI_EXT_CAP_ID_PWR 0x0004
00142 #define PCI_EXT_CAP_ID_RCLD 0x0005
00143 #define PCI_EXT_CAP_ID_RCILC 0x0006
00144 #define PCI_EXT_CAP_ID_RCEC 0x0007
00145 #define PCI_EXT_CAP_ID_MFVC 0x0008
00146 #define PCI_EXT_CAP_ID_VC9 0x0009
00147 #define PCI_EXT_CAP_ID_RCRB 0x000A
00148 #define PCI_EXT_CAP_ID_VNDR 0x000B
00149 #define PCI_EXT_CAP_ID_CAC 0x000C
00150 #define PCI_EXT_CAP_ID_ACS 0x000D
00151 #define PCI_EXT_CAP_ID_ARI 0x000E
00152 #define PCI_EXT_CAP_ID_ATS 0x000F
00153 #define PCI_EXT_CAP_ID_SRIOV 0x0010
00154 #define PCI_EXT_CAP_ID_MRIOV 0x0011
00155 #define PCI_EXT_CAP_ID_MCAST 0x0012
00156 #define PCI_EXT_CAP_ID_PRI 0x0013
00157 #define PCI_EXT_CAP_ID_AMD_XXX 0x0014
00158 #define PCI_EXT_CAP_ID_REBAR 0x0015
00159 #define PCI_EXT_CAP_ID_DPA 0x0016
00160 #define PCI_EXT_CAP_ID_TPH 0x0017
00161 #define PCI_EXT_CAP_ID_LTR 0x0018
00162 #define PCI_EXT_CAP_ID_SECPCI 0x0019
00163 #define PCI_EXT_CAP_ID_PMUX 0x001A
00164 #define PCI_EXT_CAP_ID_PASID 0x001B
00165 #define PCI_EXT_CAP_ID_LNR 0x001C
00166 #define PCI_EXT_CAP_ID_DPC 0x001D
00167 #define PCI_EXT_CAP_ID_L1PMS 0x001E
00168 #define PCI_EXT_CAP_ID_PT 0x001F
00169 #define PCI_EXT_CAP_ID_MPHY 0x0020
00170 #define PCI_EXT_CAP_ID_FRSQ 0x0021
00171 #define PCI_EXT_CAP_ID_RTR 0x0022
00173 #define GET_CAPABILITY_STR(cap_id) \
00174     (cap_id) == 0x00 ? "Null Capability" : \
00175     (cap_id) == PCI_CAP_ID_PM ? "Power Management" : \
00176     (cap_id) == PCI_CAP_ID_AGP ? "Accelerated Graphics Port" : \
00177     (cap_id) == PCI_CAP_ID_VPD ? "Vital Product Data" : \
00178     (cap_id) == PCI_CAP_ID_SLOTID ? "Slot Identification" : \
00179     (cap_id) == PCI_CAP_ID_MSI ? "Message Signalled Interrupts (MSI)" : \
00180     (cap_id) == PCI_CAP_ID_CHSWP ? "CompactPCI HotSwap" : \
00181     (cap_id) == PCI_CAP_ID_PCIX ? "PCI-X" : \
00182     (cap_id) == PCI_CAP_ID_HT ? "HyperTransport" : \
00183     (cap_id) == PCI_CAP_ID_VNDR ? "Vendor-Specific" : \
00184     (cap_id) == PCI_CAP_ID_DBG ? "Debug port" : \
00185     (cap_id) == PCI_CAP_ID_CCRC ? "CompactPCI Central Resource Control" : \
00186     (cap_id) == PCI_CAP_ID_SHPC ? "PCI Standard Hot-Plug Controller" : \
00187     (cap_id) == PCI_CAP_ID_SSVID ? "Bridge subsystem vendor/device ID" : \
00188     (cap_id) == PCI_CAP_ID_AGP3 ? "AGP Target PCI-PCI bridge" : \
00189     (cap_id) == PCI_CAP_ID_SECDEV ? "Secure Device" : \
00190     (cap_id) == PCI_CAP_ID_EXP ? "PCI Express" : \
00191     (cap_id) == PCI_CAP_ID_MSIX ? "Extended Message Signalled Interrupts (MSI-X)" : \
00192     (cap_id) == PCI_CAP_ID_SATA ? "SATA Data/Index Conf." : \
00193     (cap_id) == PCI_CAP_ID_AF ? "PCI Advanced Features" : \
00194     "Unknown"
00195
00196 #define GET_EXTENDED_CAPABILITY_STR(cap_id) \
00197     (cap_id) == 0x0000 ? "Null Capability" : \
00198     (cap_id) == PCI_EXT_CAP_ID_ERR ? "Advanced Error Reporting (AER)" : \
00199     (cap_id) == PCI_EXT_CAP_ID_VC ? "Virtual Channel (VC)" : \
00200     (cap_id) == PCI_EXT_CAP_ID_DSN ? "Device Serial Number" : \
00201     (cap_id) == PCI_EXT_CAP_ID_PWR ? "Power Budgeting" : \
00202     (cap_id) == PCI_EXT_CAP_ID_RCLD ? "Root Complex Link Declaration" : \
00203     (cap_id) == PCI_EXT_CAP_ID_RCILC ? "Root Complex Internal Link Control" : \
```

```
00204     (cap_id) == PCI_EXT_CAP_ID_RCEC ? "Root Complex Event Collector Endpoint Association" : \
00205     (cap_id) == PCI_EXT_CAP_ID_MFVC ? "Multi-Function Virtual Channel (MFVC)" : \
00206     (cap_id) == PCI_EXT_CAP_ID_VC9 ? "Virtual Channel (VC)" : \
00207     (cap_id) == PCI_EXT_CAP_ID_RCRB ? "Root Complex Register Block (RCRB) Header" : \
00208     (cap_id) == PCI_EXT_CAP_ID_VNDR ? "Vendor-Specific Extended Capability (VSEC)" : \
00209     (cap_id) == PCI_EXT_CAP_ID_CAC ? "Configuration Access Correlation (CAC)" : \
00210     (cap_id) == PCI_EXT_CAP_ID_ACS ? "Access Control Services (ACS)" : \
00211     (cap_id) == PCI_EXT_CAP_ID_ARI ? "Alternative Routing-ID Interpretation (ARI)" : \
00212     (cap_id) == PCI_EXT_CAP_ID_ATS ? "Address Translation Services (ATS)" : \
00213     (cap_id) == PCI_EXT_CAP_ID_SRIOV ? "Single Root I/O Virtualization (SR-IOV)" : \
00214     (cap_id) == PCI_EXT_CAP_ID_MRIOV ? "Multi-Root I/O Virtualization (MR-IOV)" : \
00215     (cap_id) == PCI_EXT_CAP_ID_MCAST ? "Multicast" : \
00216     (cap_id) == PCI_EXT_CAP_ID_PRI ? "Page Request" : \
00217     (cap_id) == PCI_EXT_CAP_ID_AMD_XXX ? "Reserved for AMD" : \
00218     (cap_id) == PCI_EXT_CAP_ID_REBAR ? "Resizable BAR" : \
00219     (cap_id) == PCI_EXT_CAP_ID_DPA ? "Dynamic Power Allocation (DPA)" : \
00220     (cap_id) == PCI_EXT_CAP_ID_TPH ? "TLP Processing Hints (TPH)" : \
00221     (cap_id) == PCI_EXT_CAP_ID_LTR ? "Latency Tolerance Reporting (LTR)" : \
00222     (cap_id) == PCI_EXT_CAP_ID_SECPCI ? "Secondary PCI Express" : \
00223     (cap_id) == PCI_EXT_CAP_ID_PMUX ? "Protocol Multiplexing (PMUX)" : \
00224     (cap_id) == PCI_EXT_CAP_ID_PASID ? "Process Address Space ID (PASID)" : \
00225     (cap_id) == PCI_EXT_CAP_ID_LNR ? "LN Requester (LNR)" : \
00226     (cap_id) == PCI_EXT_CAP_ID_L1PMS ? "Downstream Port Containment (DPC)" : \
00227     (cap_id) == PCI_EXT_CAP_ID_L1PMS ? "L1 PM Substates" : \
00228     (cap_id) == PCI_EXT_CAP_ID_PTMs ? "Precision Time Measurement (PTM)" : \
00229     (cap_id) == PCI_EXT_CAP_ID_MPHY ? "PCI Express over M-PHY (M-PCIe)" : \
00230     (cap_id) == PCI_EXT_CAP_ID_FRSQ ? "FRS Queueing" : \
00231     (cap_id) == PCI_EXT_CAP_ID_RTR ? "Readiness Time Reporting" : \
00232     "Unknown"
00233
00234 #define PCI_EXP_DEVCAP_PHANTOM_SHIFT 3
00235 #define PCI_STATUS_DEVSEL_SHIFT 9
00236 #define PCI_EXP_DEVCTL_READRQ_SHIFT 12
00237 #define PCI_EXP_SLTCAP_SPLV_SHIFT 7
00238 #define PCI_EXP_FLAGS_TYPE_SHIFT 9
00239 #define PCI_EXP_DEVCAP_L1_SHIFT 9
00240 #define PCI_EXP_DEVCAP_PWD_SCL_SHIFT 26
00241 #define PCI_EXP_DEVCAP_PWR_VAL_SHIFT 18
00242 #define PCI_EXP_DEVCTL_PAYLOAD_SHIFT 5
00243 #define PCI_EXP_LNKCAP_MLW_SHIFT 4
00244 #define PCI_EXP_LNKCAP_ASPPMS_SHIFT 10
00245 #define PCI_EXP_LNKCAP_LOSEL_SHIFT 12
00246 #define PCI_EXP_LNKCAP_LIEL_SHIFT 15
00247 #define PCI_EXP_SLTCAP_SPLS_SHIFT 15
00248 #define PCI_EXP_SLTCAP_AIC_SHIFT 6
00249 #define PCI_EXP_SLTCTL_PIC_SHIFT 8
00250 #define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT 22
00251 #define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT 22
00252
00253 #define PCI_COMMAND 0x04
00254 #define PCI_COMMAND_IO 0x1
00255 #define PCI_COMMAND_MEMORY 0x2
00256 #define PCI_COMMAND_MASTER 0x4
00257 #define PCI_COMMAND_SPECIAL 0x8
00258 #define PCI_COMMAND_INVALIDATE 0x10
00259 #define PCI_COMMAND_VGA_PALETTE 0x20
00260 #define PCI_COMMAND_PARITY 0x40
00261 #define PCI_COMMAND_WAIT 0x80
00262 #define PCI_COMMAND_SERR 0x100
00263 #define PCI_COMMAND_FAST_BACK 0x200
00264 #define PCI_COMMAND_INTX_DISABLE 0x400
00265 #define PCI_STATUS 0x06
00266 #define PCI_STATUS_INTERRUPT 0x08
00267 #define PCI_STATUS_CAP_LIST 0x10
00268 #define PCI_STATUS_66MHZ 0x20
00269 #define PCI_STATUS_UDF 0x40
00270 #define PCI_STATUS_FAST_BACK 0x80
00271 #define PCI_STATUS_PARITY 0x100
00272 #define PCI_STATUS_DEVSEL_MASK 0x600
00273 #define PCI_STATUS_DEVSEL_FAST 0x000
00274 #define PCI_STATUS_DEVSEL_MEDIUM 0x200
00275 #define PCI_STATUS_DEVSEL_SLOW 0x400
00276 #define PCI_STATUS_SIG_TARGET_ABORT 0x800
00277 #define PCI_STATUS_REC_TARGET_ABORT 0x1000
00278 #define PCI_STATUS_REC_MASTER_ABORT 0x2000
00279 #define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000
00280 #define PCI_STATUS_DETECTED_PARITY 0x8000
00281 #
00282 /* PCI Express capability registers */
00283
00284 #define PCI_EXP_FLAGS 2
00285 #define PCI_EXP_FLAGS_VERS 0x000f
00286 #define PCI_EXP_FLAGS_TYPE 0x00f0
00287 #define PCI_EXP_TYPE_ENDPOINT 0x0
00288 #define PCI_EXP_TYPE_LEG_END 0x1
00289 #define PCI_EXP_TYPE_ROOT_PORT 0x4
00290 #define PCI_EXP_TYPE_UPSTREAM 0x5
```

```
00294 #define PCI_EXP_TYPE_DOWNSTREAM 0x6
00295 #define PCI_EXP_TYPE_PCI_BRIDGE 0x7
00296 #define PCI_EXP_TYPE_PCIE_BRIDGE 0x8
00297 #define PCI_EXP_TYPE_RC_END 0x9
00298 #define PCI_EXP_TYPE_RC_EC 0xa
00299 #define PCI_EXP_FLAGS_SLOT 0x0100
00300 #define PCI_EXP_FLAGS_IRQ 0x3e00
00301 #define PCI_EXP_DEVCAP 4
00302 #define PCI_EXP_DEVCAP_PAYLOAD 0x00000007
00303 #define PCI_EXP_DEVCAP_PHANTOM 0x00000018
00304 #define PCI_EXP_DEVCAP_EXT_TAG 0x00000020
00305 #define PCI_EXP_DEVCAP_L0S 0x000001c0
00306 #define PCI_EXP_DEVCAP_L1 0x00000e00
00307 #define PCI_EXP_DEVCAP_ATN_BUT 0x00001000
00308 #define PCI_EXP_DEVCAP_ATN_IND 0x00002000
00309 #define PCI_EXP_DEVCAP_PWR_IND 0x00004000
00310 #define PCI_EXP_DEVCAP_RBER 0x00008000
00311 #define PCI_EXP_DEVCAP_PWR_VAL 0x3fc0000
00312 #define PCI_EXP_DEVCAP_PWR_SCL 0x0c000000
00313 #define PCI_EXP_DEVCAP_FLR 0x10000000
00314 #define PCI_EXP_DEVCTL 8
00315 #define PCI_EXP_DEVCTL_CERE 0x0001
00316 #define PCI_EXP_DEVCTL_NFERE 0x0002
00317 #define PCI_EXP_DEVCTL_FERE 0x0004
00318 #define PCI_EXP_DEVCTL_URRE 0x0008
00319 #define PCI_EXP_DEVCTL_RELAX_EN 0x0010
00320 #define PCI_EXP_DEVCTL_PAYLOAD 0x0e0
00321 #define PCI_EXP_DEVCTL_EXT_TAG 0x0100
00322 #define PCI_EXP_DEVCTL_PHANTOM 0x0200
00323 #define PCI_EXP_DEVCTL_AUX_PME 0x0400
00324 #define PCI_EXP_DEVCTL_NOSNOOP_EN 0x0800
00325 #define PCI_EXP_DEVCTL_READRQ 0x7000
00326 #define PCI_EXP_DEVCTL_READRQ_128B 0x0000
00327 #define PCI_EXP_DEVCTL_READRQ_256B 0x1000
00328 #define PCI_EXP_DEVCTL_READRQ_512B 0x2000
00329 #define PCI_EXP_DEVCTL_READRQ_1024B 0x3000
00330 #define PCI_EXP_DEVCTL_BCR_FLR 0x8000
00331 #define PCI_EXP_DEVSTA 10
00332 #define PCI_EXP_DEVSTA_CED 0x0001
00333 #define PCI_EXP_DEVSTA_NFED 0x0002
00334 #define PCI_EXP_DEVSTA_FED 0x0004
00335 #define PCI_EXP_DEVSTA_URD 0x0008
00336 #define PCI_EXP_DEVSTA_AUXPD 0x0010
00337 #define PCI_EXP_DEVSTA_TRPND 0x0020
00338 #define PCI_EXP_LNKCAP 12
00339 #define PCI_EXP_LNKCAP_SLS 0x0000000f
00340 #define PCI_EXP_LNKCAP_SLS_2_5GB 0x00000001
00341 #define PCI_EXP_LNKCAP_SLS_5_0GB 0x00000002
00342 #define PCI_EXP_LNKCAP_MLW 0x000003f0
00343 #define PCI_EXP_LNKCAP_ASPLMS 0x00000c00
00344 #define PCI_EXP_LNKCAP_L0SEL 0x00007000
00345 #define PCI_EXP_LNKCAP_L1EL 0x00038000
00346 #define PCI_EXP_LNKCAP_CLKPM 0x00040000
00347 #define PCI_EXP_LNKCAP_SDERC 0x00080000
00349 #define PCI_EXP_LNKCAP_DLLLARC 0x00100000
00351 #define PCI_EXP_LNKCAP_LBNC 0x00200000
00353 #define PCI_EXP_LNKCAP_PN 0xff000000
00354 #define PCI_EXP_LNKCTL 16
00355 #define PCI_EXP_LNKCTL_ASPMC 0x0003
00356 #define PCI_EXP_LNKCTL_ASPM_LOS 0x0001
00357 #define PCI_EXP_LNKCTL_ASPM_L1 0x0002
00358 #define PCI_EXP_LNKCTL_RCB 0x0008
00359 #define PCI_EXP_LNKCTL_LD 0x0010
00360 #define PCI_EXP_LNKCTL_RL 0x0020
00361 #define PCI_EXP_LNKCTL_CCC 0x0040
00362 #define PCI_EXP_LNKCTL_ES 0x0080
00363 #define PCI_EXP_LNKCTL_CLKREQ_EN 0x0100
00364 #define PCI_EXP_LNKCTL_HAWD 0x0200
00365 #define PCI_EXP_LNKCTL_LBMIE 0x0400
00367 #define PCI_EXP_LNKCTL_LABIE 0x0800
00369 #define PCI_EXP_LNKSTA 18
00370 #define PCI_EXP_LNKSTA_CLS 0x000f
00371 #define PCI_EXP_LNKSTA_CLS_2_5GB 0x0001
00372 #define PCI_EXP_LNKSTA_CLS_5_0GB 0x0002
00373 #define PCI_EXP_LNKSTA_CLS_8_0GB 0x0003
00374 #define PCI_EXP_LNKSTA_NLW 0x03f0
00375 #define PCI_EXP_LNKSTA_NLW_X1 0x0010
00376 #define PCI_EXP_LNKSTA_NLW_X2 0x0020
00377 #define PCI_EXP_LNKSTA_NLW_X4 0x0040
00378 #define PCI_EXP_LNKSTA_NLW_X8 0x0080
00379 #define PCI_EXP_LNKSTA_NLW_SHIFT 4
00380 #define PCI_EXP_LNKSTA_LT 0x0800
00381 #define PCI_EXP_LNKSTA_SLC 0x1000
00382 #define PCI_EXP_LNKSTA_DLLLA 0x2000
00383 #define PCI_EXP_LNKSTA_LBMS 0x4000
00384 #define PCI_EXP_LNKSTA_LABS 0x8000
00385 #define PCI_CAP_EXP_ENDPOINT_SIZEOF_V1 20
```

```
00386 #define PCI_EXP_SLTCAP      20
00387 #define PCI_EXP_SLTCAP_ABP 0x00000001
00388 #define PCI_EXP_SLTCAP_PCP 0x00000002
00389 #define PCI_EXP_SLTCAP_MRLSP 0x00000004
00390 #define PCI_EXP_SLTCAP_AIP 0x00000008
00391 #define PCI_EXP_SLTCAP_PIP 0x00000010
00392 #define PCI_EXP_SLTCAP_HPS 0x00000020
00393 #define PCI_EXP_SLTCAP_HPC 0x00000040
00394 #define PCI_EXP_SLTCAP_SPLV 0x00007f80
00395 #define PCI_EXP_SLTCAP SPLS 0x00018000
00396 #define PCI_EXP_SLTCAP_EIP 0x00020000
00397 #define PCI_EXP_SLTCAP_NCCS 0x00040000
00398 #define PCI_EXP_SLTCAP_PSN 0xffff8000
00399 #define PCI_EXP_SLTCCTL 24
00400 #define PCI_EXP_SLTCCTL_ABPE 0x0001
00401 #define PCI_EXP_SLTCCTL_PFDE 0x0002
00402 #define PCI_EXP_SLTCCTL_MRLSCE 0x0004
00403 #define PCI_EXP_SLTCCTL_PDCE 0x0008
00404 #define PCI_EXP_SLTCCTL_CCIE 0x0010
00405 #define PCI_EXP_SLTCCTL_HPIE 0x0020
00406 #define PCI_EXP_SLTCCTL_AIC 0x00c0
00407 #define PCI_EXP_SLTCCTL_ATTN_IND_ON 0x0040
00408 #define PCI_EXP_SLTCCTL_ATTN_IND_BLINK 0x0080
00409 #define PCI_EXP_SLTCCTL_ATTN_IND_OFF 0x00c0
00410 #define PCI_EXP_SLTCCTL_PIC 0x0300
00411 #define PCI_EXP_SLTCCTL_PWR_IND_ON 0x0100
00412 #define PCI_EXP_SLTCCTL_PWR_IND_BLINK 0x0200
00413 #define PCI_EXP_SLTCCTL_PWR_IND_OFF 0x0300
00414 #define PCI_EXP_SLTCCTL_PCC 0x0400
00415 #define PCI_EXP_SLTCCTL_PWR_ON 0x0000
00416 #define PCI_EXP_SLTCCTL_PWR_OFF 0x0400
00417 #define PCI_EXP_SLTCCTL_EIC 0x0800
00418 #define PCI_EXP_SLTCCTL_DLLSCE 0x1000
00420 #define PCI_EXP_SLTSTA 26
00421 #define PCI_EXP_SLTSTA_ABP 0x0001
00422 #define PCI_EXP_SLTSTA_PFD 0x0002
00423 #define PCI_EXP_SLTSTA_MRLSC 0x0004
00424 #define PCI_EXP_SLTSTA_FDC 0x0008
00425 #define PCI_EXP_SLTSTA_CC 0x0010
00426 #define PCI_EXP_SLTSTA_MRLSS 0x0020
00427 #define PCI_EXP_SLTSTA_PDS 0x0040
00428 #define PCI_EXP_SLTSTA_EIS 0x0080
00429 #define PCI_EXP_SLTSTA_DLLSC 0x0100
00430 #define PCI_EXP_RTCTL 28
00431 #define PCI_EXP_RTCTL_SECEE 0x0001
00432 #define PCI_EXP_RTCTL_SENFEE 0x0002
00433 #define PCI_EXP_RTCTL_SEFEE 0x0004
00434 #define PCI_EXP_RTCTL_PMEIE 0x0008
00435 #define PCI_EXP_RTCTL_CRSSVE 0x0010
00436 #define PCI_EXP_RTCAP 30
00437 #define PCI_EXP_RTCAP_CRSVIS 0x0001
00438 #define PCI_EXP_RTSTA 32
00439 #define PCI_EXP_RTSTA_PME 0x00010000
00440 #define PCI_EXP_RTSTA_PENDING 0x00020000
00442 #define PCI_EXP_DEVCAP2 36
00443 #define PCI_EXP_DEVCAP2_RANGE_A 0x1
00444 #define PCI_EXP_DEVCAP2_RANGE_B 0x2
00445 #define PCI_EXP_DEVCAP2_RANGE_C 0x4
00446 #define PCI_EXP_DEVCAP2_RANGE_D 0x8
00447 #define PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP 0xF
00449 #define PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP 0x000010
00451 #define PCI_EXP_DEVCAP2_ARI 0x00000020
00452 #define PCI_EXP_DEVCAP2_ATOMIC_ROUTE 0x00000040
00453 #define PCI_EXP_DEVCAP2_ATOMIC_COMP32 0x000080
00455 #define PCI_EXP_DEVCAP2_ATOMIC_COMP64 0x00000100
00456 #define PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP 0x000200
00458 #define PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR 0x000400
00460 #define PCI_EXP_DEVCAP2_LTR 0x00000800
00461 #define PCI_EXP_DEVCAP2_TPH_COMP_SUPP 0x001000
00463 #define PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP 0x002000
00465 #define PCI_EXP_DEVCAP2_OBFF_MASK 0x000c0000
00466 #define PCI_EXP_DEVCAP2_OBFF_MSG 0x00040000
00467 #define PCI_EXP_DEVCAP2_OBFF_WAKE 0x00080000
00468 #define PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP 0x100000
00470 #define PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP 0x200000
00472 #define PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES 0xC00000
00475 #define PCI_EXP_DEVCTL2 40
00476 #define PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE 0x0010
00478 #define PCI_EXP_DEVCTL2_COMP_TIMEOUT 0x000f
00479 #define PCI_EXP_DEVCTL2_ARI 0x0020
00480 #define PCI_EXP_DEVCTL2_ATOMIC_REQ 0x0040
00481 #define PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK 0x0080
00482 #define PCI_EXP_DEVCTL2_IDO_REQ_EN 0x0100
00483 #define PCI_EXP_DEVCTL2_IDO_CMP_EN 0x0200
00484 #define PCI_EXP_DEVCTL2_LTR_EN 0x0400
00485 #define PCI_EXP_DEVCTL2_OBFF_MSGA_EN 0x2000
00486 #define PCI_EXP_DEVCTL2_OBFF_MSGB_EN 0x4000
```

```

00487 #define PCI_EXP_DEVCTL2_OBFF_WAKE_EN 0x6000
00488 #define PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK 0x8000
00490 #define PCI_EXP_DEVSTA2 42
00491 #define PCI_CAP_EXP_ENDPOINT_SIZEOF_V2 44
00492 #define PCI_EXP_LNKCAP2 44
00493 #define PCI_EXP_LNKCAP2_SLS_2_5GB 0x00000002
00494 #define PCI_EXP_LNKCAP2_SLS_5_0GB 0x00000004
00495 #define PCI_EXP_LNKCAP2_SLS_8_0GB 0x00000008
00496 #define PCI_EXP_LNKCAP2_CROSSLINK 0x00000100
00497 #define PCI_EXP_LNKCTL2 48
00498 #define PCI_EXP_LNKCTL2_LNK_SPEED_2_5 0x00000001
00499 #define PCI_EXP_LNKCTL2_LNK_SPEED_5_0 0x00000002
00500 #define PCI_EXP_LNKCTL2_LNK_SPEED_8_0 0x00000003
00501 #define PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK 0x0000000f
00503 #define PCI_EXP_LNKCTL2_ENTER_COMP 0x00000010
00504 #define PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS 0x00000020
00506 #define PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH 0x00000040
00508 #define PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK 0x00000380
00509 #define PCI_EXP_LNKCTL2_ENTER_MOD_COMP 0x00000400
00511 #define PCI_EXP_LNKCTL2_COMP_SOS 0x00000800
00512 #define PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL 0x00001000
00514 #define PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL 0x0000f000
00516 #define PCI_EXP_LNKSTA2 50
00517 #define PCI_EXP_LNKSTA2_CDL 0x00000001
00519 #define PCI_EXP_LNKSTA2_EQUALIZ_COMP 0x00000002
00520 #define PCI_EXP_LNKSTA2_EQUALIZ_PH1 0x00000004
00521 #define PCI_EXP_LNKSTA2_EQUALIZ_PH2 0x00000008
00522 #define PCI_EXP_LNKSTA2_EQUALIZ_PH3 0x00000010
00523 #define PCI_EXP_LNKSTA2_LINE_EQ_REQ 0x00000020
00524 #define PCI_EXP_SLTCAP2 52
00525 #define PCI_EXP_SLTCTL2 56
00526 #define PCI_EXP_SLTSTA2 58
00528 #endif /* _PCI_REGS_H_ */
00529

```

pci_strings.h File Reference

```
#include "wdc_defs.h"
```

Functions

- **DWORD [DLLCALLCONV PciConfRegData2Str](#) ([_In_ WDC_DEVICE_HANDLE hDev](#), [_In_ DWORD dwOffset](#), [_Outptr_ PCHAR pBuf](#), [_In_ DWORD dwInLen](#), [_Outptr_ DWORD *pdwOutLen](#))**
Reads data from a PCI device's configuration space and parses the data to a string, if such a parsing is available.
- **DWORD [DLLCALLCONV PciExpressConfRegData2Str](#) ([_In_ WDC_DEVICE_HANDLE hDev](#), [_In_ DWORD dwOffset](#), [_Outptr_ PCHAR pBuf](#), [_In_ DWORD dwInLen](#), [_Outptr_ DWORD *pdwOutLen](#))**
Reads data from a PCI Express device's extended configuration space and parses the data to a string, if such a parsing is available.

Function Documentation

PciConfRegData2Str()

```
DWORD DLLCALLCONV PciConfRegData2Str (
    \_In\_ WDC\_DEVICE\_HANDLE hDev,
    \_In\_ DWORD dwOffset,
    \_Outptr\_ PCHAR pBuf,
    \_In\_ DWORD dwInLen,
    \_Outptr\_ DWORD \* pdwOutLen )
```

Reads data from a PCI device's configuration space and parses the data to a string, if such a parsing is available.

The parsing is based upon information obtained from the official PCI Specification, 3rd Generation.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	Offset of a register in the configuration space to be read and parsed to a string, if a parsing is available.
out	<i>pBuf</i>	A pointer to a user preallocated buffer to be filled by the function. The buffer will not be filled if no parsing is available.
in	<i>dwInLen</i>	Size of the user preallocated buffer.
out	<i>pdwOutLen</i>	A pointer to a DWORD which will hold the actual length of the string the function wrote to the buffer. If the buffer is too small for the text written by the function then the text will be truncated. We recommend preallocating a 1024 byte buffer to avoid any truncation.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

PciExpressConfRegData2Str()

```
DWORD DLLCALLCONV PciExpressConfRegData2Str (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _Outptr_ PCHAR pBuf,
    _In_ DWORD dwInLen,
    _Outptr_ DWORD * pdwOutLen )
```

Reads data from a PCI Express device's extended configuration space and parses the data to a string, if such a parsing is available.

The parsing is based upon information obtained from the official PCI Specification, 3rd Generation.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	Offset of a register in the configuration space to be read and parsed to a string, if a parsing is available. Offset should be the sum of the value obtained from WDC_PciGetExpressOffset() WDC_PciGetExpressOffset() and the desired PCI Express offset.
out	<i>pBuf</i>	A pointer to a user preallocated buffer to be filled by the function. The buffer will not be filled if no parsing is available.
in	<i>dwInLen</i>	Size of the user preallocated buffer.
out	<i>pdwOutLen</i>	A pointer to a DWORD which will hold the actual length of the string the function wrote to the buffer. If the buffer is too small for the text written by the function then the text will be truncated. We recommend preallocating a 1024 byte buffer to avoid any truncation.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

pci_strings.h

[Go to the documentation of this file.](#)

```
00001 #ifndef _PCI_STRINGS_H_
00002 #define _PCI_STRINGS_H_
```

```
00003
00004 #include "wdc_def.h"
00005
00006 #ifdef __cplusplus
00007 extern "C" {
00008 #endif
00009
00036 DWORD DLLCALLCONV PciConfRegData2Str(_In_ WDC_DEVICE_HANDLE hDev,
00037     _In_ DWORD dwOffset, _Outptr_ PCHAR pBuf, _In_ DWORD dwInLen,
00038     _Outptr_ DWORD *pdwOutLen);
00039
00071 DWORD DLLCALLCONV PciExpressConfRegData2Str(_In_ WDC_DEVICE_HANDLE hDev,
00072     _In_ DWORD dwOffset, _Outptr_ PCHAR pBuf, _In_ DWORD dwInLen,
00073     _Outptr_ DWORD *pdwOutLen);
00074 #ifdef __cplusplus
00075 }
00076 #endif
00077
00078 #endif /* _PCI_STRINGS_ */
```

status_strings.h File Reference

```
#include "windrvr.h"
```

Functions

- const char *DLLCALLCONV Stat2Str (_In_ DWORD dwStatus)
Retrieves the status string that corresponds to a status code.

Function Documentation

Stat2Str()

```
const char *DLLCALLCONV Stat2Str (
    _In_ DWORD dwStatus )
```

Retrieves the status string that corresponds to a status code.

Parameters

in	<i>dwStatus</i>	A numeric status code
----	-----------------	-----------------------

Returns

Returns the verbal status description (string) that corresponds to the specified numeric status code.

status_strings.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _STATUS_STRINGS_H_
00004 #define _STATUS_STRINGS_H_
00005
00006 #if defined(__cplusplus)
00007 extern "C" {
00008 #endif
00009
00010 #include "windrvr.h"
00011
```

```
00022 const char * DLLCALLCONV Stat2Str(_In_ DWORD dwStatus);  
00023  
00024 #ifdef __cplusplus  
00025 }  
00026 #endif  
00027  
00028 #endif /* _STATUS_STRINGS_H */  
00029
```

utils.h File Reference

```
#include <stdio.h>  
#include "windrvr.h"
```

Macros

- #define MAX_PATH 4096
- #define snprintf _snprintf
- #define vsnprintf _vsnprintf
- #define OsMemoryBarrier() MemoryBarrier()
- #define INFINITE 0xffffffff

Typedefs

- typedef void(DLLCALLCONV * HANDLER_FUNC) (void *pData)

Functions

- DWORD DLLCALLCONV OsEventCreate (_Outptr_ HANDLE *phOsEvent)
Creates an event object.
- void DLLCALLCONV OsEventClose (_In_ HANDLE hOsEvent)
Closes a handle to an event object.
- DWORD DLLCALLCONV OsEventWait (_In_ HANDLE hOsEvent, _In_ DWORD dwSecTimeout)
Waits until a specified event object is in the signaled state or the time-out interval elapses.
- DWORD DLLCALLCONV OsEventSignal (_In_ HANDLE hOsEvent)
Sets the specified event object to the signaled state.
- DWORD DLLCALLCONV OsEventReset (_In_ HANDLE hOsEvent)
Resets the specified event object to the non-signaled state.
- DWORD DLLCALLCONV OsMutexCreate (_Outptr_ HANDLE *phOsMutex)
Creates a mutex object.
- void DLLCALLCONV OsMutexClose (_In_ HANDLE hOsMutex)
Closes a handle to a mutex object.
- DWORD DLLCALLCONV OsMutexLock (_In_ HANDLE hOsMutex)
Locks the specified mutex object.
- DWORD DLLCALLCONV OsMutexUnlock (_In_ HANDLE hOsMutex)
Releases (unlocks) a locked mutex object.
- void DLLCALLCONV SleepWrapper (_In_ DWORD dwMicroSecs)
Wrapper to WD_Sleep, Sleeps dwMicroSecs microseconds.
- int print2wstr (wchar_t *buffer, size_t count, const wchar_t *format,...)
- void DLLCALLCONV vPrintDbgMessage (_In_ DWORD dwLevel, _In_ DWORD dwSection, _In_ const char *format, _In_ va_list ap)
Sends debug messages to the Debug Monitor.

- void **DLLCALLCONV PrintDbgMessage** (DWORD dwLevel, DWORD dwSection, const char *format,...)
Sends debug messages to the Debug Monitor.
- int **DLLCALLCONV GetPageSize** (void)
Returns the page size in the OS.
- int **DLLCALLCONV GetNumberOfProcessors** (void)
Returns the number of processors currently online (available)
- BOOL **DLLCALLCONV UtilGetFileSize** (_In_ const PCHAR sFileName, _Outptr_ DWORD *pdwFileSize, _In_ PCHAR sErrString)
Writes the file size with name sFileName in dwFileSize.
- DWORD **DLLCALLCONV UtilGetStringFromUser** (_Out_ PCHAR pcString, _In_ DWORD dwSizeStr, _In_ const CHAR *pcInputText, _In_ const CHAR *pcDefaultString)
Gets a string from user input and out it in pcString.
- DWORD **DLLCALLCONV UtilGetFileName** (_Out_ PCHAR pcFileName, _In_ DWORD dwFileNameSize, _In_ const CHAR *pcDefaultFileName)
Gets a file name from user input and out it in pcFileName.

Macro Definition Documentation

INFINITE

```
#define INFINITE 0xffffffff
```

Definition at line 324 of file [utils.h](#).

MAX_PATH

```
#define MAX_PATH 4096
```

Definition at line 19 of file [utils.h](#).

OsMemoryBarrier

```
#define OsMemoryBarrier( ) MemoryBarrier()
```

Definition at line 190 of file [utils.h](#).

snprintf

```
#define snprintf _snprintf
```

Definition at line 23 of file [utils.h](#).

vsnprintf

```
#define vsnprintf _vsnprintf
```

Definition at line 25 of file [utils.h](#).

Typedef Documentation

HANDLER_FUNC

```
typedef void (DCALLCONV * HANDLER_FUNC) (void *pData)
```

Definition at line 29 of file [utils.h](#).

Function Documentation

GetNumberOfProcessors()

```
int DLLCALLCONV GetNumberOfProcessors (
    void )
```

Returns the number of processors currently online (available)

Returns

Number of processors currently online (available)

GetPageSize()

```
int DLLCALLCONV GetPageSize (
    void )
```

Returns the page size in the OS.

Returns

Page size in OS

OsEventClose()

```
void DLLCALLCONV OsEventClose (
    _In_ HANDLE hOsEvent )
```

Closes a handle to an event object.

Parameters

in	<i>hOsEvent</i>	The handle to the event object to be closed
----	-----------------	---

Returns

None

OsEventCreate()

```
DWORD DLLCALLCONV OsEventCreate (
    _Outptr_ HANDLE * phOsEvent )
```

Creates an event object.

Parameters

out	<i>phOsEvent</i>	The pointer to a variable that receives a handle to the newly created event object
------------	------------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsEventReset()

```
DWORD DLLCALLCONV OsEventReset (
    _In_ HANDLE hOsEvent )
```

Resets the specified event object to the non-signaled state.

Parameters

in	<i>hOsEvent</i>	The handle to the event object
-----------	-----------------	--------------------------------

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsEventSignal()

```
DWORD DLLCALLCONV OsEventSignal (
    _In_ HANDLE hOsEvent )
```

Sets the specified event object to the signaled state.

Parameters

in	<i>hOsEvent</i>	The handle to the event object
-----------	-----------------	--------------------------------

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsEventWait()

```
DWORD DLLCALLCONV OsEventWait (
    _In_ HANDLE hOsEvent,
    _In_ DWORD dwSecTimeout )
```

Waits until a specified event object is in the signaled state or the time-out interval elapses.

Parameters

in	<i>hOsEvent</i>	The handle to the event object
in	<i>dwSecTimeout</i>	Time-out interval of the event, in seconds. For an infinite wait, set the timeout to INFINITE.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsMutexClose()

```
void DLLCALLCONV OsMutexClose (
    _In_ HANDLE hOsMutex )
```

Closes a handle to a mutex object.

Parameters

in	<i>hOsMutex</i>	The handle to the mutex object to be closed
----	-----------------	---

Returns

None

OsMutexCreate()

```
DWORD DLLCALLCONV OsMutexCreate (
    _Outptr_ HANDLE * phOsMutex )
```

Creates a mutex object.

Parameters

out	<i>phOsMutex</i>	The pointer to a variable that receives a handle to the newly created mutex object
-----	------------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsMutexLock()

```
DWORD DLLCALLCONV OsMutexLock (
```

In HANDLE hOsMutex)

Locks the specified mutex object.

Parameters

in	hOsMutex	The handle to the mutex object to be locked
----	----------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

OsMutexUnlock()

```
DWORD DLLCALLCONV OsMutexUnlock (
    In HANDLE hOsMutex )
```

Releases (unlocks) a locked mutex object.

Parameters

in	hOsMutex	The handle to the mutex object to be unlocked
----	----------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

print2wstr()

```
int print2wstr (
    wchar_t * buffer,
    size_t count,
    const wchar_t * format,
    ...
)
```

PrintDbgMessage()

```
void DLLCALLCONV PrintDbgMessage (
    DWORD dwLevel,
    DWORD dwSection,
    const char * format,
    ...
)
```

Sends debug messages to the Debug Monitor.

Parameters

in	<i>dwLevel</i>	Assigns the level in the Debug Monitor, in which the data will be declared. If zero, D_ERROR will be declared. For more details please refer to DEBUG_LEVEL in windrvr.h .
in	<i>dwSection</i>	Assigns the section in the Debug Monitor, in which the data will be declared. If zero, S_MISC will be declared. For more details please refer to DEBUG_SECTION in windrvr.h .
in	<i>format</i>	Format-control string

Parameters

in	...	Optional arguments, limited to 256 bytes
----	-----	--

Returns

None

SleepWrapper()

```
void DLLCALLCONV SleepWrapper (
    _In_ DWORD dwMicroSecs )
```

Wrapper to WD_Sleep, Sleeps dwMicroSecs microseconds.

Parameters

in	<i>dwMicroSecs</i>	Time in microseconds to sleep
----	--------------------	-------------------------------

Returns

None

UtilGetFileName()

```
DWORD DLLCALLCONV UtilGetFileName (
    _Out_ PCHAR pcFileName,
    _In_ DWORD dwFileNameSize,
    _In_ const CHAR * pcDefaultFileName )
```

Gets a file name from user input and out it in pcFileName.

Parameters

out	<i>pcFileName</i>	Pointer to buffer that will be filled with user input
in	<i>dwFileNameSize</i>	Number of bytes in file name
in	<i>pcDefaultFileName</i>	Default file name to write to pcFileName in case of no input

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

UtilGetFileSize()

```
BOOL DLLCALLCONV UtilGetFileSize (
    _In_ const PCHAR sFileName,
    _Outptr_ DWORD * pdwFileSize,
    _In_ PCHAR sErrString )
```

Writes the file size with name sFileName in dwFileSize.

Parameters

in	<i>sFileName</i>	Name of the file
out	<i>pdwFileSize</i>	Pointer to DWORD that will be written with the file's size.
in	<i>sErrString</i>	Optional error message

Returns

TURE if the function succeeded in getting the file size, else FALSE

UtilGetStringFromUser()

```
DWORD DLLCALLCONV UtilGetStringFromUser (
    _Out_ PCHAR pcString,
    _In_ DWORD dwSizeStr,
    _In_ const CHAR * pcInputText,
    _In_ const CHAR * pcDefaultString )
```

Gets a string from user input and out it in pcString.

Parameters

out	<i>pcString</i>	Pointer to buffer that will be filled with user input
in	<i>dwSizeStr</i>	Number of bytes to read from user
in	<i>pcInputText</i>	Input text that will be written to stdout on prompt
in	<i>pcDefaultString</i>	Default String to write to pcString in case of no input

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

vPrintDbgMessage()

```
void DLLCALLCONV vPrintDbgMessage (
    _In_ DWORD dwLevel,
    _In_ DWORD dwSection,
    _In_ const char * format,
    _In_ va_list ap )
```

Sends debug messages to the Debug Monitor.

Parameters

in	<i>dwLevel</i>	Assigns the level in the Debug Monitor, in which the data will be declared. If zero, D_ERROR will be declared. For more details please refer to DEBUG_LEVEL in windrvr.h .
in	<i>dwSection</i>	Assigns the section in the Debug Monitor, in which the data will be declared. If zero, S_MISC will be declared. For more details please refer to DEBUG_SECTION in windrvr.h .
in	<i>format</i>	Format-control string
in	<i>ap</i>	Optional Arguments

Returns

None

utils.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WD_UTILS_H_
00004 #define _WD_UTILS_H_
00005
00006 #if defined(__KERNEL__)
00007     #include "kpstdlib.h"
00008 #else
00009     #include <stdio.h>
00010 #endif
00011
00012 #include "windrvr.h"
00013
00014 #if defined(__cplusplus)
00015 extern "C" {
00016 #endif
00017
00018 #if !defined(MAX_PATH)
00019     #define MAX_PATH 4096
00020 #endif
00021
00022 #if defined(WIN32)
00023     #define snprintf _snprintf
00024     #if !defined(vsnprintf)
00025         #define vsnprintf _vsnprintf
00026     #endif
00027 #endif
00028
00029 typedef void (DCALLCONV *HANDLER_FUNC)(void *pData);
00030
00031 #if !defined(WIN32) || defined(_MT)
00032
00048 DWORD DLLCALLCONV ThreadStart(_Outptr_ HANDLE *phThread,
00049     _In_ HANDLER_FUNC pFunc,
00050     _In_ void *pData);
00051
00061 void DLLCALLCONV ThreadWait(_In_ HANDLE hThread);
00062 #endif
00063
00075 DWORD DLLCALLCONV OsEventCreate(_Outptr_ HANDLE *phOsEvent);
00076
00086 void DLLCALLCONV OsEventClose(_In_ HANDLE hOsEvent);
00087
00102 DWORD DLLCALLCONV OsEventWait(_In_ HANDLE hOsEvent, _In_ DWORD dwSecTimeout);
00103
00114 DWORD DLLCALLCONV OsEventSignal(_In_ HANDLE hOsEvent);
00115
00126 DWORD DLLCALLCONV OsEventReset(_In_ HANDLE hOsEvent);
00127
00139 DWORD DLLCALLCONV OsMutexCreate(_Outptr_ HANDLE *phOsMutex);
00140
00150 void DLLCALLCONV OsMutexClose(_In_ HANDLE hOsMutex);
00151
00162 DWORD DLLCALLCONV OsMutexLock(_In_ HANDLE hOsMutex);
00163
00174 DWORD DLLCALLCONV OsMutexUnlock(_In_ HANDLE hOsMutex);
00175
00185 void DLLCALLCONV SleepWrapper(_In_ DWORD dwMicroSecs);
00186
00187 #if defined(UNIX)
00188     #define OsMemoryBarrier() __sync_synchronize()
00189 #elif defined(WIN32)
00190     #define OsMemoryBarrier() MemoryBarrier()
00191 #endif
00192
00193 #if !defined(__KERNEL__)
00194
00195
00196 int print2wstr(wchar_t *buffer, size_t count, const wchar_t *format, ...);
00197
00217 void DLLCALLCONV vPrintDbgMessage(_In_ DWORD dwLevel, _In_ DWORD dwSection,
00218     _In_ const char *format, _In_ va_list ap);
00219
00239 void DLLCALLCONV PrintDbgMessage(DWORD dwLevel, DWORD dwSection,
00240     const char *format, ...);
00241
```

```

00249 int DLLCALLCONV GetPageSize(void);
00250
00258 int DLLCALLCONV GetNumberOfProcessors(void);
00259
00272 BOOL DLLCALLCONV UtilGetFileSize(_In_ const PCHAR sFileName,
00273     _Outptr_ DWORD *pdwFileSize, _In_ PCHAR sErrString);
00274
00291 DWORD DLLCALLCONV UtilGetStringFromUser(_Out_ PCHAR pcString,
00292     _In_ DWORD dwSizeStr, _In_ const CHAR *pcInputText,
00293     _In_ const CHAR *pcDefaultString);
00294
00295
00310 DWORD DLLCALLCONV UtilGetFileName(_Out_ PCHAR pcFileName,
00311     _In_ DWORD dwFileNameSize, _In_ const CHAR *pcDefaultFileName);
00312 #endif
00313
00314 #if defined(UNIX)
00315     #if !defined(stricmp)
00316         #define stricmp strcasecmp
00317     #endif
00318     #if !defined(strnicmp)
00319         #define strnicmp strncasecmp
00320     #endif
00321 #endif
00322
00323 #if !defined(INFINITE)
00324     #define INFINITE 0xffffffff
00325 #endif
00326
00327 #ifdef __cplusplus
00328 }
00329 #endif
00330
00331 #endif /* _WD_UTILS_H_ */
00332

```

wd_kp.h File Reference

```
#include "windrvr.h"
```

Data Structures

- struct KP_OPEN_CALL
- struct KP_INIT

Macros

- #define __KERNEL__
- #define __KERPLUG__

Typedefs

- typedef void(__cdecl * KP_FUNC_CLOSE) (PVOID pDrvContext)
Called when WD_KernelPlugInClose() is called.
- typedef void(__cdecl * KP_FUNC_CALL) (PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall)
Called when WD_KernelPlugInCall() is called.
- typedef BOOL(__cdecl * KP_FUNC_INT_ENABLE) (PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall, PVOID *pIntContext)
Called when WD_IntEnable() is called, with a kernel plugin handler specified the plntContext will be passed to the rest of the functions handling interrupts.
- typedef void(__cdecl * KP_FUNC_INT_DISABLE) (PVOID pIntContext)
Called when WD_IntDisable() is called.
- typedef BOOL(__cdecl * KP_FUNC_INT_AT_IRQL) (PVOID pIntContext, BOOL *pfIsMyInterrupt)
Returns TRUE if needs DPC.

- **typedef DWORD(_cdecl * KP_FUNC_INT_AT_DPC) (PVOID pIntContext, DWORD dwCount)**
Returns the number of times to notify user-mode (i.e.
- **typedef BOOL(_cdecl * KP_FUNC_INT_AT_IRQL_MSI) (PVOID pIntContext, ULONG dwLastMessage, DWORD dwReserved)**
Returns TRUE if needs DPC.
- **typedef DWORD(_cdecl * KP_FUNC_INT_AT_DPC_MSI) (PVOID pIntContext, DWORD dwCount, ULONG dwLastMessage, DWORD dwReserved)**
Returns the number of times to notify user-mode (i.e.
- **typedef BOOL(_cdecl * KP_FUNC_EVENT) (PVOID pDrvContext, WD_EVENT *wd_event)**
returns TRUE if user need notification
- **typedef BOOL(_cdecl * KP_FUNC_OPEN) (KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID p←OpenData, PVOID *ppDrvContext)**
Called when WD_KernelPlugInOpen() is called.

Functions

- **BOOL __cdecl KP_Init (KP_INIT *kplInit)**
You must define a KP_Init() function to link to the device driver.

Macro Definition Documentation

__KERNEL__

```
#define __KERNEL__
```

Definition at line 7 of file wd_kp.h.

__KERPLUG__

```
#define __KERPLUG__
```

Definition at line 11 of file wd_kp.h.

Typedef Documentation

KP_FUNC_CALL

```
typedef void(__cdecl * KP_FUNC_CALL) (PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall)
```

Called when WD_KernelPlugInCall() is called.

Definition at line 23 of file wd_kp.h.

KP_FUNC_CLOSE

```
typedef void(__cdecl * KP_FUNC_CLOSE) (PVOID pDrvContext)
```

Called when WD_KernelPlugInClose() is called.

Definition at line 21 of file [wd_kp.h](#).

KP_FUNC_EVENT

`typedef BOOL(__cdecl * KP_FUNC_EVENT) (PVOID pDrvContext, WD_EVENT *wd_event)`

Returns TRUE if user need notification

Definition at line 46 of file [wd_kp.h](#).

KP_FUNC_INT_AT_DPC

`typedef DWORD(__cdecl * KP_FUNC_INT_AT_DPC) (PVOID pIntContext, DWORD dwCount)`

Returns the number of times to notify user-mode (i.e.

return from WD_IntWait)

Definition at line 37 of file [wd_kp.h](#).

KP_FUNC_INT_AT_DPC_MSI

`typedef DWORD(__cdecl * KP_FUNC_INT_AT_DPC_MSI) (PVOID pIntContext, DWORD dwCount, ULONG dwLastMessage, DWORD dwReserved)`

Returns the number of times to notify user-mode (i.e.

return from WD_IntWait)

Definition at line 43 of file [wd_kp.h](#).

KP_FUNC_INT_AT_IRQL

`typedef BOOL(__cdecl * KP_FUNC_INT_AT_IRQL) (PVOID pIntContext, BOOL *pfIsMyInterrupt)`

Returns TRUE if needs DPC.

Definition at line 33 of file [wd_kp.h](#).

KP_FUNC_INT_AT_IRQL_MSI

`typedef BOOL(__cdecl * KP_FUNC_INT_AT_IRQL_MSI) (PVOID pIntContext, ULONG dwLastMessage, DWORD dwReserved)`

Returns TRUE if needs DPC.

Definition at line 39 of file [wd_kp.h](#).

KP_FUNC_INT_DISABLE

`typedef void(__cdecl * KP_FUNC_INT_DISABLE) (PVOID pIntContext)`

Called when [WD_IntDisable\(\)](#) is called.

Definition at line 31 of file [wd_kp.h](#).

KP_FUNC_INT_ENABLE

```
typedef BOOL(__cdecl * KP_FUNC_INT_ENABLE) (PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall,  
PVOID *ppIntContext)
```

Called when [WD_IntEnable\(\)](#) is called, with a kernel plugin handler specified the pIntContext will be passed to the rest of the functions handling interrupts.

Returns TRUE if enable is successful

Definition at line [28](#) of file [wd_kp.h](#).

KP_FUNC_OPEN

```
typedef BOOL(__cdecl * KP_FUNC_OPEN) (KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData,  
PVOID *ppDrvContext)
```

Called when [WD_KernelPlugInOpen\(\)](#) is called.

pDrvContext returned will be passed to rest of the functions

Definition at line [63](#) of file [wd_kp.h](#).

Function Documentation

KP_Init()

```
BOOL __cdecl KP_Init (  
    KP_INIT * kpInit )
```

You must define a [KP_Init\(\)](#) function to link to the device driver.

You must define a [KP_Init\(\)](#) function to link to the device driver.

This function sets the name of the Kernel PlugIn driver and the driver's open callback function(s).

Parameters

out	<i>kplnIt</i>	Pointer to a pre-allocated Kernel PlugIn initialization information structure, whose fields should be updated by the function
-----	---------------	---

Returns

TRUE if successful. Otherwise FALSE.

Definition at line [65](#) of file [kp_pci.c](#).

wd_kp.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */  
00002  
00003 #ifndef _WD_KP_H_  
00004 #define _WD_KP_H_  
00005  
00006 #ifndef __KERNEL__  
00007     #define __KERNEL__  
00008 #endif  
00009  
00010 #ifndef __KERPLUG__
```

```

00011     #define __KERPLUG__
00012 #endif
00013
00014 #include "windrvr.h"
00015
00016 #ifdef __cplusplus
00017     extern "C" {
00018 #endif /* __cplusplus */
00019
00021 typedef void (__cdecl *KP_FUNC_CLOSE) (PVOID pDrvContext);
00023 typedef void (__cdecl *KP_FUNC_CALL) (PVOID pDrvContext,
00024     WD_KERNEL_PLUGIN_CALL *kpCall);
00028 typedef BOOL (__cdecl *KP_FUNC_INT_ENABLE) (PVOID pDrvContext,
00029     WD_KERNEL_PLUGIN_CALL *kpCall, PVOID *ppIntContext);
00031 typedef void (__cdecl *KP_FUNC_INT_DISABLE) (PVOID pIntContext);
00033 typedef BOOL (__cdecl *KP_FUNC_INT_AT_IRQL) (PVOID pIntContext,
00034     BOOL *pfIsMyInterrupt);
00037 typedef DWORD (__cdecl *KP_FUNC_INT_AT_DPC) (PVOID pIntContext, DWORD dwCount);
00039 typedef BOOL (__cdecl *KP_FUNC_INT_AT_IRQL_MSI) (PVOID pIntContext,
00040     ULONG dwLastMessage, DWORD dwReserved);
00043 typedef DWORD (__cdecl *KP_FUNC_INT_AT_DPC_MSI) (PVOID pIntContext,
00044     DWORD dwCount, ULONG dwLastMessage, DWORD dwReserved);
00046 typedef BOOL (__cdecl *KP_FUNC_EVENT) (PVOID pDrvContext, WD_EVENT *wd_event);
00047
00048 typedef struct
00049 {
00050     KP_FUNC_CLOSE funcClose;
00051     KP_FUNC_CALL funcCall;
00052     KP_FUNC_INT_ENABLE funcIntEnable;
00053     KP_FUNC_INT_DISABLE funcIntDisable;
00054     KP_FUNC_INT_AT_IRQL funcIntAtIrql;
00055     KP_FUNC_INT_AT_DPC funcIntAtDpc;
00056     KP_FUNC_INT_AT_IRQL_MSI funcIntAtIrqlMSI;
00057     KP_FUNC_INT_AT_DPC_MSI funcIntAtDpcMSI;
00058     KP_FUNC_EVENT funcEvent;
00059 } KP_OPEN_CALL;
00060
00063 typedef BOOL (__cdecl *KP_FUNC_OPEN) (KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
00064     PVOID pOpenData, PVOID *ppDrvContext);
00065
00066 typedef struct
00067 {
00068     DWORD dwVerWD;
00069     CHAR cDriverName[WD_MAX_KP_NAME_LENGTH];
00071     KP_FUNC_OPEN funcOpen;
00072     KP_FUNC_OPEN funcOpen_32_64;
00074 } KP_INIT;
00075
00077 BOOL __cdecl KP_Init(KP_INIT *kpInit);
00078
00079 #ifdef __cplusplus
00080     }
00081 #endif /* __cplusplus */
00082
00083 #endif /* _WD_KP_H_ */
00084

```

wd_log.h File Reference

Macros

- #define WD_FUNCTION WdFunctionLog
- #define WD_Close WD_CloseLog
- #define WD_Open WD_OpenLog

Functions

- ULONG **DLLCALLCONV WdFunctionLog** (DWORD wFuncNum, HANDLE h, PVOID pParam, DWORD dw←
Size, BOOL fWait)
- HANDLE **DLLCALLCONV WD_OpenLog** (void)
- void **DLLCALLCONV WD_CloseLog** (HANDLE hWD)
- DWORD **DLLCALLCONV WD_LogStart** (const char *sFileName, const char *sMode)
Opens a log file.
- VOID **DLLCALLCONV WD_LogStop** (void)

- Closes a log file.*
- VOID **DLLCALLCONV WD_LogAdd** (const char *sFormat,...)
Opens a log file.

Macro Definition Documentation

WD_Close

```
#define WD_Close WD_CloseLog
```

Definition at line 64 of file [wd_log.h](#).

WD_FUNCTION

```
#define WD_FUNCTION WdFunctionLog
```

Definition at line 63 of file [wd_log.h](#).

WD_Open

```
#define WD_Open WD_OpenLog
```

Definition at line 65 of file [wd_log.h](#).

Function Documentation

WD_CloseLog()

```
void DLLCALLCONV WD_CloseLog (
    HANDLE hWD )
```

WD_LogAdd()

```
VOID DLLCALLCONV WD_LogAdd (
    const char * sFormat,
    ... )
```

Opens a log file.

Parameters

in	sFormat	Adds user printouts into log file.
in	...	Optional format arguments

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

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\(\)](#)

WD_LogStart()

```
DWORD DLLCALLCONV WD_LogStart (
    const char * sFileName,
    const char * sMode )
```

Opens a log file.

Parameters

in	sFileName	Name of log file to be opened
in	sMode	Type of access permitted. For example, NULL or w opens an empty file for writing, and if the given file exists, its contents are destroyed; a opens a file for writing at the end of the file (i.e., append).

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

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\(\)](#)

WD_LogStop()

```
VOID DLLCALLCONV WD_LogStop (
    void )
```

Closes a log file.

Returns

None

WD_OpenLog()

```
HANDLE DLLCALLCONV WD_OpenLog (
    void )
```

WdFunctionLog()

```
ULONG DLLCALLCONV WdFunctionLog (
    DWORD wFuncNum,
    HANDLE h,
```

```
PVOID pParam,  
DWORD dwSize,  
BOOL fWait )
```

wd_log.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */  
00002  
00003 #ifndef _WD_LOG_H_  
00004 #define _WD_LOG_H_  
00005  
00006 #ifdef __cplusplus  
00007     extern "C" {  
00008 #endif  
00009  
00010 ULONG DLLCALLCONV WdFunctionLog(DWORD wFuncNum, HANDLE h, PVOID pParam,  
00011     DWORD dwSize, BOOL fWait);  
00012 HANDLE DLLCALLCONV WD_OpenLog(void);  
00013 void DLLCALLCONV WD_CloseLog(HANDLE hWD);  
00014  
00032 DWORD DLLCALLCONV WD_LogStart(const char *sFileName, const char *sMode);  
00033  
00041 VOID DLLCALLCONV WD_LogStop(void);  
00042  
00057 VOID DLLCALLCONV WD_LogAdd(const char *sFormat, ...);  
00058  
00059 #undef WD_FUNCTION  
00060 #undef WD_Close  
00061 #undef WD_Open  
00062  
00063 #define WD_FUNCTION WdFunctionLog  
00064 #define WD_Close WD_CloseLog  
00065 #define WD_Open WD_OpenLog  
00066  
00067 #ifdef __cplusplus  
00068 }  
00069 #endif  
00070  
00071 #endif /* _WD_LOG_H_ */  
00072
```

wd_types.h File Reference

Typedefs

- `typedef unsigned char u8`
- `typedef unsigned short u16`
- `typedef unsigned int u32`
- `typedef unsigned long long u64`

Typedef Documentation

u16

```
typedef unsigned short u16
```

Definition at line 9 of file [wd_types.h](#).

u32

```
typedef unsigned int u32
```

Definition at line 10 of file [wd_types.h](#).

u64

```
typedef unsigned long long u64
```

Definition at line 15 of file [wd_types.h](#).

u8

```
typedef unsigned char u8
```

Definition at line 8 of file [wd_types.h](#).

wd_types.h

[Go to the documentation of this file.](#)

```
00001 #ifndef _WD_TYPES_H_
00002 #define _WD_TYPES_H_
00003
00004 #if defined(__cplusplus)
00005     extern "C" {
00006 #endif
00007
00008     typedef unsigned char u8;
00009     typedef unsigned short u16;
00010     typedef unsigned int u32;
00011
00012     #if defined(WINNT)
00013         typedef unsigned __int64 u64;
00014     #else
00015         typedef unsigned long long u64;
00016     #endif
00017
00018 #if defined(__cplusplus)
00019 }
00020 #endif
00021
00022 #endif /* _WD_TYPES_H_ */
00023
```

wd_ver.h File Reference

Macros

- `#define WD_MAJOR_VER 15`
- `#define WD_MINOR_VER 1`
- `#define WD_SUB_MINOR_VER 1`
- `#define WD_MAJOR_VER_STR "15"`
- `#define WD_MINOR_VER_STR "1"`
- `#define WD_SUB_MINOR_VER_STR "1"`
- `#define WD_VER_BETA_STR ""`
- `#define WD_VERSION_MAC_STR`
- `#define WD_VERSION_STR`
- `#define WD_VER (WD_MAJOR_VER * 100 + WD_MINOR_VER * 10 + WD_SUB_MINOR_VER)`
- `#define WD_VER_ITOA WD_MAJOR_VER_STR WD_MINOR_VER_STR WD_SUB_MINOR_VER_STR`
- `#define COPYRIGHTS_YEAR_STR "2022"`
- `#define COPYRIGHTS_FULL_STR`

Macro Definition Documentation

COPYRIGHTS_FULL_STR

```
#define COPYRIGHTS_FULL_STR
```

Value:

```
"Jungo Connectivity Confidential. Copyright (c) \" \
COPYRIGHTS_YEAR_STR " Jungo Connectivity Ltd. https://www.jungo.com"
```

Definition at line [29](#) of file [wd_ver.h](#).

COPYRIGHTS_YEAR_STR

```
#define COPYRIGHTS_YEAR_STR "2022"
```

Definition at line [28](#) of file [wd_ver.h](#).

WD_MAJOR_VER

```
#define WD_MAJOR_VER 15
```

Definition at line [10](#) of file [wd_ver.h](#).

WD_MAJOR_VER_STR

```
#define WD_MAJOR_VER_STR "15"
```

Definition at line [14](#) of file [wd_ver.h](#).

WD_MINOR_VER

```
#define WD_MINOR_VER 1
```

Definition at line [11](#) of file [wd_ver.h](#).

WD_MINOR_VER_STR

```
#define WD_MINOR_VER_STR "1"
```

Definition at line [15](#) of file [wd_ver.h](#).

WD_SUB_MINOR_VER

```
#define WD_SUB_MINOR_VER 1
```

Definition at line [12](#) of file [wd_ver.h](#).

WD_SUB_MINOR_VER_STR

```
#define WD_SUB_MINOR_VER_STR "1"
```

Definition at line 16 of file [wd_ver.h](#).

WD_VER

```
#define WD_VER (WD_MAJOR_VER * 100 + WD_MINOR_VER * 10 + WD_SUB_MINOR_VER)
```

Definition at line 25 of file [wd_ver.h](#).

WD_VER_BETA_STR

```
#define WD_VER_BETA_STR ""
```

Definition at line 19 of file [wd_ver.h](#).

WD_VER_ITOA

```
#define WD_VER_ITOA WD_MAJOR_VER_STR WD_MINOR_VER_STR WD_SUB_MINOR_VER_STR
```

Definition at line 26 of file [wd_ver.h](#).

WD_VERSION_MAC_STR

```
#define WD_VERSION_MAC_STR
```

Value:

```
WD_MAJOR_VER_STR "." WD_MINOR_VER_STR "." \
WD_SUB_MINOR_VER_STR " " WD_VER_BETA_STR
```

Definition at line 21 of file [wd_ver.h](#).

WD_VERSION_STR

```
#define WD_VERSION_STR
```

Value:

```
WD_MAJOR_VER_STR "." WD_MINOR_VER_STR "." \
WD_SUB_MINOR_VER_STR WD_VER_BETA_STR
```

Definition at line 23 of file [wd_ver.h](#).

wd_ver.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 /* @JUNGO_COPYRIGHT_GPL@ */
00004
00005 /* @JUNGO_COPYRIGHT_GPL_OTHER_OS@ */
00006
00007 #ifndef _WD_VER_H_
00008 #define _WD_VER_H_
00009
00010 #define WD_MAJOR_VER 15
```

```

00011 #define WD_MINOR_VER 1
00012 #define WD_SUB_MINOR_VER 1
00013
00014 #define WD_MAJOR_VER_STR "15"
00015 #define WD_MINOR_VER_STR "1"
00016 #define WD_SUB_MINOR_VER_STR "1"
00017
00018 /* %% Replace with empty string for non-beta version %% */
00019 #define WD_VER_BETA_STR ""
00020
00021 #define WD_VERSION_MAC_STR WD_MAJOR_VER_STR "." WD_MINOR_VER_STR "." \
00022     WD_SUB_MINOR_VER_STR " " WD_VER_BETA_STR
00023 #define WD_VERSION_STR WD_MAJOR_VER_STR "." WD_MINOR_VER_STR "." \
00024     WD_SUB_MINOR_VER_STR WD_VER_BETA_STR
00025 #define WD_VER (WD_MAJOR_VER * 100 + WD_MINOR_VER * 10 + WD_SUB_MINOR_VER)
00026 #define WD_VER_ITOA WD_MAJOR_VER_STR WD_MINOR_VER_STR WD_SUB_MINOR_VER_STR
00027
00028 #define COPYRIGHTS_YEAR_STR "2022"
00029 #define COPYRIGHTS_FULL_STR "Jungo Connectivity Confidential. Copyright (c) " \
00030     COPYRIGHTS_YEAR_STR " Jungo Connectivity Ltd. https://www.jungo.com"
00031
00032 #endif /* _WD_VER_H_ */
00033

```

wdc_defs.h File Reference

```
#include "wdc_lib.h"
```

Data Structures

- struct [WDC_ADDR_DESC](#)
Address space information struct.
- struct [WDC_DEVICE](#)
Device information struct.

Macros

- #define [WDC_MEM_DIRECT_ADDR](#)(pAddrDesc) (pAddrDesc)->pUserDirectMemAddr
Get direct memory address pointer.
- #define [WDC_ADDR_IS_MEM](#)(pAddrDesc) (pAddrDesc)->flsMemory
Check if memory or I/O address.
- #define [WDC_GET_KP_HANDLE](#)(pDev) (([WDC_DEVICE](#) *)(([PWDC_DEVICE](#))(pDev)))->kerPlug.hKernelPlugIn
Get Kernel PlugIn handle.
- #define [WDC_IS_KP](#)(pDev) (BOOL)([WDC_GET_KP_HANDLE](#)(pDev))
Does the device use a Kernel PlugIn driver.
- #define [WDC_GET_PCARD](#)(pDev) (&(([PWDC_DEVICE](#))(pDev))->cardReg.Card))
Get pointer to device's resources struct ([WD_CARD](#)) from device information struct pointer.
- #define [WDC_GET_CARD_HANDLE](#)(pDev) ((([PWDC_DEVICE](#))(pDev))->cardReg.hCard)
Get card handle from device information struct pointer.
- #define [WDC_GET_PPCI_SLOT](#)(pDev) (&((([PWDC_DEVICE](#))(pDev))->slot))
Get pointer to WD PCI slot from device information struct pointer.
- #define [WDC_GET_PPCI_ID](#)(pDev) (&((([PWDC_DEVICE](#))(pDev))->id))
Get pointer to WD id from device information struct pointer.
- #define [WDC_GET_ADDR_DESC](#)(pDev, dwAddrSpace) (&((([PWDC_DEVICE](#))(pDev))->pAddrDesc[dwAddrSpace]))
Get address space descriptor from device information struct pointer.
- #define [WDC_GET_ADDR_SPACE_SIZE](#)(pDev, dwAddrSpace) ((([PWDC_DEVICE](#))(pDev))->pAddrDesc[dwAddrSpace]).qwBytes)

- `#define WDC_GET_ENABLED_INT_TYPE(pDev) ((PWDC_DEVICE)pDev)->Int.dwEnabledIntType`
Get type of enabled interrupt from device information struct pointer.
- `#define WDC_GET_INT_OPTIONS(pDev) ((PWDC_DEVICE)pDev)->Int.dwOptions`
Get interrupt options field from device information struct pointer.
- `#define WDC_INT_IS_MSI(dwIntType) (dwIntType & (INTERRUPT_MESSAGE | INTERRUPT_MESSAGE_X))`
Returns whether the MSI/MSI-X interrupt option is set.
- `#define WDC_GET_ENABLED_INT_LAST_MSG(pDev)`
Get the message data of the last received MSI/MSI-X interrupt.

Typedefs

- `typedef struct WDC_DEVICE WDC_DEVICE`
Device information struct.
- `typedef struct WDC_DEVICE * PWDC_DEVICE`

Macro Definition Documentation

WDC_ADDR_IS_MEM

```
#define WDC_ADDR_IS_MEM(
    pAddrDesc ) (pAddrDesc)->fIsMemory
```

Check if memory or I/O address.

Definition at line 87 of file [wdc_defs.h](#).

WDC_GET_ADDR_DESC

```
#define WDC_GET_ADDR_DESC (
    pDev,
    dwAddrSpace ) (&(((PWDC_DEVICE)(pDev))->pAddrDesc[dwAddrSpace]))
```

Get address space descriptor from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
in	<i>dwAddrSpace</i>	Address space index

Returns

Returns address space descriptor

Definition at line 166 of file [wdc_defs.h](#).

WDC_GET_ADDR_SPACE_SIZE

```
#define WDC_GET_ADDR_SPACE_SIZE (
    pDev,
    dwAddrSpace ) (((((PWDC_DEVICE)(pDev))->pAddrDesc[dwAddrSpace]).qwBytes)
```

Get address space descriptor size from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
in	<i>dwAddrSpace</i>	Address space index

Returns

Returns address space descriptor size

Definition at line 179 of file [wdc_defs.h](#).

WDC_GET_CARD_HANDLE

```
#define WDC_GET_CARD_HANDLE( \
    pDev ) (((PWDC_DEVICE)(pDev))->cardReg.hCard)
```

Get card handle from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns card handle

Definition at line 132 of file [wdc_defs.h](#).

WDC_GET_ENABLED_INT_LAST_MSG

```
#define WDC_GET_ENABLED_INT_LAST_MSG( \
    pDev )
```

Value:

```
WDC_INT_IS_MSI(WDC_GET_ENABLED_INT_TYPE(pDev)) ? \
(((PWDC_DEVICE)pDev)->Int.dwLastMessage) : 0
```

Get the message data of the last received MSI/MSI-X interrupt.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns message data of the last received MSI/MSI-X interrupt

Definition at line 227 of file [wdc_defs.h](#).

WDC_GET_ENABLED_INT_TYPE

```
#define WDC_GET_ENABLED_INT_TYPE( pDev ) ( (PWDC_DEVICE)pDev )->Int.dwEnabledIntType
```

Get type of enabled interrupt from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns type of enabled interrupt

Definition at line 191 of file [wdc_defs.h](#).

WDC_GET_INT_OPTIONS

```
#define WDC_GET_INT_OPTIONS( pDev ) ( (PWDC_DEVICE)pDev )->Int.dwOptions
```

Get interrupt options field from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns interrupt options field

Definition at line 203 of file [wdc_defs.h](#).

WDC_GET_KP_HANDLE

```
#define WDC_GET_KP_HANDLE( pDev ) ( (WDC_DEVICE *) ( (PWDC_DEVICE) (pDev) ) )->kerPlug.hKernelPlugIn
```

Get Kernel PlugIn handle.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns Kernel PlugIn handle

Definition at line 101 of file [wdc_defs.h](#).

WDC_GET_PCARD

```
#define WDC_GET_PCARD (pDev) (&(((PWDC_DEVICE)(pDev))->cardReg.Card))
```

Get pointer to device's resources struct ([WD_CARD](#)) from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns pointer to device's resources struct ([WD_CARD](#))

Definition at line 121 of file [wdc_defs.h](#).

WDC_GET_PPCI_ID

```
#define WDC_GET_PPCI_ID (pDev) (&(((PWDC_DEVICE)(pDev))->id))
```

Get pointer to WD id from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns pointer to WD id

Definition at line 154 of file [wdc_defs.h](#).

WDC_GET_PPCI_SLOT

```
#define WDC_GET_PPCI_SLOT (pDev) (&(((PWDC_DEVICE)(pDev))->slot))
```

Get pointer to WD PCI slot from device information struct pointer.

Parameters

in	<i>pDev</i>	Pointer to device information struct
----	-------------	--------------------------------------

Returns

Returns pointer to WD PCI slot

Definition at line 143 of file [wdc_defs.h](#).

WDC_INT_IS_MSI

```
#define WDC_INT_IS_MSI( dwIntType ) (dwIntType & (INTERRUPT_MESSAGE | INTERRUPT_MESSAGE_X))
```

Returns whether the MSI/MSI-X interrupt option is set.

Parameters

in	<i>dwIntType</i>	Interrupt type
----	------------------	----------------

Returns

TRUE if the MSI/MSI-X interrupt option is set else FALSE

Definition at line 215 of file [wdc_defs.h](#).

WDC_IS_KP

```
#define WDC_IS_KP( pDev ) (BOOL) (WDC_GET_KP_HANDLE(pDev))
```

Does the device use a Kernel Plugin driver.

Definition at line 105 of file [wdc_defs.h](#).

WDC_MEM_DIRECT_ADDR

```
#define WDC_MEM_DIRECT_ADDR( pAddrDesc ) (pAddrDesc) ->pUserDirectMemAddr
```

Get direct memory address pointer.

Definition at line 83 of file [wdc_defs.h](#).

TypeDef Documentation

PWDC_DEVICE

```
typedef struct WDC_DEVICE * PWDC_DEVICE
```

WDC_DEVICE

```
typedef struct WDC_DEVICE WDC_DEVICE
```

Device information struct.

wdc_defs.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WDC_DEFS_H_
00004 #define _WDC_DEFS_H_
00005
00006 /*****
00007 * File: wdc_defs.h - WD card (WDC) library low-level definitions header.
00008 * This file is used by the WDC library's source files and from
00009 * device-specific sample/generated code XXX library files, but not
00010 * from high-level diagnostic code (wdc_diag/xxx_diag).
00011 *****/
00012
00013 #include "wdc_lib.h"
00014
00015 #ifdef __cplusplus
00016     extern "C" {
00017 #endif
00018
00019 /*****
00020 General definitions
00021 *****/
00022 /* -----
00023     Memory / I/O / Registers
00024 ----- */
00025
00026 typedef struct {
00027     DWORD dwAddrSpace;
00028     BOOL fIsMemory;
00029     DWORD dwItemIndex;
00030     DWORD reserved;
00031     UINT64 qwBytes;
00032     KPTR pAddr;
00033     UPTR pUserDirectMemAddr;
00034     PAD_TO_64(pUserDirectMemAddr)
00035 } WDC_ADDR_DESC;
00036
00037 /* -----
00038     General
00039 ----- */
00040
00041 typedef struct WDC_DEVICE {
00042     WD_PCI_ID id;
00043     WD_PCI_SLOT slot;
00044     DWORD dwNumAddrSpaces;
00045     PAD_TO_64(dwNumAddrSpaces)
00046     WDC_ADDR_DESC *pAddrDesc;
00047     PAD_TO_64(pAddrDesc)
00048     WD_CARD_REGISTER cardReg;
00049     WD_KERNEL_PLUGIN kerPlug;
00050     WD_INTERRUPT Int;
00051     HANDLE hIntThread;
00052     PAD_TO_64(hIntThread)
00053
00054     WD_EVENT Event;
00055     HANDLE hEvent;
00056     PAD_TO_64(hEvent)
00057
00058     PVOID pCtx;
00059     PAD_TO_64(pCtx)
00060 } WDC_DEVICE, *PWDC_DEVICE;
00061
00062 /*****
00063 General utility macros
00064 *****/
00065 /* -----
00066     Memory / I/O / Registers
00067 ----- */
00068
00069 /* NOTE: pAddrDesc param should be of type WDC_ADDR_DESC */
00070 #if defined(__KERNEL__)
00071     #define WDC_MEM_DIRECT_ADDR(pAddrDesc) (pAddrDesc)->pAddr
00072     #else
00073         #define WDC_MEM_DIRECT_ADDR(pAddrDesc) (pAddrDesc)->pUserDirectMemAddr
00074     #endif
00075
00076     #define WDC_ADDR_IS_MEM(pAddrDesc) (pAddrDesc)->fIsMemory
00077
00078 /* -----
00079     Kernel PlugIn
00080 ----- */
00081 #define WDC_GET_KP_HANDLE(pDev) \
00082     ((WDC_DEVICE *)((PWDC_DEVICE)(pDev)))->kerPlug.hKernelPlugIn
00083
00084 #define WDC_IS_KP(pDev) (BOOL)(WDC_GET_KP_HANDLE(pDev))
00085
```

```

00106
00107 /* -----
00108     General
00109 ----- */
00110
00111 #define WDC_GET_PCARD(pDev) (&(((PWDC_DEVICE)(pDev))->cardReg.Card))
00112
00113 #define WDC_GET_CARD_HANDLE(pDev) (((PWDC_DEVICE)(pDev))->cardReg.hCard)
00114
00115 #define WDC_GET_PPCI_SLOT(pDev) (&(((PWDC_DEVICE)(pDev))->slot))
00116
00117 #define WDC_GET_PPCT_ID(pDev) (&(((PWDC_DEVICE)(pDev))->id))
00118
00119 #define WDC_GET_ADDR_DESC(pDev, dwAddrSpace) \
00120     (&(((PWDC_DEVICE)(pDev))->pAddrDesc[dwAddrSpace]))
00121
00122 #define WDC_GET_ADDR_SPACE_SIZE(pDev, dwAddrSpace) \
00123     (((PWDC_DEVICE)(pDev))->pAddrDesc[dwAddrSpace]).qwBytes)
00124
00125 #define WDC_GET_ENABLED_INT_TYPE(pDev) \
00126     (((PWDC_DEVICE)(pDev))->Int.dwEnabledIntType)
00127
00128 #define WDC_GET_INT_OPTIONS(pDev) ((PWDC_DEVICE)pDev)->Int.dwOptions
00129
00130 #define WDC_INT_IS_MSI(dwIntType) \
00131     (dwIntType & (INTERRUPT_MESSAGE | INTERRUPT_MESSAGE_X))
00132
00133 #define WDC_GET_ENABLED_INT_LAST_MSG(pDev) \
00134     WDC_INT_IS_MSI(WDC_GET_ENABLED_INT_TYPE(pDev)) ? \
00135     (((PWDC_DEVICE)pDev)->Int.dwLastMessage) : 0
00136
00137 #ifdef __cplusplus
00138 }
00139#endif
00140 #endif /* _WDC_DEFS_H_ */
00141

```

wdc_lib.h File Reference

```

#include "windrvr.h"
#include "windrvr_int_thread.h"
#include "windrvr_events.h"
#include "bits.h"
#include "pci_regs.h"

```

Data Structures

- struct [WDC_PCI_SCAN_RESULT](#)
PCI scan results.
- struct [WDC_PCI_SCAN_CAPS_RESULT](#)
PCI capabilities scan results.
- struct [WDC_INTERRUPT_PARAMS](#)

Macros

- #define MAX_NAME 128
- #define MAX_DESC 128
- #define MAX_NAME_DISPLAY 22
- #define WDC_DRV_OPEN_CHECK_VER 0x1
Compare source files WinDriver version with that of the running WinDriver kernel.
- #define WDC_DRV_OPEN_REG_LIC 0x2
Register WinDriver license.
- #define WDC_DRV_OPEN_BASIC 0x0

- No option -> perform only the basic open driver tasks, which are always performed by WDC_DriverOpen (mainly - open a handle to WinDriver)
- #define **WDC_DRV_OPEN_KP** WDC_DRV_OPEN_BASIC
 - Kernel PlugIn driver open options <=> basic.
 - #define **WDC_DRV_OPEN_ALL** (WDC_DRV_OPEN_CHECK_VER | WDC_DRV_OPEN_REG_LIC)
 - #define **WDC_DRV_OPEN_DEFAULT** WDC_DRV_OPEN_ALL
 - #define **WDC_DBG_OUT_DBM** 0x1
 - Send WDC debug messages to the Debug Monitor.
 - #define **WDC_DBG_OUT_FILE** 0x2
 - Send WDC debug messages to a debug file (default: stderr) [User-mode only].
 - #define **WDC_DBG_LEVEL_ERR** 0x10
 - Display only error WDC debug messages.
 - #define **WDC_DBG_LEVEL_TRACE** 0x20
 - Display error and trace WDC debug messages.
 - #define **WDC_DBG_NONE** 0x100
 - Do not print debug messages.
 - #define **WDC_DBG_DEFAULT** (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
 - Convenient debug options combinations/defintions.
 - #define **WDC_DBG_DBM_ERR** (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_ERR)
 - #define **WDC_DBG_DBM_TRACE** (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
 - #define **WDC_DBG_FILE_ERR** (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
 - #define **WDC_DBG_FILE_TRACE** (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
 - #define **WDC_DBG_DBM_FILE_ERR** (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
 - #define **WDC_DBG_DBM_FILE_TRACE** (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
 - #define **WDC_DBG_FULL** (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
 - #define **WDC_SLEEP_BUSY** 0
 - Sleep options.
 - #define **WDC_SLEEP_NON_BUSY** SLEEP_NON_BUSY
 - #define **WDC_SIZE_8** ((DWORD)sizeof(BYTE))
 - #define **WDC_SIZE_16** ((DWORD)sizeof(WORD))
 - #define **WDC_SIZE_32** ((DWORD)sizeof(UINT32))
 - #define **WDC_SIZE_64** ((DWORD)sizeof(UINT64))
 - #define **WDC_ADDR_MODE_TO_SIZE**(mode) (DWORD)mode
 - #define **WDC_ADDR_SIZE_TO_MODE**(size)
 - #define **WDC_AD_CFG_SPACE** 0xFF
 - Device configuration space identifier (PCI configuration space)
 - #define **WDC_ReadMem8**(addr, off) *(volatile BYTE *)((UPTR)(addr) + (UPTR)(off))
 - Direct memory read/write macros.
 - #define **WDC_ReadMem16**(addr, off) *(volatile WORD *)((UPTR)(addr) + (UPTR)(off))
 - reads 2 byte (16 bits) from a specified memory address.
 - #define **WDC_ReadMem32**(addr, off) *(volatile UINT32 *)((UPTR)(addr) + (UPTR)(off))
 - reads 4 byte (32 bits) from a specified memory address.
 - #define **WDC_ReadMem64**(addr, off) *(volatile UINT64 *)((UPTR)(addr) + (UPTR)(off))
 - reads 8 byte (64 bits) from a specified memory address.
 - #define **WDC_WriteMem8**(addr, off, val) *((volatile BYTE *)(((UPTR)(addr) + (UPTR)(off))) = (val))
 - writes 1 byte (8 bits) to a specified memory address.
 - #define **WDC_WriteMem16**(addr, off, val) *((volatile WORD *)(((UPTR)(addr) + (UPTR)(off))) = (val))
 - writes 2 byte (16 bits) to a specified memory address.
 - #define **WDC_WriteMem32**(addr, off, val) *((volatile UINT32 *)(((UPTR)(addr) + (UPTR)(off))) = (val))
 - writes 4 byte (32 bits) to a specified memory address.
 - #define **WDC_WriteMem64**(addr, off, val) *((volatile UINT64 *)(((UPTR)(addr) + (UPTR)(off))) = (val))
 - writes 8 byte (64 bits) to a specified memory address.

- #define **WDC_ReadAddrBlock8**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_ReadAddrBlock with 1 byte mode.
- #define **WDC_ReadAddrBlock16**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_ReadAddrBlock with 2 bytes mode.
- #define **WDC_ReadAddrBlock32**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_ReadAddrBlock with 4 bytes mode.
- #define **WDC_ReadAddrBlock64**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_ReadAddrBlock with 8 bytes mode.
- #define **WDC_WriteAddrBlock8**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_WriteAddrBlock with 1 byte mode.
- #define **WDC_WriteAddrBlock16**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_WriteAddrBlock with 2 bytes mode.
- #define **WDC_WriteAddrBlock32**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_WriteAddrBlock with 4 bytes mode.
- #define **WDC_WriteAddrBlock64**(hDev, dwAddrSpace, dwOffset, dwBytes, pData, options)
WDC_WriteAddrBlock with 8 bytes mode.
- #define **WDC_DMAGetGlobalHandle**(pDma) ((pDma)->hDma)
Utility macro that returns a contiguous DMA global handle that can be used for buffer sharing between multiple processes.

Typedefs

- typedef void * **WDC_DEVICE_HANDLE**
Handle to device information struct.
- typedef DWORD **WDC_DRV_OPEN_OPTIONS**
- typedef DWORD **WDC_DBG_OPTIONS**
- typedef DWORD **WDC_SLEEP_OPTIONS**
- typedef DWORD **WDC_ADDR_SIZE**

Enumerations

- enum **WDC_DIRECTION** { **WDC_WRITE** , **WDC_READ** , **WDC_READ_WRITE** }
- enum **WDC_ADDR_RW_OPTIONS** { **WDC_ADDR_RW_DEFAULT** = 0x0 , **WDC_ADDR_RW_NO_AUTOINC** = 0x4 }
Read/write address options.
- enum **WDC_ADDR_MODE** { **WDC_MODE_8** = **WDC_SIZE_8** , **WDC_MODE_16** = **WDC_SIZE_16** , **WDC_MODE_32** = **WDC_SIZE_32** , **WDC_MODE_64** = **WDC_SIZE_64** }

Functions

- HANDLE **DLLCALLCONV WDC_GetWDHandle** (void)
Get a handle to WinDriver.
- PVOID **DLLCALLCONV WDC_GetDevContext** (_In_ **WDC_DEVICE_HANDLE** hDev)
Returns the device's user context information.
- **WD_BUS_TYPE DLLCALLCONV WDC_GetBusType** (_In_ **WDC_DEVICE_HANDLE** hDev)
*Returns the device's bus type: **WD_BUS_PCI**, **WD_BUS_ISA** or **WD_BUS_UNKNOWN**.*
- DWORD **DLLCALLCONV WDC_Sleep** (_In_ DWORD dwMicroSecs, _In_ **WDC_SLEEP_OPTIONS** options)
Delays execution for the specified duration of time (in microseconds).
- DWORD **DLLCALLCONV WDC_Version** (_Outptr_ CHAR *pcVersion, _In_ DWORD dwLen, _Outptr_ DWORD *pdwVersion)
Returns the version number of the WinDriver kernel module used by the WDC library.

- DWORD **DLLCALLCONV WDC_DriverOpen** (_In_ WDC_DRV_OPEN_OPTIONS openOptions, _In_ const CHAR *pcLicense)
Opens and stores a handle to WinDriver's kernel module and initializes the WDC library according to the open options passed to it.
- DWORD **DLLCALLCONV WDC_DriverClose** (void)
Closes the WDC WinDriver handle (acquired and stored by a previous call to [WDC_DriverOpen\(\)](#)) and uninitialized the WDC library.
- DWORD **DLLCALLCONV WDC_PciScanDevices** (_In_ DWORD dwVendorId, _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult)
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 locations.
- DWORD **DLLCALLCONV WDC_PciScanDevicesByTopology** (_In_ DWORD dwVendorId, _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult)
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 locations.
- DWORD **DLLCALLCONV WDC_PciScanRegisteredDevices** (_In_ DWORD dwVendorId, _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult)
Scans the PCI bus for all devices with the specified vendor and device ID combination that have been registered to work with WinDriver, and returns information regarding the matching devices that were found and their locations.
- DWORD **DLLCALLCONV WDC_PciGetExpressOffset** (_In_ WDC_DEVICE_HANDLE hDev, _Outptr_ DWORD *pdwOffset)
Retrieves the PCI Express configuration registers' offset in the device's configuration space.
- DWORD **DLLCALLCONV WDC_PciGetHeaderType** (_In_ WDC_DEVICE_HANDLE hDev, _Outptr_ WDC_PCI_HEADER_TYPE *pHeaderType)
Retrieves the PCI device's configuration space header type.
- DWORD **DLLCALLCONV WDC_PciScanCaps** (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult)
Scans the basic PCI capabilities of the given device for the specified capability (or for all capabilities).
- DWORD **DLLCALLCONV WDC_PciScanCapsBySlot** (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult)
Scans the basic PCI capabilities of the given device for the specified capability (or for all capabilities).
- DWORD **DLLCALLCONV WDC_PciScanExtCaps** (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult)
Scans the extended (PCI Express) PCI capabilities of the given device for the specified capability (or for all capabilities).
- DWORD **DLLCALLCONV WDC_PciGetExpressGenBySlot** (_In_ WD_PCI_SLOT *pPciSlot)
Retrieves the PCI Express generation of a device.
- DWORD **DLLCALLCONV WDC_PciGetExpressGen** (_In_ WDC_DEVICE_HANDLE hDev)
Retrieves the PCI Express generation of a device.
- DWORD **DLLCALLCONV WDC_PciGetDeviceInfo** (_Inout_ WD_PCI_CARD_INFO *pDeviceInfo)
Retrieves a PCI device's resources information (memory and I/O ranges and interrupt information).
- DWORD **DLLCALLCONV WDC_PciSriovEnable** (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwNumVFs)
SR-IOV API functions are not part of the standard WinDriver API, and not included in the standard version of WinDriver.
- DWORD **DLLCALLCONV WDC_PciSriovDisable** (_In_ WDC_DEVICE_HANDLE hDev)
Disables SR-IOV for a supported device and removes all the assigned VFs.
- DWORD **DLLCALLCONV WDC_PciSriovGetNumVFs** (_In_ WDC_DEVICE_HANDLE hDev, _Outptr_ DWORD *pdwNumVFs)
Gets the number of virtual functions assigned to a supported device.
- DWORD **DLLCALLCONV WDC_PciDeviceOpen** (_Outptr_ WDC_DEVICE_HANDLE *phDev, _In_ const WD_PCI_CARD_INFO *pDeviceInfo, _In_ const PVOID pDevCtx)
Allocates and initializes a WDC PCI device structure, registers the device with WinDriver, and returns a handle to the device.

- **DWORD [DLLCALLTYPE WDC_IsaDeviceOpen](#) (_Outptr_ WDC_DEVICE_HANDLE *phDev, _In_ const WD_CARD *pDeviceInfo, _In_ const PVOID pDevCtx)**
Allocates and initializes a WDC ISA device structure, registers the device with WinDriver, and returns a handle to the device.
- **DWORD [DLLCALLTYPE WDC_PciDeviceClose](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Uninitializes a WDC PCI device structure and frees the memory allocated for it.
- **DWORD [DLLCALLTYPE WDC_IsaDeviceClose](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Uninitializes a WDC ISA device structure and frees the memory allocated for it.
- **DWORD [WDC_CardCleanupSetup](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ WD_TRANSFER *pTransCmds, _In_ DWORD dwCmds, _In_ BOOL fForceCleanup)**
Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:
- **DWORD [DLLCALLTYPE WDC_KernelPlugInOpen](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ const CHAR *pcKPDName, _In_ PVOID pKPOpenData)**
Opens a handle to a Kernel Plugin driver.
- **DWORD [DLLCALLTYPE WDC_CallKerPlug](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwMsg, _Inout_ PVOID pData, _Outptr_ PDWORD pdwResult)**
Sends a message from a user-mode application to a Kernel Plugin driver.
- **DWORD [DLLCALLTYPE WDC_ReadAddr8](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ BYTE *pbVal)**
Read/write a device's address space (8/16/32/64 bits)
- **DWORD [DLLCALLTYPE WDC_ReadAddr16](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ WORD *pwVal)**
reads 2 byte (16 bits) from a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_ReadAddr32](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ UINT32 *pdwVal)**
reads 4 byte (32 bits) from a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_ReadAddr64](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ UINT64 *pqwVal)**
reads 8 byte (64 bits) from a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_WriteAddr8](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ BYTE bVal)**
writes 1 byte (8 bits) to a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_WriteAddr16](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ WORD wVal)**
writes 2 byte (16 bits) to a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_WriteAddr32](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ UINT32 dwVal)**
writes 4 byte (32 bits) to a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_WriteAddr64](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ UINT64 qwVal)**
writes 8 byte (64 bits) to a specified memory or I/O address.
- **DWORD [DLLCALLTYPE WDC_ReadAddrBlock](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ DWORD dwBytes, _Outptr_ PVOID pData, _In_ WDC_ADDR_MODE mode, _In_ WDC_ADDR_RW_OPTIONS options)**
Reads a block of data from the device.
- **DWORD [DLLCALLTYPE WDC_WriteAddrBlock](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ DWORD dwBytes, _In_ PVOID pData, _In_ WDC_ADDR_MODE mode, _In_ WDC_ADDR_RW_OPTIONS options)**
Writes a block of data to the device.
- **DWORD [DLLCALLTYPE WDC_MultiTransfer](#) (_In_ WD_TRANSFER *pTransCmds, _In_ DWORD dwNumTrans)**
Performs a group of memory and/or I/O read/write transfers.

- **BOOL `DLLCALLCONV WDC_AddrSpaceIsActive (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dw← AddrSpace)`**

Checks if the specified memory or I/O address space is active ,i.e., if its size is not zero.
- **DWORD `DLLCALLCONV WDC_PciReadCfgBySlot (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _Outptr_ PVOID pData, _In_ DWORD dwBytes)`**

Read/write a block of any length from the PCI configuration space.
- **DWORD `DLLCALLCONV WDC_PciWriteCfgBySlot (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _In_ PVOID pData, _In_ DWORD dwBytes)`**

Write data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfg (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _Outptr_ PVOID pData, _In_ DWORD dwBytes)`**

Identify device by handle.
- **DWORD `DLLCALLCONV WDC_PciWriteCfg (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _In_ PVOID pData, _In_ DWORD dwBytes)`**

Writes data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfgBySlot8 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _Outptr_ BYTE *pbVal)`**

Read/write 8/16/32/64 bits from the PCI configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfgBySlot16 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _Outptr_ WORD *pwVal)`**

Reads 2 bytes (16 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfgBySlot32 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _Outptr_ UINT32 *pdwVal)`**

Reads 4 bytes (32 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfgBySlot64 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _Outptr_ UINT64 *pqwVal)`**

Reads 8 bytes (64 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciWriteCfgBySlot8 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _In_ BYTE bVal)`**

writes 1 byte (8 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciWriteCfgBySlot16 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _In_ WORD wVal)`**

writes 2 bytes (16 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciWriteCfgBySlot32 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _In_ UINT32 dwVal)`**

writes 4 bytes (32 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciWriteCfgBySlot64 (_In_ WD_PCI_SLOT *pPciSlot, _In_ DWORD dwOffset, _In_ UINT64 qwVal)`**

writes 8 bytes (64 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfg8 (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _Outptr_ BYTE *pbVal)`**

Read/write 8/16/32/64 bits from the PCI configuration space.
- **DWORD `DLLCALLCONV WDC_PciReadCfg16 (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _Outptr_ WORD *pwVal)`**

Reads 2 bytes (16 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

- **DWORD [DLLCALLTYPE WDC_PciReadCfg32](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _Outptr_ [UINT32](#) *pdwVal*)
Reads 4 bytes (32 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_PciReadCfg64](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _Outptr_ [UINT64](#) *pqwVal*)
Reads 8 bytes (64 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_PciWriteCfg8](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _In_ [BYTE](#) bVal*)
Writes 1 byte (8 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_PciWriteCfg16](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _In_ [WORD](#) wVal*)
Writes 2 bytes (16 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_PciWriteCfg32](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _In_ [UINT32](#) dwVal*)
Writes 4 bytes (32 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_PciWriteCfg64](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwOffset, _In_ [UINT64](#) qwVal*)
Writes 8 bytes (64 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.
- **DWORD [DLLCALLTYPE WDC_DMAContigBufLock](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _Outptr_ [VOID](#) *ppBuf, _In_ [DWORD](#) dwOptions, _In_ [DWORD](#) dwDMABufSize, _Outptr_ [WD_DMA](#) **ppDma*)
Allocates a contiguous DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.
- **DWORD [DLLCALLTYPE WDC_DMASGBufLock](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ [VOID](#) pBuf, _In_ [DWORD](#) dwOptions, _In_ [DWORD](#) dwDMABufSize, _Outptr_ [WD_DMA](#) **ppDma*)
Locks a pre-allocated user-mode memory buffer for DMA and returns the corresponding physical mappings of the locked DMA pages.
- **DWORD [DLLCALLTYPE WDC_DMATransactionContigInit](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _Outptr_ [VOID](#) *ppBuf, _In_ [DWORD](#) dwOptions, _In_ [DWORD](#) dwDMABufSize, _Outptr_ [WD_DMA](#) **ppDma, _In_ [WDC_INTERRUPT_PARAMS](#) *pInterruptParams, _In_ [DWORD](#) dwAlignment*)
Initializes the transaction, allocates a contiguous DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.
- **DWORD [DLLCALLTYPE WDC_DMATransactionsSGInit](#)** (*_In_ WDC_DEVICE_HANDLE hDev, _In_ [VOID](#) pBuf, _In_ [DWORD](#) dwOptions, _In_ [DWORD](#) dwDMABufSize, _Outptr_ [WD_DMA](#) **ppDma, _In_ [WDC_INTERRUPT_PARAMS](#) *pInterruptParams, _In_ [DWORD](#) dwMaxTransferSize, _In_ [DWORD](#) dwTransferElementSize*)
Initializes the transaction and locks a pre-allocated user-mode memory buffer for DMA.
- **DWORD [DLLCALLTYPE WDC_DMATransactionExecute](#)** (*_Inout_ [WD_DMA](#) *pDma, _In_ [DMA_TRANSACTION_CALLBACK](#) funcDMATransactionCallback, _In_ [VOID](#) DMATransactionCallbackCtx*)
Begins the execution of a specified DMA transaction.
- **DWORD [DLLCALLTYPE WDC_DMATransferCompletedAndCheck](#)** (*_Inout_ [WD_DMA](#) *pDma, _In_ [BOOL](#) fRunCallback*)
Notifies WinDriver that a device's DMA transfer operation is completed.
- **DWORD [DLLCALLTYPE WDC_DMATransactionRelease](#)** (*_In_ [WD_DMA](#) *pDma*)
Terminates a specified DMA transaction without deleting the associated [WD_DMA](#) transaction structure.
- **DWORD [DLLCALLTYPE WDC_DMATransactionUninit](#)** (*_In_ [WD_DMA](#) *pDma*)
Unlocks and frees the memory allocated for a DMA buffer transaction by a previous call to [WDC_DMATransactionContigInit\(\)](#) or [WDC_DMATransactionsSGInit\(\)](#).

- **DWORD [DLLCALLTYPE WDC_DMAReservedBufLock](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ PHYS_ADDR qwAddr, _Outptr_ PVOID *ppBuf, _In_ DWORD dwOptions, _In_ DWORD dwDMABufSize, _Outptr_ WD_DMA **ppDma)**
Locks a physical reserved memory buffer for DMA and returns the corresponding user mode address of locked DMA buffer.
- **DWORD [DLLCALLTYPE WDC_DMABufUnlock](#) (_In_ WD_DMA *pDma)**
Unlocks and frees the memory allocated for a DMA buffer by a previous call to [WDC_DMAContigBufLock\(\)](#), [WDC_DMASGBufLock\(\)](#) or [WDC_DMAReservedBufLock\(\)](#)
- **DWORD [DLLCALLTYPE WDC_DMABufGet](#) (_In_ DWORD hDma, _Outptr_ WD_DMA **ppDma)**
Retrieves a contiguous DMA buffer which was allocated by another process.
- **DWORD [DLLCALLTYPE WDC_DMASyncCpu](#) (_In_ WD_DMA *pDma)**
Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.
- **DWORD [DLLCALLTYPE WDC_DMASyncIclo](#) (_In_ WD_DMA *pDma)**
Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.
- **DWORD [DLLCALLTYPE WDC_IntEnable](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ WD_TRANSFER *pTransCmds, _In_ DWORD dwNumCmds, _In_ DWORD dwOptions, _In_ INT_HANDLER funcIntHandler, _In_ PVOID pData, _In_ BOOL fUseKP)**
Enables interrupt handling for the device.
- **DWORD [DLLCALLTYPE WDC_IntDisable](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Disables interrupt interrupt handling for the device, pursuant to a previous call to [WDC_IntEnable\(\)](#)
- **BOOL [DLLCALLTYPE WDC_IntIsEnabled](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Checks if a device's interrupts are currently enabled.
- **const CHAR *[DLLCALLTYPE WDC_IntType2Str](#) (_In_ DWORD dwIntType)**
Converts interrupt type to string.
- **DWORD [DLLCALLTYPE WDC_EventRegister](#) (_In_ WDC_DEVICE_HANDLE hDev, _In_ DWORD dwActions, _In_ EVENT_HANDLER funcEventHandler, _In_ PVOID pData, _In_ BOOL fUseKP)**
Registers the application to receive Plug-and-Play and power management events notifications for the device.
- **DWORD [DLLCALLTYPE WDC_EventUnregister](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Unregisters an application from a receiving Plug-and-Play and power management notifications for a device, pursuant to a previous call to [WDC_EventRegister\(\)](#)
- **BOOL [DLLCALLTYPE WDC_EventIsRegistered](#) (_In_ WDC_DEVICE_HANDLE hDev)**
Checks if the application is currently registered to receive Plug-and-Play and power management notifications for the device.
- **DWORD [DLLCALLTYPE WDC_SetDebugOptions](#) (_In_ WDC_DBG_OPTIONS dbgOptions, _In_ const CHAR *pcDbgFile)**
Sets debug options for the WDC library - see the description of [WDC_DBG_OPTIONS](#) for details regarding the possible debug options to set.
- **void [DLLCALLTYPE WDC_Err](#) (const CHAR *format,...)**
Displays debug error messages according to the WDC debug options.
- **void [DLLCALLTYPE WDC_Trace](#) (const CHAR *format,...)**
Displays debug trace messages according to the WDC debug options.

Macro Definition Documentation

MAX_DESC

```
#define MAX_DESC 128
```

Definition at line 29 of file [wdc.lib.h](#).

MAX_NAME

```
#define MAX_NAME 128
```

Definition at line 28 of file [wdc_lib.h](#).

MAX_NAME_DISPLAY

```
#define MAX_NAME_DISPLAY 22
```

Definition at line 30 of file [wdc_lib.h](#).

WDC_AD_CFG_SPACE

```
#define WDC_AD_CFG_SPACE 0xFF
```

Device configuration space identifier (PCI configuration space)

Definition at line 150 of file [wdc_lib.h](#).

WDC_ADDR_MODE_TO_SIZE

```
#define WDC_ADDR_MODE_TO_SIZE( mode ) (DWORD)mode
```

Definition at line 144 of file [wdc_lib.h](#).

WDC_ADDR_SIZE_TO_MODE

```
#define WDC_ADDR_SIZE_TO_MODE( size )
```

Value:

```
((size) > WDC_SIZE_32) ? WDC_MODE_64 : \
((size) > WDC_SIZE_16) ? WDC_MODE_32 : \
((size) > WDC_SIZE_8) ? WDC_MODE_16 : WDC_MODE_8)
```

Definition at line 145 of file [wdc_lib.h](#).

WDC_DBG_DBM_ERR

```
#define WDC_DBG_DBM_ERR (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_ERR)
```

Definition at line 88 of file [wdc_lib.h](#).

WDC_DBG_DBM_FILE_ERR

```
#define WDC_DBG_DBM_FILE_ERR (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
```

Definition at line 95 of file [wdc_lib.h](#).

WDC_DBG_DBM_FILE_TRACE

```
#define WDC_DBG_DBM_FILE_TRACE (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
```

Definition at line 97 of file [wdc_lib.h](#).

WDC_DBG_DBM_TRACE

```
#define WDC_DBG_DBM_TRACE (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
```

Definition at line 89 of file [wdc_lib.h](#).

WDC_DBG_DEFAULT

```
#define WDC_DBG_DEFAULT (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
```

Convenient debug options combinations/definitions.

Definition at line 86 of file [wdc_lib.h](#).

WDC_DBG_FILE_ERR

```
#define WDC_DBG_FILE_ERR (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
```

Definition at line 92 of file [wdc_lib.h](#).

WDC_DBG_FILE_TRACE

```
#define WDC_DBG_FILE_TRACE (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
```

Definition at line 93 of file [wdc_lib.h](#).

WDC_DBG_FULL

```
#define WDC_DBG_FULL (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
```

Definition at line 100 of file [wdc_lib.h](#).

WDC_DBG_LEVEL_ERR

```
#define WDC_DBG_LEVEL_ERR 0x10
```

Display only error WDC debug messages.

Definition at line 79 of file [wdc_lib.h](#).

WDC_DBG_LEVEL_TRACE

```
#define WDC_DBG_LEVEL_TRACE 0x20
```

Display error and trace WDC debug messages.

Definition at line 81 of file [wdc_lib.h](#).

WDC_DBG_NONE

```
#define WDC_DBG_NONE 0x100
```

Do not print debug messages.

Definition at line 83 of file [wdc_lib.h](#).

WDC_DBG_OUT_DBM

```
#define WDC_DBG_OUT_DBM 0x1
```

Send WDC debug messages to the Debug Monitor.

Definition at line 75 of file [wdc_lib.h](#).

WDC_DBG_OUT_FILE

```
#define WDC_DBG_OUT_FILE 0x2
```

Send WDC debug messages to a debug file (default: stderr) [User-mode only].

Definition at line 77 of file [wdc_lib.h](#).

WDC_DMAGetGlobalHandle

```
#define WDC_DMAGetGlobalHandle( pDma ) ((pDma)->hDma)
```

Utility macro that returns a contiguous DMA global handle that can be used for buffer sharing between multiple processes.

Definition at line 1750 of file [wdc_lib.h](#).

WDC_DRV_OPEN_ALL

```
#define WDC_DRV_OPEN_ALL (WDC_DRV_OPEN_CHECK_VER | WDC_DRV_OPEN_REG_LIC)
```

Definition at line 65 of file [wdc_lib.h](#).

WDC_DRV_OPEN_BASIC

```
#define WDC_DRV_OPEN_BASIC 0x0
```

No option -> perform only the basic open driver tasks, which are always performed by WDC_DriverOpen (mainly - open a handle to WinDriver)

Definition at line 62 of file [wdc_lib.h](#).

WDC_DRV_OPEN_CHECK_VER

```
#define WDC_DRV_OPEN_CHECK_VER 0x1
```

Compare source files WinDriver version with that of the running WinDriver kernel.

Definition at line 56 of file [wdc_lib.h](#).

WDC_DRV_OPEN_DEFAULT

```
#define WDC_DRV_OPEN_DEFAULT WDC_DRV_OPEN_ALL
```

Definition at line 69 of file [wdc_lib.h](#).

WDC_DRV_OPEN_KP

```
#define WDC_DRV_OPEN_KP WDC_DRV_OPEN_BASIC
```

Kernel PlugIn driver open options <=> basic.

Definition at line 64 of file [wdc_lib.h](#).

WDC_DRV_OPEN_REG_LIC

```
#define WDC_DRV_OPEN_REG_LIC 0x2
```

Register WinDriver license.

Definition at line 57 of file [wdc_lib.h](#).

WDC_ReadAddrBlock16

```
#define WDC_ReadAddrBlock16(  
    hDev,  
    dwAddrSpace,  
    dwOffset,  
    dwBytes,  
    pData,  
    options )
```

Value:

```
WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
    WDC_MODE_16, options)
```

WDC_ReadAddrBlock with 2 bytes mode.

Definition at line 950 of file [wdc_lib.h](#).

WDC_ReadAddrBlock32

```
#define WDC_ReadAddrBlock32(  
    hDev,  
    dwAddrSpace,  
    dwOffset,  
    dwBytes,
```

```
    pData,  
    options )
```

Value:

```
WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
    WDC_MODE_32, options)
```

WDC_ReadAddrBlock with 4 bytes mode.

Definition at line 956 of file [wdc_lib.h](#).

WDC_ReadAddrBlock64

```
#define WDC_ReadAddrBlock64(  
    hDev,  
    dwAddrSpace,  
    dwOffset,  
    dwBytes,  
    pData,  
    options )
```

Value:

```
WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
    WDC_MODE_64, options)
```

WDC_ReadAddrBlock with 8 bytes mode.

Definition at line 962 of file [wdc_lib.h](#).

WDC_ReadAddrBlock8

```
#define WDC_ReadAddrBlock8(  
    hDev,  
    dwAddrSpace,  
    dwOffset,  
    dwBytes,  
    pData,  
    options )
```

Value:

```
WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
    WDC_MODE_8, options)
```

WDC_ReadAddrBlock with 1 byte mode.

Definition at line 944 of file [wdc_lib.h](#).

WDC_ReadMem16

```
#define WDC_ReadMem16(  
    addr,  
    off )  *(volatile WORD *) ((UPTR)(addr) + (UPTR)(off))
```

reads 2 byte (16 bits) from a specified memory address.

The address is read directly in the calling context (user mode / kernel mode).

Definition at line 718 of file [wdc_lib.h](#).

WDC_ReadMem32

```
#define WDC_ReadMem32 (
    addr,
    off )  *(volatile UINT32 *) ((UPTR)(addr) + (UPTR)(off))
```

reads 4 byte (32 bits) from a specified memory address.

The address is read directly in the calling context (user mode / kernel mode).

Definition at line 724 of file [wdc_lib.h](#).

WDC_ReadMem64

```
#define WDC_ReadMem64 (
    addr,
    off )  *(volatile UINT64 *) ((UPTR)(addr) + (UPTR)(off))
```

reads 8 byte (64 bits) from a specified memory address.

The address is read directly in the calling context (user mode / kernel mode).

Definition at line 730 of file [wdc_lib.h](#).

WDC_ReadMem8

```
#define WDC_ReadMem8 (
    addr,
    off )  *(volatile BYTE *) ((UPTR)(addr) + (UPTR)(off))
```

Direct memory read/write macros.

Read/Write memory and I/O addresses

reads 1 byte (8 bits) from a specified memory address. The address is read directly in the calling context (user mode / kernel mode).

Definition at line 713 of file [wdc_lib.h](#).

WDC_SIZE_16

```
#define WDC_SIZE_16 ((DWORD)sizeof(WORD))
```

Definition at line 132 of file [wdc_lib.h](#).

WDC_SIZE_32

```
#define WDC_SIZE_32 ((DWORD)sizeof(UINT32))
```

Definition at line 133 of file [wdc_lib.h](#).

WDC_SIZE_64

```
#define WDC_SIZE_64 ((DWORD)sizeof(UINT64))
```

Definition at line 134 of file [wdc_lib.h](#).

WDC_SIZE_8

```
#define WDC_SIZE_8 ((DWORD)sizeof(BYTE))
```

Definition at line 131 of file [wdc_lib.h](#).

WDC_SLEEP_BUSY

```
#define WDC_SLEEP_BUSY 0
```

Sleep options.

Definition at line 109 of file [wdc_lib.h](#).

WDC_SLEEP_NON_BUSY

```
#define WDC_SLEEP_NON_BUSY SLEEP_NON_BUSY
```

Definition at line 110 of file [wdc_lib.h](#).

WDC_WriteAddrBlock16

```
#define WDC_WriteAddrBlock16( hDev,  
                           dwAddrSpace,  
                           dwOffset,  
                           dwBytes,  
                           pData,  
                           options )
```

Value:

```
    WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
                       WDC_MODE_16, options)
```

WDC_WriteAddrBlock with 2 bytes mode.

Definition at line 974 of file [wdc_lib.h](#).

WDC_WriteAddrBlock32

```
#define WDC_WriteAddrBlock32( hDev,  
                           dwAddrSpace,  
                           dwOffset,  
                           dwBytes,  
                           pData,  
                           options )
```

Value:

```
    WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
                       WDC_MODE_32, options)
```

WDC_WriteAddrBlock with 4 bytes mode.

Definition at line 980 of file [wdc_lib.h](#).

WDC_WriteAddrBlock64

```
#define WDC_WriteAddrBlock64( hDev,  
                           dwAddrSpace,  
                           dwOffset,  
                           dwBytes,  
                           pData,  
                           options )
```

Value:

```
    WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
                       WDC_MODE_64, options)
```

WDC_WriteAddrBlock with 8 bytes mode.

Definition at line 986 of file [wdc_lib.h](#).

WDC_WriteAddrBlock8

```
#define WDC_WriteAddrBlock8(  
    hDev,  
    dwAddrSpace,  
    dwOffset,  
    dwBytes,  
    pData,  
    options )
```

Value:

```
    WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \  
        WDC_MODE_8, options)
```

WDC_WriteAddrBlock with 1 byte mode.

Definition at line 968 of file [wdc_lib.h](#).

WDC_WriteMem16

```
#define WDC_WriteMem16(  
    addr,  
    off,  
    val) *(volatile WORD *)(((UPTR)(addr) + (UPTR)(off))) = (val)
```

writes 2 byte (16 bits) to a specified memory address.

The address is written to directly in the calling context (user mode / kernel mode)

Definition at line 743 of file [wdc_lib.h](#).

WDC_WriteMem32

```
#define WDC_WriteMem32(  
    addr,  
    off,  
    val) *(volatile UINT32 *)(((UPTR)(addr) + (UPTR)(off))) = (val)
```

writes 4 byte (32 bits) to a specified memory address.

The address is written to directly in the calling context (user mode / kernel mode)

Definition at line 749 of file [wdc_lib.h](#).

WDC_WriteMem64

```
#define WDC_WriteMem64(  
    addr,  
    off,  
    val) *(volatile UINT64 *)(((UPTR)(addr) + (UPTR)(off))) = (val)
```

writes 8 byte (64 bits) to a specified memory address.

The address is written to directly in the calling context (user mode / kernel mode)

Definition at line 755 of file [wdc_lib.h](#).

WDC_WriteMem8

```
#define WDC_WriteMem8(  
    addr,  
    off,  
    val) *(volatile BYTE *)(((UPTR)(addr) + (UPTR)(off))) = (val)
```

writes 1 byte (8 bits) to a specified memory address.

The address is written to directly in the calling context (user mode / kernel mode)

Definition at line [737](#) of file [wdc_lib.h](#).

Typedef Documentation

WDC_ADDR_SIZE

`typedef DWORD WDC_ADDR_SIZE`

Definition at line [135](#) of file [wdc_lib.h](#).

WDC_DBG_OPTIONS

`typedef DWORD WDC_DBG_OPTIONS`

Definition at line [106](#) of file [wdc_lib.h](#).

WDC_DEVICE_HANDLE

`typedef void* WDC_DEVICE_HANDLE`

Handle to device information struct.

Definition at line [33](#) of file [wdc_lib.h](#).

WDC_DRV_OPEN_OPTIONS

`typedef DWORD WDC_DRV_OPEN_OPTIONS`

Definition at line [71](#) of file [wdc_lib.h](#).

WDC_SLEEP_OPTIONS

`typedef DWORD WDC_SLEEP_OPTIONS`

Definition at line [111](#) of file [wdc_lib.h](#).

Enumeration Type Documentation

WDC_ADDR_MODE

`enum WDC_ADDR_MODE`

Enumerator

<code>WDC_MODE_8</code>	
<code>WDC_MODE_16</code>	
<code>WDC_MODE_32</code>	
<code>WDC_MODE_64</code>	

Definition at line [137](#) of file [wdc_lib.h](#).

WDC_ADDR_RW_OPTIONS

`enum WDC_ADDR_RW_OPTIONS`

Read/write address options.

Enumerator

WDC_ADDR_RW_DEFAULT	Default: memory resource - direct access; autoincrement on block transfers.
WDC_ADDR_RW_NO_AUTOINC	Hold device address constant while reading/writing a block.

Definition at line 123 of file [wdc_lib.h](#).

WDC_DIRECTION

enum **WDC_DIRECTION**

Enumerator

WDC_WRITE	
WDC_READ	
WDC_READ_WRITE	

Definition at line 116 of file [wdc_lib.h](#).

Function Documentation**WDC_AddrSpaceIsActive()**

```
BOOL DLLCALLCONV WDC_AddrSpaceIsActive (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace )
```

Checks if the specified memory or I/O address space is active ,i.e., if its size is not zero.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen() .
in	<i>dwAddrSpace</i>	The memory or I/O address space to look for.

Returns

Returns TRUE if the specified address space is active; otherwise returns FALSE.

WDC_CallKerPlug()

```
DWORD DLLCALLCONV WDC_CallKerPlug (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwMsg,
    _Inout_ PVOID pData,
    _Outptr_ PDWORD pdwResult )
```

Sends a message from a user-mode application to a Kernel PlugIn driver.

Send Kernel Plugin messages

The function passes a message ID from the application to the Kernel PlugIn's KP_Call function, which should be implemented to handle the specified message ID, and returns the result from the Kernel PlugIn to the user-mode application.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwMsg</i>	A message ID to pass to the Kernel PlugIn driver (specifically to KP_Call)
in,out	<i>pData</i>	Pointer to data to pass between the Kernel PlugIn driver and the user-mode application
out	<i>pdwResult</i>	Result returned by the Kernel PlugIn driver (KP_Call) for the operation performed in the kernel as a result of the message that was sent

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_CardCleanupSetup()

```
DWORD WDC_CardCleanupSetup (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ WD_TRANSFER * pTransCmds,
    _In_ DWORD dwCmds,
    _In_ BOOL fForceCleanup )
```

Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:

Set card cleanup commands

- 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.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>pTransCmds</i>	Pointer to an array of cleanup transfer commands to be performed.
in	<i>dwCmds</i>	Number of cleanup commands in the Cmd array.
in	<i>fForceCleanup</i>	If FALSE: The cleanup transfer commands (Cmd) will be performed in either of the following cases: When the application exits abnormally. When the application exits normally without closing the card by calling one of the WDC_xxxDeviceClose() functions

If TRUE: The cleanup transfer commands will be performed both in the two cases described above, as well as in the following case: When the relevant WD_xxxDeviceClose() function is called for the card.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_DMABufGet()

```
DWORD DLLCALLCONV WDC_DMABufGet (
    _In_ DWORD hDma,
    _Outptr_ WD_DMA ** ppDma )
```

Retrieves a contiguous DMA buffer which was allocated by another process.

Parameters

in	<i>hDma</i>	DMA buffer handle.
out	<i>ppDma</i>	Pointer to a pointer to a DMA buffer information structure, which is associated with hDma.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA functions, please refer to [11.2. Performing Direct Memory Access \(DMA\)](#)

```
static void ipc_msg_event_cb(WDS_IPC_MSG_RX *pIpcRxMsg, void *pData)
{
    printf("\n\nReceived an IPC message:\n"
        "msgID [0x%lx], msgData [0x%llx] from process [0x%lx]\n",
        pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData, pIpcRxMsg->dwSenderUID);
    /* In this example, the hDma was sent by the other process and saved it
     * in pIpcRxMsg->qwMsgData. The current process gets the buffer by the IPC
     * message event callback */
    switch (pIpcRxMsg->dwMsgID)
    {
        case IPC_MSG_CONTIG_DMA_BUFFER_READY:
        {
            DWORD dwStatus;
            WD_DMA *pDma = NULL;
            printf("\nThis is a DMA buffer, getting it...\n");
            dwStatus = WDC_DMABufGet((DWORD)pIpcRxMsg->qwMsgData, &pDma);
            if (WD_STATUS_SUCCESS != dwStatus)
            {
                printf("ipc_msg_event_cb: Failed getting DMA buffer. "
                    "Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
                return;
            }
            printf("Got a DMA buffer. UserAddr [%p], "
                "pPhysicalAddr [0x%"PRI64"x], size [%ld(0x%lx)]\n",
                pDma->pUserAddr, pDma->Page[0].pPhysicalAddr,
                pDma->Page[0].dwBytes, pDma->Page[0].dwBytes);
            /* For sample purpose we immediately release the buffer */
            WDC_DMABufUnlock(pDma);
        }
        break;
    default:
        printf("Unknown IPC type. msgID [0x%lx], msgData [0x%llx] from "
            "process [0x%lx]\n\n", pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData,
            pIpcRxMsg->dwSenderUID);
    }
}
```

WDC_DMABufUnlock()

```
WORD DLLCALLCONV WDC_DMABufUnlock (
    _In_ WD_DMA * pDma )
```

Unlocks and frees the memory allocated for a DMA buffer by a previous call to [WDC_DMAContigBufLock\(\)](#), [WDC_DMASGBufLock\(\)](#) or [WDC_DMAReservedBufLock\(\)](#)

Parameters

in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
----	-------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA functions, please refer to [11.2. Performing Direct Memory Access \(DMA\)](#)

WDC_DMAContigBufLock()

```
DWORD DLLCALLCONV WDC_DMAContigBufLock (
    _In_ WDC_DEVICE_HANDLE hDev,
    _Outptr_ PVOID * ppBuf,
    _In_ DWORD dwOptions,
    _In_ DWORD dwDMABufSize,
    _Outptr_ WD_DMA ** ppDma )
```

Allocates a contiguous DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.

DMA (Direct Memory Access)

Parameters

in	hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
out	ppBuf	Pointer to a pointer to be filled by the function with the user-mode mapped address of the allocated DMA buffer
in	dwOptions	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): DMA_FROM_DEVICE: Synchronize the DMA buffer for transfers from the device to memory. DMA_TO_DEVICE: Synchronize the DMA buffer for transfers from memory to the device. DMA_TO_FROM_DEVICE: Synchronize the DMA buffer for transfers in both directions i.e., from the device to memory and from memory to the device (<=> DMA_FROM_DEVICE DMA_TO_DEVICE). DMA_ALLOW_CACHE: Allow caching of the memory. DMA_KBUF_BELOW_16M: Allocate the physical DMA buffer within the lower 16MB of the main memory. DMA_ALLOW_64BIT_ADDRESS: Allow allocation of 64-bit DMA addresses. DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH: When using this flag, the width of the address must be entered in the fourth byte of dwOptions and then the allocated address will be limited to this width. Linux: works with contiguous buffers only. DMA_GET_PREALLOCATED_BUFFERS_ONLY: Windows: Only preallocated buffer is allowed.
in	dwDMABufSize	The size (in bytes) of the DMA buffer.
out	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() when the DMA buffer is no longer needed.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC contiguous-buffer DMA implementaiton, please refer to [11.2.2. Implementing Contiguous-Buffer DMA](#)

WDC_DMAReservedBufLock()

```
DWORD DLLCALLCONV WDC_DMAReservedBufLock (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ PHYS_ADDR qwAddr,
    _Outptr_ PVOID * ppBuf,
    _In_ DWORD dwOptions,
    _In_ DWORD dwDMABufSize,
    _Outptr_ WD_DMA ** ppDma )
```

Locks a physical reserved memory buffer for DMA and returns the corresponding user mode address of locked DMA buffer.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>qwAddr</i>	Physical address of the reserved buffer to lock
out	<i>ppBuf</i>	Pointer to a pointer to be filled by the function with the user-mode mapped address of the locked DMA buffer
in	<i>dwOptions</i>	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): DMA_FROM_DEVICE: Synchronize the DMA buffer for transfers from the device to memory. DMA_TO_DEVICE: Synchronize the DMA buffer for transfers from memory to the device. DMA_TO_FROM_DEVICE: Synchronize the DMA buffer for transfers in both directions i.e., from the device to memory and from memory to the device (<=> DMA_FROM_DEVICE DMA_TO_DEVICE). DMA_ALLOW_64BIT_ADDRESS: Allow allocation of 64-bit DMA addresses. This flag is supported on Windows and Linux.
in	<i>dwDMABufSize</i>	The size (in bytes) of the DMA buffer
out	<i>ppDma</i>	Pointer to a pointer to a DMA buffer information structure, which is allocated by the function. The pointer to this structure (* <i>ppDma</i>) should be passed to WDC_DMABufUnlock() when the DMA buffer is no longer needed.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA functions, please refer to [11.2. Performing Direct Memory Access \(DMA\)](#)

WDC_DMASGBufLock()

```
DWORD DLLCALLCONV WDC_DMASGBufLock (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ PVOID pBuf,
    _In_ DWORD dwOptions,
    _In_ DWORD dwDMABufSize,
    _Outptr_ WD_DMA ** ppDma )
```

Locks a pre-allocated user-mode memory buffer for DMA and returns the corresponding physical mappings of the locked DMA pages.

On Windows the function also returns a kernel-mode mapping of the buffer.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>pBuf</i>	Pointer to a user-mode buffer to be mapped to the allocated physical DMA buffer(s)
in	<i>dwOptions</i>	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): DMA_FROM_DEVICE: Synchronize the DMA buffer for transfers from the device to memory. DMA_TO_DEVICE: Synchronize the DMA buffer for transfers from memory to the device. DMA_TO_FROM_DEVICE: Synchronize the DMA buffer for transfers in both directions i.e., from the device to memory and from memory to the device (<=> DMA_FROM_DEVICE DMA_TO_DEVICE). DMA_ALLOW_CACHE: Allow caching of the memory. DMA_ALLOW_64BIT_ADDRESS: Allow allocation of 64-bit DMA addresses.
in	<i>dwDMABufSize</i>	The size (in bytes) of the DMA buffer.
out	<i>ppDma</i>	Pointer to a pointer to a DMA buffer information structure, which is allocated by the function. The pointer to this structure (* <i>ppDma</i>) should be passed to WDC_DMABufUnlock() when the DMA buffer is no longer needed.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC Scatter/Gather DMA implementaiton, please refer to [11.2.1. Implementing Scatter/Gather DMA](#)

WDC_DMASyncCpu()

```
DWORD DLLCALLCONV WDC_DMASyncCpu (
    _In_ WD_DMA * pDma )
```

Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.

Parameters

in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
----	-------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA functions, please refer to [11.2. Performing Direct Memory Access \(DMA\)](#)

WDC_DMASyncIo()

```
DWORD DLLCALLCONV WDC_DMASyncIo (
    _In_ WD_DMA * pDma )
```

Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.

Parameters

in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
----	-------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA functions, please refer to [11.2. Performing Direct Memory Access \(DMA\)](#)

WDC_DMATransactionContigInit()

```
DWORD DLLCALLCONV WDC_DMATransactionContigInit (
    _In_ WDC_DEVICE_HANDLE hDev,
    _Outptr_ PVOID * ppBuf,
    _In_ DWORD dwOptions,
    _In_ DWORD dwDMABufSize,
    _Outptr_ WD_DMA ** ppDma,
    _In_ WDC_INTERRUPT_PARAMS * pInterruptParams,
    _In_ DWORD dwAlignment )
```

Initializes the transaction, allocates a contiguous DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
out	<i>ppBuf</i>	Pointer to a pointer to be filled by the function with the user-mode mapped address of the allocated DMA buffer.
in	<i>dwOptions</i>	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): DMA_FROM_DEVICE: Synchronize the DMA buffer for transfers from the device to memory. DMA_TO_DEVICE: Synchronize the DMA buffer for transfers from memory to the device. DMA_TO_FROM_DEVICE: Synchronize the DMA buffer for transfers in both directions i.e., from the device to memory and from memory to the device (<=> DMA_FROM_DEVICE DMA_TO_DEVICE). DMA_ALLOW_CACHE: Allow caching of the memory. DMA_ALLOW_64BIT_ADDRESS: Allow allocation of 64-bit DMA addresses.
in	<i>dwDMABufSize</i>	The size (in bytes) of the DMA buffer.
out	<i>ppDma</i>	Pointer to a pointer to a DMA buffer information structur, which is allocated by the function. The pointer to this structure (* <i>ppDma</i>) should be passed to WDC_DMATransactionUninit() when the DMA buffer is no longer needed.
in	<i>pInterruptParams</i>	WDC_DMATransactionContigInit() invokes WDC_IntEnable() with the relevant parameters from the structure (WDC_INTERRUPT_PARAMS). No action will be taken if this parameter is NULL.
in	<i>dwAlignment</i>	This value specifies the alignment requirement for the contiguous buffer.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

WDC_DMATransactionExecute()

```
DWORD DLLCALLCONV WDC_DMATransactionExecute (
    _Inout_ WD_DMA * pDma,
    _In_ DMA_TRANSACTION_CALLBACK funcDMATransactionCallback,
    _In_ PVOID DMATransactionCallbackCtx )
```

Begins the execution of a specified DMA transaction.

Parameters

in,out	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) * <i>ppDma</i> returned by these functions
in	<i>funcDMATransactionCallback</i>	If the execution is completed successfully, this callback function will be called directly with <i>DMATransactionCallbackCtx</i> as context. No action will be taken if this parameter is NULL.
in	<i>DMATransactionCallbackCtx</i>	Pointer to a DMA transaction callback context

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

WDC_DMATransactionRelease()

```
DWORD DLLCALLCONV WDC_DMATransactionRelease (
    _In_ WD_DMA * pDma )
```

Terminates a specified DMA transaction without deleting the associated [WD_DMA](#) transaction structure.

Parameters

in	pDma	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
-----------	-------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

WDC_DMATransactionSGInit()

```
DWORD DLLCALLCONV WDC_DMATransactionSGInit (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ PVOID pBuf,
    _In_ DWORD dwOptions,
    _In_ DWORD dwDMABufSize,
    _Outptr_ WD_DMA ** ppDma,
    _In_ WDC_INTERRUPT_PARAMS * pInterruptParams,
    _In_ DWORD dwMaxTransferSize,
    _In_ DWORD dwTransferElementSize )
```

Initializes the transaction and locks a pre-allocated user-mode memory buffer for DMA.

Parameters

in	hDev	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	pBuf	Pointer to a user-mode buffer to be mapped to the allocated physical DMA buffer(s).
in	dwOptions	A bit mask of any of the following flags (defined in an enumeration in windrvr.h): DMA_FROM_DEVICE: Synchronize the DMA buffer for transfers from the device to memory. DMA_TO_DEVICE: Synchronize the DMA buffer for transfers from memory to the device. DMA_TO_FROM_DEVICE: Synchronize the DMA buffer for transfers in both directions i.e., from the device to memory and from memory to the device (<=> DMA_FROM_DEVICE DMA_TO_DEVICE). DMA_ALLOW_CACHE: Allow caching of the memory. DMA_ALLOW_64BIT_ADDRESS: Allow allocation of 64-bit DMA addresses.
in	dwDMABufSize	The size (in bytes) of the DMA buffer.
out	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_DMATransactionUninit() when the DMA buffer is no longer needed.

Parameters

in	<i>pInterruptParams</i>	WDC_DMATransactionSGInit() invokes WDC_IntEnable() with the relevant parameters from the structure (WDC_INTERRUPT_PARAMS). No action will be taken if this parameter is NULL.
in	<i>dwMaxTransferSize</i>	The maximum size for each of the transfers.
in	<i>dwTransferElementSize</i>	The size (in bytes) of the DMA transfer element (descriptor).

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

[WDC_DMATransactionUninit\(\)](#)

```
DWORD DLLCALLCONV WDC_DMATransactionUninit (
    _In_ WD_DMA * pDma )
```

Unlocks and frees the memory allocated for a DMA buffer transaction by a previous call to [WDC_DMATransactionContigInit\(\)](#) or [WDC_DMATransactionSGInit\(\)](#)

Parameters

in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
----	-------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

[WDC_DMATransferCompletedAndCheck\(\)](#)

```
DWORD DLLCALLCONV WDC_DMATransferCompletedAndCheck (
    _Inout_ WD_DMA * pDma,
    _In_ BOOL fRunCallback )
```

Notifies WinDriver that a device's DMA transfer operation is completed.

Parameters

in,out	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WDC_DMATransactionContigInit() (for a Contiguous DMA Buffer Transaction) or WDC_DMATransactionSGInit() (for a Scatter/Gather DMA buffer Transaction) *ppDma returned by these functions
in	<i>fRunCallback</i>	If this value is TRUE and the transaction is not over (there are more transfers to be made) the callback function that was previously provided as a callback to the WDC_DMATransactionExecute() function will be invoked. Otherwise, the callback function will not be invoked.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed usage of the WDC_DMA Transactions functions, please refer to [11.3. Performing Direct Memory Access \(DMA\) transactions](#)

WDC_DriverClose()

```
DWORD DLLCALLCONV WDC_DriverClose (
    void )
```

Closes the WDC WinDriver handle (acquired and stored by a previous call to [WDC_DriverOpen\(\)](#)) and uninitialized 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.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    UINT32 u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        printf("Failed to set the driver name for WDC library.\n");
        return WD_INTERNAL_ERROR;
    }
    dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize debug options for WDC library.\n"
               "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    /* Open a handle on the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    BZERO(deviceInfo);
    deviceInfo.pciSlot = slot;
    dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed retrieving the device's resources "
               "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    /* Open a device handle */
    dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    /* Read/Write memory examples: */
    /* Check BYTE */
    dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
                                   &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
                                 &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    /* Check WORD */
    dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
                                   &wData, WDC_ADDR_RW_DEFAULT);
```

```

    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
        &wData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
Exit:
    if (hDev)
    {
        dwStatus = WDC_PciDeviceClose(hDev);
        if (WD_STATUS_SUCCESS != dwStatus)
        {
            printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
                "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
        }
    }
    WDC_DriverClose();
}

```

WDC_DriverOpen()

```

DWORD DLICALLCONV WDC_DriverOpen (
    _In_ WDC_DRV_OPEN_OPTIONS openOptions,
    _In_ const CHAR * pcLicense )

```

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.

Parameters

in	<i>openOptions</i>	A mask of any of the supported open flags, which determines the initialization actions that will be performed by the function.
in	<i>pcLicense</i>	WinDriver license registration string. This argument is ignored if the WDC_DRV_OPEN_REG_LIC flag is not set in the <i>openOptions</i> 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.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

This function is currently only supported from the user mode. This function is supported only for Windows and Linux.

```

int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */

```

```
WD_PCI_CARD_INFO deviceInfo;
BYTE bData = 0x1;
WORD wData = 0x2;
UINT32 u32Data = 0x3;
UINT64 u64Data = 0x4;
/* Make sure to fully initialize WinDriver! */
if (!WD_DriverName(DEFAULT_DRIVER_NAME))
{
    printf("Failed to set the driver name for WDC library.\n");
    return WD_INTERNAL_ERROR;
}
dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize debug options for WDC library.\n"
        "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
/* Open a handle to the driver and initialize the WDC library */
dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
BZERO(deviceInfo);
deviceInfo.pciSlot = slot;
dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed retrieving the device's resources "
        "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Open a device handle */
dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Read/Write memory examples: */
/* Check BYTE */
dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check WORD */
dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
Exit:
if (hDev)
{
    dwStatus = WDC_PciDeviceClose(hDev);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
            "Error 0x%lx - %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
}
```

```

        "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
    WDC_DriverClose();
}

```

WDC_Err()

```
void DLLCALLCONV WDC_Err (
    const CHAR * format,
    ...
)
```

Displays debug error messages according to the WDC debug options.

- see [WDC_DBG_OPTIONS](#) and [WDC_SetDebugOptions\(\)](#)

Parameters

in	<i>format</i>	Format-control string, which contains the error message to display. The string is limited to 256 characters (CHAR)
in	...	Optional arguments for the format string

Returns

None

WDC_EventIsRegistered()

```
BOOL DLLCALLCONV WDC_EventIsRegistered (
    _In_ WDC_DEVICE_HANDLE hDev )

```

Checks if the application is currently registered to receive Plug-and-Play and power management notifications for the device.

Parameters

in	<i>hDev</i>	Handle to a Plug-and-Play WDC device, returned by WDC_PciDeviceOpen()
----	-------------	---

Returns

Returns TRUE if the application is currently registered to receive Plug-and-Play and power management notifications for the device; otherwise returns FALSE.

```
//WinDriver Callback functions must use DLLCALLCONV macro
static void DLLCALLCONV EventHandlerExample(WDC_DEVICE_HANDLE hDev,
    DWORD dwAction)
{
    printf("\nReceived event notification (device handle 0x%p): ", hDev);
    switch (dwAction)
    {
        case WD_INSERT:
            printf("WD_INSERT\n");
            break;
        case WD_REMOVE:
            printf("WD_REMOVE\n");
            break;
        case WD_POWER_CHANGED_D0:
            printf("WD_POWER_CHANGED_D0\n");
            break;
        case WD_POWER_CHANGED_D1:
            printf("WD_POWER_CHANGED_D1\n");
            break;
        case WD_POWER_CHANGED_D2:
            printf("WD_POWER_CHANGED_D2\n");
            break;
        case WD_POWER_CHANGED_D3:
            printf("WD_POWER_CHANGED_D3\n");
            break;
    }
}
```

```
    printf("WD_POWER_CHANGED_D3\n");
    break;
case WD_POWER_SYSTEM_WORKING:
    printf("WD_POWER_SYSTEM_WORKING\n");
    break;
case WD_POWER_SYSTEM_SLEEPING1:
    printf("WD_POWER_SYSTEM_SLEEPING1\n");
    break;
case WD_POWER_SYSTEM_SLEEPING2:
    printf("WD_POWER_SYSTEM_SLEEPING2\n");
    break;
case WD_POWER_SYSTEM_SLEEPING3:
    printf("WD_POWER_SYSTEM_SLEEPING3\n");
    break;
case WD_POWER_SYSTEM_HIBERNATE:
    printf("WD_POWER_SYSTEM_HIBERNATE\n");
    break;
case WD_POWER_SYSTEM_SHUTDOWN:
    printf("WD_POWER_SYSTEM_SHUTDOWN\n");
    break;
default:
    printf("0x%lx\n", dwAction);
    break;
}
}

DWORD EventRegisterExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus;
    DWORD dwActions = WD_ACTIONS_ALL;
    /* Check whether the event is already registered */
    if (WDC_EventIsRegistered(hDev))
    {
        return WD_OPERATION_ALREADY_DONE;
    }
    /* Register the event */
    dwStatus = WDC_EventRegister(hDev, dwActions, EventHandlerExample, hDev,
        WDC_IS_KP(hDev));
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to register events. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    printf("Events registered\n");
    // ...
    // Some more code
    // ...
    /* Cleanup */
    dwStatus = WDC_EventUnregister(hDev);
    return dwStatus;
}
```

WDC_EventRegister()

```
WORD DLLCALLCONV WDC_EventRegister (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ WORD dwActions,
    _In_ EVENT_HANDLER funcEventHandler,
    _In_ PVOID pData,
    _In_ WORD fUseKP )
```

Registers the application to receive Plug-and-Play and power management events notifications for the device.

Plug-and-play and power management events

Parameters

in	<i>hDev</i>	Handle to a Plug-and-Play WDC device, returned by WDC_PciDeviceOpen()
----	-------------	---

Parameters

in	<i>dwActions</i>	A bit mask of flags indicating which events to register to: Plug-and-Play events: WD_INSERT Device inserted WD_REMOVE Device removed Device power state change events: 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 Systems power state: WD_POWER_SYSTEM_SLEEPING1 Fully on but sleeping WD_POWER_SYSTEM_SLEEPING2 CPU off, memory on, PCI on WD_POWER_SYSTEM_SLEEPING3 CPU off, Memory is in refresh, PCI on aux power WD_POWER_SYSTEM_HIBERNATE OS saves context before shutdown WD_POWER_SYSTEM_SHUTDOWN No context saved
in	<i>funcEventHandler</i>	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 .)
in	<i>pData</i>	Data for the user-mode event handler callback routine (funcEventHandler)
in	<i>fUseKP</i>	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 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). If this function returns TRUE, the user-mode events handler callback function (funcEventHandler) will be called when the kernel-mode event processing is completed. 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.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise.

```
//WinDriver Callback functions must use DLLCALLCONV macro
static void DLLCALLCONV EventHandlerExample(WDC_DEVICE_HANDLE hDev,
    DWORD dwAction)
{
    printf("\nReceived event notification (device handle 0x%p): ", hDev);
    switch (dwAction)
    {
        case WD_INSERT:
            printf("WD_INSERT\n");
            break;
        case WD_REMOVE:
            printf("WD_REMOVE\n");
            break;
        case WD_POWER_CHANGED_D0:
            printf("WD_POWER_CHANGED_D0\n");
            break;
        case WD_POWER_CHANGED_D1:
            printf("WD_POWER_CHANGED_D1\n");
            break;
        case WD_POWER_CHANGED_D2:
            printf("WD_POWER_CHANGED_D2\n");
            break;
        case WD_POWER_CHANGED_D3:
            printf("WD_POWER_CHANGED_D3\n");
            break;
        case WD_POWER_SYSTEM_WORKING:
            printf("WD_POWER_SYSTEM_WORKING\n");
            break;
        case WD_POWER_SYSTEM_SLEEPING1:
            printf("WD_POWER_SYSTEM_SLEEPING1\n");
            break;
        case WD_POWER_SYSTEM_SLEEPING2:
            printf("WD_POWER_SYSTEM_SLEEPING2\n");
            break;
        case WD_POWER_SYSTEM_SLEEPING3:
            printf("WD_POWER_SYSTEM_SLEEPING3\n");
            break;
        case WD_POWER_SYSTEM_HIBERNATE:
            printf("WD_POWER_SYSTEM_HIBERNATE\n");
            break;
        case WD_POWER_SYSTEM_SHUTDOWN:
            printf("WD_POWER_SYSTEM_SHUTDOWN\n");
            break;
    }
}
```

```

    default:
        printf("0x%lx\n", dwAction);
        break;
    }
}

DWORD EventRegisterExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus;
    DWORD dwActions = WD_ACTIONS_ALL;
    /* Check whether the event is already registered */
    if (WDC_EventIsRegistered(hDev))
    {
        return WD_OPERATION_ALREADY_DONE;
    }
    /* Register the event */
    dwStatus = WDC_EventRegister(hDev, dwActions, EventHandlerExample, hDev,
        WDC_IS_KP(hDev));
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to register events. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    printf("Events registered\n");
    // ...
    // Some more code
    // ...
    /* Cleanup */
    dwStatus = WDC_EventUnregister(hDev);
    return dwStatus;
}

```

WDC_EventUnregister()

DWORD **DLLCALLCONV** WDC_EventUnregister (
 In WDC_DEVICE_HANDLE *hDev*)

Unregisters an application from a receiving Plug-and-Play and power management notifications for a device, pursuant to a previous call to [WDC_EventRegister\(\)](#)

Parameters

in	<i>hDev</i>	Handle to a Plug-and-Play WDC device, returned by WDC_PciDeviceOpen()
-----------	-------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```

//WinDriver Callback functions must use DLLCALLCONV macro
static void DLLCALLCONV EventHandlerExample(WDC_DEVICE_HANDLE hDev,
    DWORD dwAction)
{
    printf("\nReceived event notification (device handle 0x%p): ", hDev);
    switch (dwAction)
    {
        case WD_INSERT:
            printf("WD_INSERT\n");
            break;
        case WD_REMOVE:
            printf("WD_REMOVE\n");
            break;
        case WD_POWER_CHANGED_D0:
            printf("WD_POWER_CHANGED_D0\n");
            break;
        case WD_POWER_CHANGED_D1:
            printf("WD_POWER_CHANGED_D1\n");
            break;
        case WD_POWER_CHANGED_D2:
            printf("WD_POWER_CHANGED_D2\n");
            break;
        case WD_POWER_CHANGED_D3:
            printf("WD_POWER_CHANGED_D3\n");
            break;
        case WD_POWER_SYSTEM_WORKING:
            printf("WD_POWER_SYSTEM_WORKING\n");
            break;
        case WD_POWER_SYSTEM_SLEEPING1:
            printf("WD_POWER_SYSTEM_SLEEPING1\n");
            break;
    }
}

```

```

        case WD_POWER_SYSTEM_SLEEPING2:
            printf("WD_POWER_SYSTEM_SLEEPING2\n");
            break;
        case WD_POWER_SYSTEM_SLEEPING3:
            printf("WD_POWER_SYSTEM_SLEEPING3\n");
            break;
        case WD_POWER_SYSTEM_HIBERNATE:
            printf("WD_POWER_SYSTEM_HIBERNATE\n");
            break;
        case WD_POWER_SYSTEM_SHUTDOWN:
            printf("WD_POWER_SYSTEM_SHUTDOWN\n");
            break;
        default:
            printf("0x%lx\n", dwAction);
            break;
    }
}

DWORD EventRegisterExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus;
    DWORD dwActions = WD_ACTIONS_ALL;
    /* Check whether the event is already registered */
    if (WDC_EventIsRegistered(hDev))
    {
        return WD_OPERATION_ALREADY_DONE;
    }
    /* Register the event */
    dwStatus = WDC_EventRegister(hDev, dwActions, EventHandlerExample, hDev,
        WDC_IS_KP(hDev));
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to register events. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    printf("Events registered\n");
    // ...
    // Some more code
    // ...
    /* Cleanup */
    dwStatus = WDC_EventUnregister(hDev);
    return dwStatus;
}

```

WDC_GetBusType()

```
WD_BUS_TYPE DLLCALLCONV WDC_GetBusType (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Returns the device's bus type: WD_BUS_PCI, WD_BUS_ISA or WD_BUS_UNKNOWN.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen().
----	-------------	--

Returns

Returns the device's bus type.

WDC_GetDevContext()

```
PVOID DLLCALLCONV WDC_GetDevContext (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Returns the device's user context information.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen().
----	-------------	--

Returns

Pointer to the device's user context, or NULL if no context has been set.

WDC_GetWDHandle()

```
HANDLE DLLCALLCONV WDC_GetWDHandle (
    void )
```

Get a handle to WinDriver.

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 low-level WD_xxx API, if such APIs are used from your code.

Returns

Returns a handle to WinDriver's kernel module (or INVALID_HANDLE_VALUE in case of a failure), which is required by the basic WD_xxx WinDriver PCI/ISA API

WDC_IntDisable()

```
DWORD DLLCALLCONV WDC_IntDisable (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Disables interrupt handling for the device, pursuant to a previous call to [WDC_IntEnable\(\)](#)

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
----	-------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed information about interrupt handling, please refer to [10.1. Handling Interrupts](#)

For a sample user-mode interrupt handling code, please refer to [10.1.8. Sample User-Mode WinDriver Interrupt Handling Code](#)

WDC_IntEnable()

```
DWORD DLLCALLCONV WDC_IntEnable (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ WD_TRANSFER * pTransCmds,
    _In_ DWORD dwNumCmds,
    _In_ DWORD dwOptions,
    _In_ INT_HANDLER funcIntHandler,
    _In_ PVOID pData,
    _In_ BOOL fUseKP )
```

Enables interrupt handling for the device.

On Linux and Windows, when attempting to enable interrupts for a PCI device that supports Extended Message-Signaled Interrupts (MSI-X) or Message-Signaled Interrupts (MSI) (and was installed with a relevant INF file on Windows), the function first tries to enable MSI-X or MSI; if this fails, or if the target OS does not support MSI/MSI-X, the function attempts to enable legacy level-sensitive interrupts (if supported by the device). On Linux, you can use the function's dwOptions parameter to specify the types of PCI interrupts that may be enabled for the device (see the explanation in the parameter description). For other types of hardware (PCI with no MSI/MSI-X support / ISA), the function attempts to enable the legacy interrupt type supported by the device (Level Sensitive / Edge Triggered)

If the caller selects to handle the interrupts in the kernel, using a Kernel Plugin driver, the Kernel Plugin KP_IntAtIrql (legacy interrupts) or KP_IntAtIrqlMSI (MSI/MSI-X) function, which runs at high interrupt request level (IRQL), 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_IntAtDpc or KP_IntAtDpcMSI function completes its execution.

When handling level-sensitive interrupts (such as legacy 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, so the transfer commands in the call to [WDC_IntEnable\(\)](#) are not required (although they can still be used).

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 (legacy interrupts) or KP_IntAtDpcMSI (MSI/MSI-X) will determine how many times (if at all) the user-mode interrupt handler will be called (provided KP_IntAtDpc or KP_IntAtDpcMSI itself is executed which is determined by the return value of the Kernel PlugIn KP_IntAtIrql or KP_IntAtIrqlMSI function).

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>pTransCmds</i>	An array of transfer commands information structures 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. <ul style="list-style-type: none"> • NOTE: Memory allocated for the transfer commands must remain available until the interrupts are disabled. When handling level-sensitive interrupts (such as legacy PCI interrupts) without a Kernel PlugIn, you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received. • For an explanation on how to set the transfer commands, refer to the description of WD_TRANSFER in Section B.7.10, and to the explanation in Section 9.3.6.
in	<i>dwNumCmds</i>	Number of transfer commands in the <i>pTransCmds</i> array
in	<i>dwOptions</i>	A bit mask of interrupt handling flags can be set to zero for no options, or to a combination of any of the following flags: 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. The user will be able to access the data from his user-mode interrupt handler routine (<i>funcIntHandler</i>). <ul style="list-style-type: none"> • The following flags are applicable only to PCI interrupts on Linux. If set, these flags determine the types of interrupts that may be enabled for the device the function will attempt to enable only interrupts of the specified types, using the following precedence order, provided the type is reported as supported by the device: <ul style="list-style-type: none"> • INTERRUPT_MESSAGE_X: Extended Message-Signaled Interrupts (MSI-X) • INTERRUPT_MESSAGE: Message-Signaled Interrupts (MSI) • INTERRUPT_LEVEL_SENSITIVE Legacy level-sensitive interrupts • INTERRUPT_DONT_GET_MSI_MESSAGE: Do not read the msi message from the card.
in	<i>funcIntHandler</i>	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).
in	<i>pData</i>	Data for the user-mode interrupt handler callback routine (<i>funcIntHandler</i>)

Parameters

in	<i>fUseKP</i>	If TRUE The device's Kernel PlugIn driver's KP_IntAtIrql or KP_IntAtIrqlMSI function, which runs at high interrupt request level (IRQL), 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. 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 or KP_IntAtIrqlMSI completes its execution. If the high-IRQL handler returns TRUE, the Kernel PlugIn deferred interrupt processing routine KP_IntAtDpc or KP_IntAtDpcMSI 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. 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.
----	---------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

For more detailed information about interrupt handling, please refer to [10.1. Handling Interrupts](#)

For a sample user-mode interrupt handling code, please refer to [10.1.8. Sample User-Mode WinDriver Interrupt Handling Code](#)

WDC_IntIsEnabled()

```
BOOL DLLCALLCONV WDC_IntIsEnabled (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Checks if a device's interrupts are currently enabled.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
----	-------------	---

Returns

Returns TRUE if the device's interrupts are enabled; otherwise returns FALSE.

For more detailed information about interrupt handling, please refer to [10.1. Handling Interrupts](#)

For a sample user-mode interrupt handling code, please refer to [10.1.8. Sample User-Mode WinDriver Interrupt Handling Code](#)

WDC_IntType2Str()

```
const CHAR *DLLCALLCONV WDC_IntType2Str (
    _In_ DWORD dwIntType )
```

Converts interrupt type to string.

Parameters

in	<i>dwIntType</i>	Interrupt types bit-mask
----	------------------	--------------------------

Returns

Returns the string representation that corresponds to the specified numeric code.

For more detailed information about interrupt handling, please refer to [10.1. Handling Interrupts](#)

For a sample user-mode interrupt handling code, please refer to [10.1.8. Sample User-Mode WinDriver Interrupt Handling Code](#)

WDC_IsaDeviceClose()

```
DWORD DLLCALLCONV WDC_IsaDeviceClose (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Uninitializes a WDC ISA device structure and frees the memory allocated for it.

Parameters

in	hDev	Handle to a WDC ISA device structure, returned by WDC_IsaDeviceOpen()
-----------	-------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_IsaDeviceOpen()

```
DWORD DLLCALLCONV WDC_IsaDeviceOpen (
    _Output_ WDC_DEVICE_HANDLE * phDev,
    _In_ const WD_CARD * pDeviceInfo,
    _In_ const PVOID pDevCtx )
```

Allocates and initializes a WDC ISA device structure, registers the device with WinDriver, and returns a handle to the device.

- Verifies that none of the registered device resources (set in pDeviceInfo->Card.Item) are already locked for exclusive use.
- Maps the device's physical memory ranges device both to kernel-mode and user-mode address space, and stores the mapped addresses in the allocated 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, as well as retrieves and saves an interrupt handle, and this information is 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.

Parameters

out	phDev	Pointer to a handle to the WDC device allocated by the function
in	pDeviceInfo	Pointer to a card information structure, which contains information regarding the device to open
in	pDevCtx	Pointer to device context information, which will be stored in the device structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_KernelPluginOpen()

```
DWORD DLLCALLCONV WDC_KernelPluginOpen (
    _In_ WDC_DEVICE_HANDLE hDev,
```

```
    _In_ const CHAR * pcKPDriverName,
    _In_ PVOID pKPOpenData )
```

Opens a handle to a Kernel PlugIn driver.

Open a handle to Kernel PlugIn driver

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>pcKPDriverName</i>	Kernel PlugIn driver name
in	<i>pKPOpenData</i>	Kernel PlugIn driver open data to be passed to WD_KernelPlugInOpen() (see the WinDriver PCI Low-Level API Reference)

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_MultiTransfer()

```
DWORD DLLCALLCONV WDC_MultiTransfer (
    _In_ WD_TRANSFER * pTransCmds,
    _In_ DWORD dwNumTrans )
```

Performs a group of memory and/or I/O read/write transfers.

Parameters

in	<i>pTransCmds</i>	Pointer to an array of transfer commands information structures
in	<i>dwNumTrans</i>	Number of transfer commands in the <i>pTrans</i> array

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciDeviceClose()

```
DWORD DLLCALLCONV WDC_PciDeviceClose (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Uninitializes a WDC PCI device structure and frees the memory allocated for it.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
----	-------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    UINT32 u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
```

```
if (!WD_DriverName(DEFAULT_DRIVER_NAME))
{
    printf("Failed to set the driver name for WDC library.\n");
    return WD_INTERNAL_ERROR;
}
dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize debug options for WDC library.\n"
           "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
/* Open a handle to the driver and initialize the WDC library */
dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
           dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
BZERO(deviceInfo);
deviceInfo.pciSlot = slot;
dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed retrieving the device's resources "
           "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Open a device handle */
dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
           dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Read/Write memory examples: */
/* Check BYTE */
dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
                               &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
                               &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check WORD */
dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
                                &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
                               &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
                                &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
                               &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
                                &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
                               &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
Exit:
if (hDev)
{
    dwStatus = WDC_PciDeviceClose(hDev);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
               "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
}
WDC_DriverClose();
}
```

WDC_PciDeviceOpen()

```
DWORD DLLCALLCONV WDC_PciDeviceOpen (
    _Outptr_ WDC_DEVICE_HANDLE * phDev,
    _In_ const WD_PCI_CARD_INFO * pDeviceInfo,
    _In_ const PVOID pDevCtx )
```

Allocates and initializes a WDC PCI device structure, registers the device with WinDriver, and returns a handle to the device.

Open/Close device

- Verifies that none of the registered device resources (set in pDeviceInfo->Card.Item) are already locked for exclusive use.
- 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 allocated 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, as well as retrieves and saves an interrupt handle, and this information is 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.

Parameters

out	<i>phDev</i>	Pointer to a handle to the WDC device allocated by the function
in	<i>pDeviceInfo</i>	Pointer to a card information structure, which contains information regarding the device to open
in	<i>pDevCtx</i>	Pointer to device context information, which will be stored in the device structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    UINT32 u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        printf("Failed to set the driver name for WDC library.\n");
        return WD_INTERNAL_ERROR;
    }
    dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize debug options for WDC library.\n"
               "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    BZERO(deviceInfo);
    deviceInfo.pciSlot = slot;
    dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
    if (WD_STATUS_SUCCESS != dwStatus)
```

```

{
    printf("Failed retrieving the device's resources "
        "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Open a device handle */
dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Read/Write memory examples: */
/* Check BYTE */
dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check WORD */
dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
Exit:
if (hDev)
{
    dwStatus = WDC_PciDeviceClose(hDev);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
            "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
}
WDC_DriverClose();
}

```

WDC_PciGetDeviceInfo()

DWORD **DLLCALLCONV** WDC_PciGetDeviceInfo (
 Inout WD_PCI_CARD_INFO * pDeviceInfo)

Retrieves a PCI device's resources information (memory and I/O ranges and interrupt information).

Get device's resources information (PCI)

Parameters

in,out	<i>pDeviceInfo</i>	Pointer to a PCI device information structure
in	<i>pciSlot</i>	<i>pDeviceInfo.pciSlot</i>
out	<i>Card</i>	<i>pDeviceInfo.Card</i>

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    WORD u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        printf("Failed to set the driver name for WDC library.\n");
        return WD_INTERNAL_ERROR;
    }
    dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize debug options for WDC library.\n"
               "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    BZERO(deviceInfo);
    deviceInfo.pciSlot = slot;
    dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed retrieving the device's resources "
               "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    /* Open a device handle */
    dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    /* Read/Write memory examples: */
    /* Check BYTE */
    dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
                                   &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
                                 &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    /* Check WORD */
    dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
                                   &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
                                 &bData, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    /* Check UINT32 */
    dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
                                   &u32Data, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
                                 &u32Data, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    /* Check UINT64 */
    dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
                                   &u64Data, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
        goto Exit;
    dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
                                 &u64Data, WDC_ADDR_RW_DEFAULT);
    if (dwResult)
```

```

        goto Exit;
Exit:
    if (hDev)
    {
        dwStatus = WDC_PciDeviceClose(hDev);
        if (WD_STATUS_SUCCESS != dwStatus)
        {
            printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
                  "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
        }
    }
    WDC_DriverClose();
}

```

WDC_PciGetExpressGen()

DWORD **DLLCALLCONV** WDC_PciGetExpressGen (
 In WDC_DEVICE_HANDLE **hDev**)

Retrieves the PCI Express generation of a device.

Parameters

in	hDev	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
-----------	-------------	---

Returns

Returns 0 if device is not a PCI Express device, otherwise returns the PCI Express generation of the device;

WDC_PciGetExpressGenBySlot()

DWORD **DLLCALLCONV** WDC_PciGetExpressGenBySlot (
 In WD_PCI_SLOT * **pPciSlot**)

Retrieves the PCI Express generation of a device.

Parameters

in	pPciSlot	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
-----------	-----------------	---

Returns

Returns 0 if device is not a PCI Express device, otherwise returns the PCI Express generation of the device;

WDC_PciGetExpressOffset()

DWORD **DLLCALLCONV** WDC_PciGetExpressOffset (
 In WDC_DEVICE_HANDLE **hDev**,
 Outptr DWORD * **pdwOffset**)

Retrieves the PCI Express configuration registers' offset in the device's configuration space.

Scan PCI Capabilities

added to any extended configuration space register's fixed address. The fixed addresses were defined in the PCI Express Base Specification.

Parameters

in	hDev	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
out	pdwOffset	Pointer to the DWORD where the offset value will be written.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciGetType()

```
DWORD DLLCALLCONV WDC_PciGetType (
    _In_ WDC_DEVICE_HANDLE hDev,
    _Outptr_ WDC_PCI_HEADER_TYPE * pHdrType )
```

Retrieves the PCI device's configuration space header type.

The header type is hardware dependent and determines how the configuration space is arranged.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
out	<i>pHeaderType</i>	A pointer to the structure where the header type will be updated. WDC_PCI_HEADER_TYPE definition and available values can be seen in WinDriver/include/pci_regs.h

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfg()

```
DWORD DLLCALLCONV WDC_PciReadCfg (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _Outptr_ PVOID pData,
    _In_ DWORD dwBytes )
```

Identify device by handle.

Reads data from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pData</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space
in	<i>dwBytes</i>	The number of bytes to read

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfg16()

```
DWORD DLLCALLCONV WDC_PciReadCfg16 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _Outptr_ WORD * pwVal )
```

Reads 2 bytes (16 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

[WDC_PciReadCfg32\(\)](#)

```
DWORD DLLCALLCONV WDC_PciReadCfg32 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _Outptr_ UINT32 * pdwVal )
```

Reads 4 bytes (32 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

[WDC_PciReadCfg64\(\)](#)

```
DWORD DLLCALLCONV WDC_PciReadCfg64 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _Outptr_ UINT64 * pqwVal )
```

Reads 8 bytes (64 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pqwVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

[WDC_PciReadCfg8\(\)](#)

```
DWORD DLLCALLCONV WDC_PciReadCfg8 (
    _In_ WDC_DEVICE_HANDLE hDev,
```

```
_In_ DWORD dwOffset,
_Outptr_ BYTE * pbVal )
```

Read/write 8/16/32/64 bits from the PCI configuration space.

Identify device by handle Reads 1 byte (8 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pbVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfgBySlot()

```
DWORD DLLCALLTYPE WDC_PciReadCfgBySlot (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _Outptr_ PVOID pData,
    _In_ DWORD dwBytes )
```

Read/write a block of any length from the PCI configuration space.

Access PCI configuration space

Identify device by its location. Reads data from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space. The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pData</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space.
in	<i>dwBytes</i>	The number of bytes to read.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfgBySlot16()

```
DWORD DLLCALLTYPE WDC_PciReadCfgBySlot16 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _Outptr_ WORD * pwVal )
```

Reads 2 bytes (16 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pdwVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfgBySlot32()

```
DWORD DLLCALLCONV WDC_PciReadCfgBySlot32 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _Outptr_ UINT32 * pdwVal )
```

Reads 4 bytes (32 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pdwVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfgBySlot64()

```
DWORD DLLCALLCONV WDC_PciReadCfgBySlot64 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _Outptr_ UINT64 * pqwVal )
```

Reads 8 bytes (64 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pqwVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciReadCfgBySlot8()

```
DWORD DLLCALLCONV WDC_PciReadCfgBySlot8 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _Outptr_ BYTE * pbVal )
```

Read/write 8/16/32/64 bits from the PCI configuration space.

Identify device by its location. Reads 1 byte (8 bits) from a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space. The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to read from.
out	<i>pbVal</i>	Pointer to a buffer to be filled with the data that is read from the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciScanCaps()

```
DWORD DLLCALLCONV WDC_PciScanCaps (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwCapId,
    _Outptr_ WDC_PCI_SCAN_CAPS_RESULT * pScanCapsResult )
```

Scans the basic PCI capabilities of the given device for the specified capability (or for all capabilities).

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen() .
in	<i>dwCapId</i>	ID of the basic PCI capability for which to search, or WD_PCI_CAP_ID_ALL to search for all basic PCI capabilities.
out	<i>pScanCapsResult</i>	A pointer to a structure that will be updated by the function with the results of the basic PCI capabilities scan.

Returns

Returns WD_STATUS_SUCCESS (0) on success,

WDC_PciScanCapsBySlot()

```
DWORD DLLCALLCONV WDC_PciScanCapsBySlot (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwCapId,
    _Outptr_ WDC_PCI_SCAN_CAPS_RESULT * pScanCapsResult )
```

Scans the basic PCI capabilities of the given device for the specified capability (or for all capabilities).

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices() .
in	<i>dwCapId</i>	ID of the basic PCI capability for which to search, or WD_PCI_CAP_ID_ALL to search for all basic PCI capabilities.

Parameters

out	<i>pScanCapsResult</i>	A pointer to a structure that will be updated by the function with the results of the basic PCI capabilities scan.
------------	------------------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success,

WDC_PciScanDevices()

```
DWORD DLLCALLCONV WDC_PciScanDevices (
    _In_ DWORD dwVendorId,
    _In_ DWORD dwDeviceId,
    _Outptr_ WDC_PCI_SCAN_RESULT * pPciScanResult )
```

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 locations.

Scan bus (PCI)

The function performs the scan by iterating through all possible PCI buses on the host platform, then through all possible PCI slots, and then through all possible PCI functions.

Parameters

in	<i>dwVendorId</i>	Vendor ID to search for, or 0 to search for all vendor IDs
in	<i>dwDeviceId</i>	Device ID to search for, or 0 to search for all device IDs
out	<i>pPciScanResult</i>	A pointer to a structure that will be updated by the function with the results of the PCI bus scan

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciScanDevicesByTopology()

```
DWORD DLLCALLCONV WDC_PciScanDevicesByTopology (
    _In_ DWORD dwVendorId,
    _In_ DWORD dwDeviceId,
    _Outptr_ WDC_PCI_SCAN_RESULT * pPciScanResult )
```

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 locations.

The function performs the scan by topology i.e for each located bridge the function scans the connected devices and functions reported by the bridge, and only then proceeds to scan the next bridge.

Parameters

in	<i>dwVendorId</i>	Vendor ID to search for, or 0 to search for all vendor IDs
in	<i>dwDeviceId</i>	Device ID to search for, or 0 to search for all device IDs
out	<i>pPciScanResult</i>	A pointer to a structure that will be updated by the function with the results of the PCI bus scan

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciScanExtCaps()

```
DWORD DLLCALLCONV WDC_PciScanExtCaps (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwCapId,
    _Outptr_ WDC_PCI_SCAN_CAPS_RESULT * pScanCapsResult )
```

Scans the extended (PCI Express) PCI capabilities of the given device for the specified capability (or for all capabilities).

Parameters

in	<i>hDev</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices() .
in	<i>dwCapId</i>	ID of the extended PCI capability for which to search, or WD_PCI_CAP_ID_ALL to search for all extended PCI capabilities.
out	<i>pScanCapsResult</i>	A pointer to a structure that will be updated by the function with the results of the extended (PCI Express) PCI capabilities scan.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciScanRegisteredDevices()

```
DWORD DLLCALLCONV WDC_PciScanRegisteredDevices (
    _In_ DWORD dwVendorId,
    _In_ DWORD dwDeviceId,
    _Outptr_ WDC_PCI_SCAN_RESULT * pPciScanResult )
```

Scans the PCI bus for all devices with the specified vendor and device ID combination that have been registered to work with WinDriver, and returns information regarding the matching devices that were found and their locations. The function performs the scan by iterating through all possible PCI buses on the host platform, then through all possible PCI slots, and then through all possible PCI functions.

Parameters

in	<i>dwVendorId</i>	Vendor ID to search for, or 0 to search for all vendor IDs
in	<i>dwDeviceId</i>	Device ID to search for, or 0 to search for all device IDs
out	<i>pPciScanResult</i>	A pointer to a structure that will be updated by the function with the results of the PCI bus scan

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciSriovDisable()

```
DWORD DLLCALLCONV WDC_PciSriovDisable (
    _In_ WDC_DEVICE_HANDLE hDev )
```

Disables SR-IOV for a supported device and removes all the assigned VFs.

Parameters

in	<i>hDev</i>	Handle to a PlugandPlay WDC device, returned by WDC_PciDeviceOpen()
----	-------------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
void SriosExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus, dwNumVFs = 3;
    dwStatus = WDC_PciSriovEnable(hDev, dwNumVFs);
    if (dwStatus)
    {
        printf("Error! Could not enable SRIOV\n");
        return;
    }
    dwNumVFs = 0;
    dwStatus = WDC_PciSriovGetNumVFs(hDev, &dwNumVFs);
    if (dwStatus)
    {
        printf("Error! Could not get number of virtual functions\n");
        goto Error;
    }
    // Should be 3
    printf("Number of virtual functions: %ld\n", dwNumVFs);
    /*
     * More code...
     */
Error:
    dwStatus = WDC_PciSriovDisable(hDev);
    if (dwStatus)
    {
        printf("Error! Could not disable SRIOV\n");
    }
}
```

[WDC_PciSriovEnable\(\)](#)

```
DWORD DLLCALLCONV WDC_PciSriovEnable (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwNumVFs )
```

SR-IOV API functions are not part of the standard WinDriver API, and not included in the standard version of WinDriver.

Control device's SR-IOV capability (PCIe)

"WinDriver for Server" API and require "WinDriver for Server" license. Note that "WinDriver for Server" APIs are included in WinDriver evaluation version. Enables SR-IOV for a supported device.

Parameters

in	<i>hDev</i>	Handle to a Plug-and-Play WDC device, returned by WDC_PciDeviceOpen()
in	<i>dwNumVFs</i>	The number of virtual functions (VFs) to be assigned to hDev

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
void SriosExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus, dwNumVFs = 3;
    dwStatus = WDC_PciSriovEnable(hDev, dwNumVFs);
    if (dwStatus)
    {
        printf("Error! Could not enable SRIOV\n");
        return;
    }
    dwNumVFs = 0;
    dwStatus = WDC_PciSriovGetNumVFs(hDev, &dwNumVFs);
    if (dwStatus)
```

```

    {
        printf("Error! Could not get number of virtual functions\n");
        goto Error;
    }
    // Should be 3
    printf("Number of virtual functions: %ld\n", dwNumVFs);
    /*
        More code...
    */
Error:
    dwStatus = WDC_PciSriovDisable(hDev);
    if (dwStatus)
    {
        printf("Error! Could not disable SRIOV\n");
    }
}

```

WDC_PciSriovGetNumVFs()

DWORD **DLLCALLCONV** WDC_PciSriovGetNumVFs (
 In WDC_DEVICE_HANDLE hDev,
 Outptr DWORD * pdwNumVFs)

Gets the number of virtual functions assigned to a supported device.

Parameters

in	<i>hDev</i>	Handle to a PlugandPlay WDC device, returned by WDC_PciDeviceOpen()
out	<i>pdwNumVFs</i>	A pointer to a DWORD to store the number of VFs assigned

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```

void SriovExample(WDC_DEVICE_HANDLE hDev)
{
    DWORD dwStatus, dwNumVFs = 3;
    dwStatus = WDC_PciSriovEnable(hDev, dwNumVFs);
    if (dwStatus)
    {
        printf("Error! Could not enable SRIOV\n");
        return;
    }
    dwNumVFs = 0;
    dwStatus = WDC_PciSriovGetNumVFs(hDev, &dwNumVFs);
    if (dwStatus)
    {
        printf("Error! Could not get number of virtual functions\n");
        goto Error;
    }
    // Should be 3
    printf("Number of virtual functions: %ld\n", dwNumVFs);
    /*
        More code...
    */
Error:
    dwStatus = WDC_PciSriovDisable(hDev);
    if (dwStatus)
    {
        printf("Error! Could not disable SRIOV\n");
    }
}

```

WDC_PciWriteCfg()

DWORD **DLLCALLCONV** WDC_PciWriteCfg (
 In WDC_DEVICE_HANDLE hDev,
 In DWORD dwOffset,
 In PVOID pData,
 In DWORD dwBytes)

Writes data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>pData</i>	Pointer to a data buffer that holds the data to write
in	<i>dwBytes</i>	The number of bytes to write

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfg16()

```
DWORD DLLCALLCONV WDC_PciWriteCfg16 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _In_ WORD wVal )
```

Writes 2 bytes (16 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>wVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfg32()

```
DWORD DLLCALLCONV WDC_PciWriteCfg32 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _In_ UINT32 dwVal )
```

Writes 4 bytes (32 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>dwVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfg64()

```
DWORD DLLCALLCONV WDC_PciWriteCfg64 (
```

```
_In_ WDC_DEVICE_HANDLE hDev,
_In_ DWORD dwOffset,
_In_ UINT64 qwVal )
```

Writes 8 bytes (64 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>qwVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfg8()

```
DWORD DLLCALLTYPE WDC_PciWriteCfg8 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwOffset,
    _In_ BYTE bVal )
```

Writes 1 byte (8 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

Parameters

in	<i>hDev</i>	Handle to a WDC PCI device structure, returned by WDC_PciDeviceOpen()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>bVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfgBySlot()

```
DWORD DLLCALLTYPE WDC_PciWriteCfgBySlot (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _In_ PVOID pData,
    _In_ DWORD dwBytes )
```

Write data to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>pData</i>	Pointer to a data buffer that holds the data to write
in	<i>dwBytes</i>	The number of bytes to write

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfgBySlot16()

```
DWORD DLLCALLCONV WDC_PciWriteCfgBySlot16 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _In_ WORD wVal )
```

writes 2 bytes (16 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	pPciSlot	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	dwOffset	The offset from the beginning of the PCI configuration space to write to.
in	wVal	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfgBySlot32()

```
DWORD DLLCALLCONV WDC_PciWriteCfgBySlot32 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _In_ UINT32 dwVal )
```

writes 4 bytes (32 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	pPciSlot	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	dwOffset	The offset from the beginning of the PCI configuration space to write to.
in	dwVal	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfgBySlot64()

```
DWORD DLLCALLCONV WDC_PciWriteCfgBySlot64 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _In_ UINT64 qwVal )
```

writes 8 bytes (64 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>qwVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_PciWriteCfgBySlot8()

```
DWORD DLLCALLCONV WDC_PciWriteCfgBySlot8 (
    _In_ WD_PCI_SLOT * pPciSlot,
    _In_ DWORD dwOffset,
    _In_ BYTE bVal )
```

writes 1 byte (8 bits) to a specified offset in a PCI device's configuration space or a PCI Express device's extended configuration space.

The device is identified by its location on the PCI bus.

Parameters

in	<i>pPciSlot</i>	Pointer to a PCI device location information structure, which can be acquired by calling WDC_PciScanDevices()
in	<i>dwOffset</i>	The offset from the beginning of the PCI configuration space to write to.
in	<i>bVal</i>	The data to write to the PCI configuration space

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_ReadAddr16()

```
DWORD DLLCALLCONV WDC_ReadAddr16 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _Outptr_ WORD * pwVal )
```

reads 2 byte (16 bits) from a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to read from
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to read from
out	<i>pwVal</i>	Pointer to a buffer to be filled with the data that is read from the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_ReadAddr32()

```
DWORD DLLCALLCONV WDC_ReadAddr32 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _Outptr_ UINT32 * pdwVal )
```

reads 4 byte (32 bits) from a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to read from
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to read from
out	<i>pdwVal</i>	Pointer to a buffer to be filled with the data that is read from the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_ReadAddr64()

```
DWORD DLLCALLCONV WDC_ReadAddr64 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _Outptr_ UINT64 * pqwVal )
```

reads 8 byte (64 bits) from a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to read from
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to read from
out	<i>pqwVal</i>	Pointer to a buffer to be filled with the data that is read from the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_ReadAddr8()

```
DWORD DLLCALLCONV WDC_ReadAddr8 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _Outptr_ BYTE * pbVal )
```

Read/write a device's address space (8/16/32/64 bits)
reads 1 byte (8 bits) from a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to read from
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to read from
out	<i>pbVal</i>	Pointer to a buffer to be filled with the data that is read from the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_ReadAddrBlock()

```
DWORD DLLCALLCONV WDC_ReadAddrBlock (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ DWORD dwBytes,
    _Outptr_ PVOID pData,
    _In_ WDC_ADDR_MODE mode,
    _In_ WDC_ADDR_RW_OPTIONS options )
```

Reads a block of data from the device.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to read from
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to read from
in	<i>dwBytes</i>	The number of bytes to read
out	<i>pData</i>	Pointer to a buffer to be filled with the data that is read from the device
in	<i>mode</i>	The read access mode -see WDC_ADDR_MODE
in	<i>options</i>	A bit mask that determines how the data will be read see WDC_ADDR_RW_OPTIONS. The function automatically sets the WDC_RW_BLOCK flag.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_SetDebugOptions()

```
DWORD DLLCALLCONV WDC_SetDebugOptions (
    _In_ WDC_DBG_OPTIONS dbgOptions,
    _In_ const CHAR * pcDbgFile )
```

Sets debug options for the WDC library - see the description of WDC_DBG_OPTIONS for details regarding the possible debug options to set.

Debugging and error handling

This function is typically called at the beginning of the application, after the call to [WDC_DriverOpen\(\)](#), and can be re-called at any time while the WDC library is in use (i.e, [WDC_DriverClose\(\)](#) 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](#).

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.

Parameters

in	<i>dbgOptions</i>	A bit mask of flags indicating the desired debug settings - see WDC_DBG_OPTIONS . If this parameter is set to zero, the default debug options will be used <ul style="list-style-type: none"> • see WDC_DBG_DEFAULT
in	<i>pcDbgFile</i>	WDC debug output file. This parameter is relevant only if the WDC_DBG_OUT_FILE flag is set in the debug options (<i>dbgOptions</i>) (either directly or via one of the convenience debug options combinations <ul style="list-style-type: none"> • see WDC_DBG_OPTIONS). If the WDC_DBG_OUT_FILE debug flag is set and <i>sDbgFile</i> is NULL, WDC debug messages will be logged to the default debug file <i>stderr</i>.

Returns

Returns [WD_STATUS_SUCCESS](#) (0) on success, or an appropriate error code otherwise

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    UINT32 u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        printf("Failed to set the driver name for WDC library.\n");
        return WD_INTERNAL_ERROR;
    }
    dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize debug options for WDC library.\n"
               "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    BZERO(deviceInfo);
    deviceInfo.pciSlot = slot;
    dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed retrieving the device's resources "
               "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    /* Open a device handle */
    dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
    if (WD_STATUS_SUCCESS != dwStatus)
```

```

{
    printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Read/Write memory examples: */
/* Check BYTE */
dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check WORD */
dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
    goto Exit;
Exit:
if (hDev)
{
    dwStatus = WDC_PciDeviceClose(hDev);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
            "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
}
WDC_DriverClose();
}

```

WDC_Sleep()

```

DWORD DLLCALLCONV WDC_Sleep (
    _In_ DWORD dwMicroSecs,
    _In_ WDC_SLEEP_OPTIONS options )

```

Delays execution for the specified duration of time (in microseconds).
By default the function performs a busy sleep (consumes CPU cycles).

Parameters

in	<i>dwMicroSecs</i>	The number of microseconds to sleep.
in	<i>options</i>	Sleep options.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_Trace()

```
void DLLCALLCONV WDC_Trace (
    const CHAR * format,
    ...
)
```

Displays debug trace messages according to the WDC debug options.

- see WDC_DBG_OPTIONS and [WDC_SetDebugOptions\(\)](#)

Parameters

in	<i>format</i>	Format-control string, which contains the error message to display. The string is limited to 256 characters (CHAR)
in	...	Optional arguments for the format string

Returns

None

WDC_Version()

```
DWORD DLLCALLCONV WDC_Version (
    _Outptr_ CHAR * pcVersion,
    _In_     DWORD dwLen,
    _Outptr_ DWORD * pdwVersion )
```

Returns the version number of the WinDriver kernel module used by the WDC library.

Parameters

out	<i>pcVersion</i>	Pointer to a pre-allocated buffer to be filled by the function with the driver's version information string. The size of the version string buffer must be at least 128 bytes (characters).
in	<i>dwLen</i>	Length of <i>sVersion</i> . If length will be too short, this function will fail.
out	<i>pdwVersion</i>	Pointer to a value indicating the version number of the WinDriver kernel module used by the WDC library

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_WriteAddr16()

```
DWORD DLLCALLCONV WDC_WriteAddr16 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ WORD wVal )
```

writes 2 byte (16 bits) to a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to write to
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to write to
in	<i>wVal</i>	The data to write to the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_WriteAddr32()

```
DWORD DLLCALLCONV WDC_WriteAddr32 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ UINT32 dwVal )
```

writes 4 byte (32 bits) to a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to write to
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to write to
in	<i>dwVal</i>	The data to write to the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_WriteAddr64()

```
DWORD DLLCALLCONV WDC_WriteAddr64 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ UINT64 qwVal )
```

writes 8 byte (64 bits) to a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to write to
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to write to
in	<i>qwVal</i>	The data to write to the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_WriteAddr8()

```
DWORD DLLCALLCONV WDC_WriteAddr8 (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ BYTE bVal )
```

writes 1 byte (8 bits) to a specified memory or I/O address.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to write to
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to write to
in	<i>bVal</i>	The data to write to the specified address

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDC_WriteAddrBlock()

```
DWORD DLLCALLCONV WDC_WriteAddrBlock (
    _In_ WDC_DEVICE_HANDLE hDev,
    _In_ DWORD dwAddrSpace,
    _In_ KPTR dwOffset,
    _In_ DWORD dwBytes,
    _In_ PVOID pData,
    _In_ WDC_ADDR_MODE mode,
    _In_ WDC_ADDR_RW_OPTIONS options )
```

Writes a block of data to the device.

Parameters

in	<i>hDev</i>	Handle to a WDC device, returned by WDC_xxxDeviceOpen()
in	<i>dwAddrSpace</i>	The memory or I/O address space to write to
in	<i>dwOffset</i>	The offset from the beginning of the specified address space (dwAddrSpace) to write to
in	<i>dwBytes</i>	The number of bytes to write
in	<i>pData</i>	Pointer to a buffer that holds the data to write to the device
in	<i>mode</i>	The write access mode -see WDC_ADDR_MODE
in	<i>options</i>	A bit mask that determines how the data will be written see WDC_ADDR_RW_OPTIONS. The function automatically sets the WDC_RW_BLOCK flag.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

wdc_lib.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WDC_LIB_H_
00004 #define _WDC_LIB_H_
```

```
00005
00006 /*****
00007 * File: wdc_lib.h - Shared WD card (WDC) library header.
00008 * This file defines the WDC library's high-level
00009 * interface
00010 *****/
00011
00012 #if defined(__KERNEL__)
00013     #include "kpstdlib.h"
00014 #endif
00015 #include "windrvr.h"
00016 #include "windrvr_int_thread.h"
00017 #include "windrvr_events.h"
00018 #include "bits.h"
00019 #include "pci_regs.h"
00020
00021 #ifdef __cplusplus
00022     extern "C" {
00023 #endif
00024 *****
00025 General definitions
00026 *****
00027
00028 #define MAX_NAME 128
00029 #define MAX_DESC 128
00030 #define MAX_NAME_DISPLAY 22
00031
00032 typedef void *WDC_DEVICE_HANDLE;
00033
00034
00035 #ifndef __KERNEL__
00036
00037 typedef struct {
00038     DWORD      dwNumDevices;
00039     WD_PCI_ID  deviceId[WD_PCI_CARDS];
00040     WD_PCI_SLOT deviceSlot[WD_PCI_CARDS];
00041 } WDC_PCI_SCAN_RESULT;
00042 #endif
00043
00044
00045 typedef struct {
00046     DWORD      dwNumCaps;
00047     WD_PCI_CAP pciCaps[WD_PCI_MAX_CAPS];
00048 } WDC_PCI_SCAN_CAPS_RESULT;
00049
00050
00051 /* Driver open options */
00052 /* Basic driver open flags */
00053 #define WDC_DRV_OPEN_CHECK_VER 0x1
00054 #define WDC_DRV_OPEN_REG_LIC 0x2
00055 /* Convenient driver open options */
00056 #define WDC_DRV_OPEN_BASIC 0x0
00057 #define WDC_DRV_OPEN_KP WDC_DRV_OPEN_BASIC
00058 #define WDC_DRV_OPEN_ALL (WDC_DRV_OPEN_CHECK_VER | WDC_DRV_OPEN_REG_LIC)
00059 #if defined(__KERNEL__)
00060     #define WDC_DRV_OPEN_DEFAULT WDC_DRV_OPEN_KP
00061 #else
00062     #define WDC_DRV_OPEN_DEFAULT WDC_DRV_OPEN_ALL
00063 #endif
00064
00065 typedef DWORD WDC_DRV_OPEN_OPTIONS;
00066
00067
00068 /* Debug information display options */
00069 #define WDC_DBG_OUT_DBM 0x1
00070 #define WDC_DBG_OUT_FILE 0x2
00071 #define WDC_DBG_LEVEL_ERR 0x10
00072 #define WDC_DBG_LEVEL_TRACE 0x20
00073 #define WDC_DBG_NONE 0x100
00074 #define WDC_DBG_DEFAULT (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
00075
00076 #define WDC_DBG_DBM_ERR (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_ERR)
00077 #define WDC_DBG_DBM_TRACE (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
00078
00079 #if !defined(__KERNEL__)
00080     #define WDC_DBG_FILE_ERR (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
00081     #define WDC_DBG_FILE_TRACE (WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
00082
00083 #define WDC_DBG_DBM_FILE_ERR \
00084     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
00085 #define WDC_DBG_DBM_FILE_TRACE \
00086     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
00087
00088 #define WDC_DBG_FULL \
00089     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
00090 #else
00091     #define WDC_DBG_FULL (WDC_DBG_OUT_DBM | WDC_DBG_LEVEL_TRACE)
00092 #endif
00093
00094
00095 #define WDC_DBG_DBM_FILE_ERR \
00096     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_ERR)
00097 #define WDC_DBG_DBM_FILE_TRACE \
00098     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
00099
00100 #define WDC_DBG_FULL \
00101     (WDC_DBG_OUT_DBM | WDC_DBG_OUT_FILE | WDC_DBG_LEVEL_TRACE)
00102
00103 #define WDC_SLEEP_BUSY 0
00104 #define WDC_SLEEP_NON_BUSY SLEEP_NON_BUSY
```

```
00111 typedef DWORD WDC_SLEEP_OPTIONS;
00112
00113 /* -----
00114     Memory / I/O / Registers
00115 ----- */
00116 typedef enum {
00117     WDC_WRITE,
00118     WDC_READ,
00119     WDC_READ_WRITE
00120 } WDC_DIRECTION;
00121
00122 typedef enum {
00123     WDC_ADDR_RW_DEFAULT = 0x0,
00124     WDC_ADDR_RW_NO_AUTOINC = 0x4
00125 } WDC_ADDR_RW_OPTIONS;
00126
00127 /* Memory/I/O address size and access mode definitions (size - in bytes) */
00128 #define WDC_SIZE_8 ((DWORD)sizeof(BYTE))
00129 #define WDC_SIZE_16 ((DWORD)sizeof(WORD))
00130 #define WDC_SIZE_32 ((DWORD)sizeof(UINT32))
00131 #define WDC_SIZE_64 ((DWORD)sizeof(UINT64))
00132
00133 typedef DWORD WDC_ADDR_SIZE;
00134
00135 typedef enum {
00136     WDC_MODE_8 = WDC_SIZE_8,
00137     WDC_MODE_16 = WDC_SIZE_16,
00138     WDC_MODE_32 = WDC_SIZE_32,
00139     WDC_MODE_64 = WDC_SIZE_64
00140 } WDC_ADDR_MODE;
00141
00142
00143 #define WDC_ADDR_MODE_TO_SIZE(mode) (DWORD)mode
00144 #define WDC_ADDR_SIZE_TO_MODE(size) (((size) > WDC_SIZE_32) ? WDC_MODE_64 : \
00145     ((size) > WDC_SIZE_16) ? WDC_MODE_32 : \
00146     ((size) > WDC_SIZE_8) ? WDC_MODE_16 : WDC_MODE_8)
00147
00148
00149 #define WDC_AD_CFG_SPACE 0xFF
00150
00151
00152 /*****
00153     Function Prototypes
00154 *****/
00155 /* -----
00156     General
00157 ----- */
00158
00159 HANDLE DLLCALLCONV WDC_GetWDHandle(void);
00160
00161
00162 PVOID DLLCALLCONV WDC_GetDevContext(_In_ WDC_DEVICE_HANDLE hDev);
00163
00164 WD_BUS_TYPE DLLCALLCONV WDC_GetBusType(_In_ WDC_DEVICE_HANDLE hDev);
00165
00166
00167 DWORD DLLCALLCONV WDC_Sleep(_In_ DWORD dwMicroSecs,
00168     _In_ WDC_SLEEP_OPTIONS options);
00169
00170
00171 DWORD DLLCALLCONV WDC_Version(_Outptr_ CHAR *pcVersion, _In_ DWORD dwLen,
00172     _Outptr_ DWORD *pdwVersion);
00173
00174
00175 DWORD DLLCALLCONV WDC_DriverOpen(_In_ WDC_DRV_OPEN_OPTIONS openOptions,
00176     _In_ const CHAR *pcLicense);
00177
00178
00179 DWORD DLLCALLCONV WDC_DriverClose(void);
00180
00181
00182 #ifndef __KERNEL__
00183 DWORD DLLCALLCONV WDC_PciScanDevices(_In_ DWORD dwVendorId,
00184     _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult);
00185
00186
00187 DWORD DLLCALLCONV WDC_PciScanDevicesByTopology(_In_ DWORD dwVendorId,
00188     _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult);
00189
00190
00191 DWORD DLLCALLCONV WDC_PciScanRegisteredDevices(_In_ DWORD dwVendorId,
00192     _In_ DWORD dwDeviceId, _Outptr_ WDC_PCI_SCAN_RESULT *pPciScanResult);
00193
00194
00195 #endif
00196
00197
00198 DWORD DLLCALLCONV WDC_PciGetExpressOffset(_In_ WDC_DEVICE_HANDLE hDev,
00199     _Outptr_ DWORD *pdwOffset);
00200
00201
00202 DWORD DLLCALLCONV WDC_PciGetHeaderType(_In_ WDC_DEVICE_HANDLE hDev,
00203     _Outptr_ WDC_PCI_HEADER_TYPE *pHeaderType);
00204
00205
00206 DWORD DLLCALLCONV WDC_PciScanCaps(_In_ WDC_DEVICE_HANDLE hDev,
00207     _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult);
00208
00209
00210 DWORD DLLCALLCONV WDC_PciScanCapsBySlot(_In_ WD_PCI_SLOT *pPciSlot,
00211     _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult);
00212
00213
00214 DWORD DLLCALLCONV WDC_PciScanExtCaps(_In_ WDC_DEVICE_HANDLE hDev,
00215     _In_ DWORD dwCapId, _Outptr_ WDC_PCI_SCAN_CAPS_RESULT *pScanCapsResult);
00216
00217
00218 DWORD DLLCALLCONV WDC_PciGetExpressGenBySlot(_In_ WD_PCI_SLOT *pPciSlot);
00219
```

```
00452 DWORD DLLCALLCONV WDC_PciGetExpressGen(_In_ WDC_DEVICE_HANDLE hDev);
00453
00457 #ifndef __KERNEL__
00471 DWORD DLLCALLCONV WDC_PciGetDeviceInfo(_Inout_ WD_PCI_CARD_INFO *pDeviceInfo);
00472 #endif
00473
00496 DWORD DLLCALLCONV WDC_PciSriovEnable(_In_ WDC_DEVICE_HANDLE hDev,
00497     _In_ DWORD dwNumVFs);
00498
00510 DWORD DLLCALLCONV WDC_PciSriovDisable(_In_ WDC_DEVICE_HANDLE hDev);
00511
00525 DWORD DLLCALLCONV WDC_PciSriovGetNumVFs(_In_ WDC_DEVICE_HANDLE hDev,
00526     _Outptr_ DWORD *pdwNumVFs);
00527
00531 #if !defined(__KERNEL__)
00561 DWORD DLLCALLCONV WDC_PciDeviceOpen(_Outptr_ WDC_DEVICE_HANDLE *phDev,
00562     _In_ const WD_PCI_CARD_INFO *pDeviceInfo, _In_ const PVOID pDevCtx);
00563
00592 DWORD DLLCALLCONV WDC_IsaDeviceOpen(_Outptr_ WDC_DEVICE_HANDLE *phDev,
00593     _In_ const WD_CARD *pDeviceInfo, _In_ const PVOID pDevCtx);
00594
00607 DWORD DLLCALLCONV WDC_PciDeviceClose(_In_ WDC_DEVICE_HANDLE hDev);
00608
00619 DWORD DLLCALLCONV WDC_IsaDeviceClose(_In_ WDC_DEVICE_HANDLE hDev);
00620 #endif
00621
00655 DWORD WDC_CardCleanupSetup(_In_ WDC_DEVICE_HANDLE hDev,
00656     _In_ WD_TRANSFER *pTransCmds, _In_ DWORD dwCmds, _In_ BOOL fForceCleanup);
00657
00675 DWORD DLLCALLCONV WDC_KernelPlugInOpen(_In_ WDC_DEVICE_HANDLE hDev,
00676     _In_ const CHAR *pcKPDriverName, _In_ PVOID pKPOpenData);
00677
00702 DWORD DLLCALLCONV WDC_CallKerPlug(_In_ WDC_DEVICE_HANDLE hDev,
00703     _In_ DWORD dwMsg, _Inout_ PVOID pData, _Outptr_ PDWORD pdwResult);
00704
00713 #define WDC_ReadMem8(addr, off) *(volatile BYTE *)((UPTR)(addr) + (UPTR)(off))
00714
00718 #define WDC_ReadMem16(addr, off) \
00719     *(volatile WORD *)((UPTR)(addr) + (UPTR)(off))
00720
00724 #define WDC_ReadMem32(addr, off) \
00725     *(volatile UINT32 *)((UPTR)(addr) + (UPTR)(off))
00726
00730 #define WDC_ReadMem64(addr, off) \
00731     *(volatile UINT64 *)((UPTR)(addr) + (UPTR)(off))
00732
00733
00737 #define WDC_WriteMem8(addr, off, val) \
00738     *(volatile BYTE *)(((UPTR)(addr) + (UPTR)(off))) = (val)
00739
00743 #define WDC_WriteMem16(addr, off, val) \
00744     *(volatile WORD *)(((UPTR)(addr) + (UPTR)(off))) = (val)
00745
00749 #define WDC_WriteMem32(addr, off, val) \
00750     *(volatile UINT32 *)(((UPTR)(addr) + (UPTR)(off))) = (val)
00751
00755 #define WDC_WriteMem64(addr, off, val) \
00756     *(volatile UINT64 *)(((UPTR)(addr) + (UPTR)(off))) = (val)
00757
00758
00775 DWORD DLLCALLCONV WDC_ReadAddr8(_In_ WDC_DEVICE_HANDLE hDev,
00776     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ BYTE *pbVal);
00777
00792 DWORD DLLCALLCONV WDC_ReadAddr16(_In_ WDC_DEVICE_HANDLE hDev,
00793     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ WORD *pwVal);
00794
00809 DWORD DLLCALLCONV WDC_ReadAddr32(_In_ WDC_DEVICE_HANDLE hDev,
00810     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ UINT32 *pdwVal);
00811
00826 DWORD DLLCALLCONV WDC_ReadAddr64(_In_ WDC_DEVICE_HANDLE hDev,
00827     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _Outptr_ UINT64 *pqwVal);
00828
00842 DWORD DLLCALLCONV WDC_WriteAddr8(_In_ WDC_DEVICE_HANDLE hDev,
00843     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ BYTE bVal);
00844
00858 DWORD DLLCALLCONV WDC_WriteAddr16(_In_ WDC_DEVICE_HANDLE hDev,
00859     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ WORD wVal);
00860
00874 DWORD DLLCALLCONV WDC_WriteAddr32(_In_ WDC_DEVICE_HANDLE hDev,
00875     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ UINT32 dwVal);
00876
00890 DWORD DLLCALLCONV WDC_WriteAddr64(_In_ WDC_DEVICE_HANDLE hDev,
00891     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ UINT64 qwVal);
00892
00913 DWORD DLLCALLCONV WDC_ReadAddrBlock(_In_ WDC_DEVICE_HANDLE hDev,
00914     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ DWORD dwBytes,
00915     _Outptr_ PVOID pData, _In_ WDC_ADDR_MODE mode,
```

```
00916     _In_ WDC_ADDR_RW_OPTIONS options);
00917
00937 DWORD DLLCALLCONV WDC_WriteAddrBlock(_In_ WDC_DEVICE_HANDLE hDev,
00938     _In_ DWORD dwAddrSpace, _In_ KPTR dwOffset, _In_ DWORD dwBytes,
00939     _In_ PVOID pData, _In_ WDC_ADDR_MODE mode,
00940     _In_ WDC_ADDR_RW_OPTIONS options);
00941
00943 /* @snippet highlevel_examples.c WDC_ReadAddrBlock8 */
00944 #define WDC_ReadAddrBlock8(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00945     WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00946         WDC_MODE_8, options)
00947
00949 /* @snippet highlevel_examples.c WDC_ReadAddrBlock16 */
00950 #define WDC_ReadAddrBlock16(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00951     WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00952         WDC_MODE_16, options)
00953
00955 /* @snippet highlevel_examples.c WDC_ReadAddrBlock32 */
00956 #define WDC_ReadAddrBlock32(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00957     WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00958         WDC_MODE_32, options)
00959
00961 /* @snippet highlevel_examples.c WDC_ReadAddrBlock64 */
00962 #define WDC_ReadAddrBlock64(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00963     WDC_ReadAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00964         WDC_MODE_64, options)
00965
00967 /* @snippet highlevel_examples.c WDC_WriteAddrBlock8 */
00968 #define WDC_WriteAddrBlock8(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00969     WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00970         WDC_MODE_8, options)
00971
00973 /* @snippet highlevel_examples.c WDC_WriteAddrBlock16 */
00974 #define WDC_WriteAddrBlock16(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00975     WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00976         WDC_MODE_16, options)
00977
00979 /* @snippet highlevel_examples.c WDC_WriteAddrBlock32 */
00980 #define WDC_WriteAddrBlock32(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00981     WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00982         WDC_MODE_32, options)
00983
00985 /* @snippet highlevel_examples.c WDC_WriteAddrBlock64 */
00986 #define WDC_WriteAddrBlock64(hDev,dwAddrSpace,dwOffset,dwBytes,pData,options) \
00987     WDC_WriteAddrBlock(hDev, dwAddrSpace, dwOffset, dwBytes, pData, \
00988         WDC_MODE_64, options)
00989
01000 DWORD DLLCALLCONV WDC_MultiTransfer(_In_ WD_TRANSFER *pTransCmds,
01001     _In_ DWORD dwNumTrans);
01002
01014 BOOL DLLCALLCONV WDC_AddrSpaceIsActive(_In_ WDC_DEVICE_HANDLE hDev,
01015     _In_ DWORD dwAddrSpace);
01016
01040 DWORD DLLCALLCONV WDC_PciReadCfgBySlot(_In_ WD_PCI_SLOT *pPciSlot,
01041     _In_ DWORD dwOffset, _Outptr_ PVOID pData, _In_ DWORD dwBytes);
01042
01059 DWORD DLLCALLCONV WDC_PciWriteCfgBySlot(_In_ WD_PCI_SLOT *pPciSlot,
01060     _In_ DWORD dwOffset, _In_ PVOID pData, _In_ DWORD dwBytes);
01061
01062
01080 DWORD DLLCALLCONV WDC_PciReadCfg(_In_ WDC_DEVICE_HANDLE hDev,
01081     _In_ DWORD dwOffset, _Outptr_ PVOID pData, _In_ DWORD dwBytes);
01082
01097 DWORD DLLCALLCONV WDC_PciWriteCfg(_In_ WDC_DEVICE_HANDLE hDev,
01098     _In_ DWORD dwOffset, _In_ PVOID pData, _In_ DWORD dwBytes);
01099
01119 DWORD DLLCALLCONV WDC_PciReadCfgBySlot8(_In_ WD_PCI_SLOT *pPciSlot,
01120     _In_ DWORD dwOffset, _Outptr_ BYTE *pbVal);
01121
01138 DWORD DLLCALLCONV WDC_PciReadCfgBySlot16(_In_ WD_PCI_SLOT *pPciSlot,
01139     _In_ DWORD dwOffset, _Outptr_ WORD *pwVal);
01140
01157 DWORD DLLCALLCONV WDC_PciReadCfgBySlot32(_In_ WD_PCI_SLOT *pPciSlot,
01158     _In_ DWORD dwOffset, _Outptr_ UINT32 *pdwVal);
01159
01176 DWORD DLLCALLCONV WDC_PciReadCfgBySlot64(_In_ WD_PCI_SLOT *pPciSlot,
01177     _In_ DWORD dwOffset, _Outptr_ UINT64 *pqwVal);
01178
01194 DWORD DLLCALLCONV WDC_PciWriteCfgBySlot8(_In_ WD_PCI_SLOT *pPciSlot,
01195     _In_ DWORD dwOffset, _In_ BYTE bVal);
01196
01212 DWORD DLLCALLCONV WDC_PciWriteCfgBySlot16(_In_ WD_PCI_SLOT *pPciSlot,
01213     _In_ DWORD dwOffset, _In_ WORD wVal);
01214
01231 DWORD DLLCALLCONV WDC_PciWriteCfgBySlot32(_In_ WD_PCI_SLOT *pPciSlot,
01232     _In_ DWORD dwOffset, _In_ UINT32 dwVal);
01233
```

```
01249 DWORD DLLCALLCONV WDC_PciWriteCfgBySlot64(_In_ WD_PCI_SLOT *pPciSlot,
01250     _In_ DWORD dwOffset, _In_ UINT64 qwVal);
01251
01269 DWORD DLLCALLCONV WDC_PciReadCfg8(_In_ WDC_DEVICE_HANDLE hDev,
01270     _In_ DWORD dwOffset, _Outptr_ BYTE *pbVal);
01271
01286 DWORD DLLCALLCONV WDC_PciReadCfg16(_In_ WDC_DEVICE_HANDLE hDev,
01287     _In_ DWORD dwOffset, _Outptr_ WORD *pwVal);
01288
01303 DWORD DLLCALLCONV WDC_PciReadCfg32(_In_ WDC_DEVICE_HANDLE hDev,
01304     _In_ DWORD dwOffset, _Outptr_ UINT32 *pdwVal);
01305
01320 DWORD DLLCALLCONV WDC_PciReadCfg64(_In_ WDC_DEVICE_HANDLE hDev,
01321     _In_ DWORD dwOffset, _Outptr_ UINT64 *pqwVal);
01322
01336 DWORD DLLCALLCONV WDC_PciWriteCfg8(_In_ WDC_DEVICE_HANDLE hDev,
01337     _In_ DWORD dwOffset, _In_ BYTE bVal);
01338
01352 DWORD DLLCALLCONV WDC_PciWriteCfg16(_In_ WDC_DEVICE_HANDLE hDev,
01353     _In_ DWORD dwOffset, _In_ WORD wVal);
01354
01368 DWORD DLLCALLCONV WDC_PciWriteCfg32(_In_ WDC_DEVICE_HANDLE hDev,
01369     _In_ DWORD dwOffset, _In_ UINT32 dwVal);
01370
01384 DWORD DLLCALLCONV WDC_PciWriteCfg64(_In_ WDC_DEVICE_HANDLE hDev,
01385     _In_ DWORD dwOffset, _In_ UINT64 qwVal);
01386
01390 #if !defined(__KERNEL__)
01438 DWORD DLLCALLCONV WDC_DMAContigBufLock(_In_ WDC_DEVICE_HANDLE hDev,
01439     _Outptr_ PVOID *ppBuf, _In_ DWORD dwOptions, _In_ DWORD dwDMABufSize,
01440     _Outptr_ WD_DMA **ppDma);
01441
01477 DWORD DLLCALLCONV WDC_DMASGBufLock(_In_ WDC_DEVICE_HANDLE hDev,
01478     _In_ PVOID pBuf, _In_ DWORD dwOptions, _In_ DWORD dwDMABufSize,
01479     _Outptr_ WD_DMA **ppDma);
01480
01481
01482 typedef struct {
01483     WD_TRANSFER *pTransCmds;
01484     DWORD dwNumCmds;
01485     DWORD dwOptions;
01486     INT_HANDLER funcIntHandler;
01487     PVOID pData;
01488     BOOL fUseKP;
01489 } WDC_INTERRUPT_PARAMS;
01490
01535 DWORD DLLCALLCONV WDC_DMATransactionContigInit(_In_ WDC_DEVICE_HANDLE hDev,
01536     _Outptr_ PVOID *ppBuf, _In_ DWORD dwOptions, _In_ DWORD dwDMABufSize,
01537     _Outptr_ WD_DMA **ppDma, _In_ WDC_INTERRUPT_PARAMS *pInterruptParams,
01538     _In_ DWORD dwAlignment);
01539
01581 DWORD DLLCALLCONV WDC_DMATransactionSGInit(_In_ WDC_DEVICE_HANDLE hDev,
01582     _In_ PVOID pBuf, _In_ DWORD dwOptions, _In_ DWORD dwDMABufSize,
01583     _Outptr_ WD_DMA **ppDma, _In_ WDC_INTERRUPT_PARAMS *pInterruptParams,
01584     _In_ DWORD dwMaxTransferSize, _In_ DWORD dwTransferElementSize);
01585
01613 DWORD DLLCALLCONV WDC_DMATransactionExecute(_Inout_ WD_DMA *pDma,
01614     _In_ DMA_TRANSACTION_CALLBACK funcDMATransactionCallback,
01615     _In_ PVOID DMATransactionCallbackCtx);
01616
01642 DWORD DLLCALLCONV WDC_DMATransferCompletedAndCheck(_Inout_ WD_DMA *pDma,
01643     _In_ BOOL fRunCallback);
01644
01663 DWORD DLLCALLCONV WDC_DMATransactionRelease(_In_ WD_DMA *pDma);
01664
01684 DWORD DLLCALLCONV WDC_DMATransactionUninit(_In_ WD_DMA *pDma);
01685
01722 DWORD DLLCALLCONV WDC_DMAReservedBufLock(_In_ WDC_DEVICE_HANDLE hDev,
01723     _In_ PHYS_ADDR qwAddr, _Outptr_ PVOID *ppBuf, _In_ DWORD dwOptions,
01724     _In_ DWORD dwDMABufSize, _Outptr_ WD_DMA **ppDma);
01725
01726
01746 DWORD DLLCALLCONV WDC_DMABufUnlock(_In_ WD_DMA *pDma);
01747
01750 #define WDC_DMAGetGlobalHandle(pDma) ((pDma)->hDma)
01751
01767 DWORD DLLCALLCONV WDC_DMABufGet(_In_ DWORD hDma, _Outptr_ WD_DMA **ppDma);
01768
01787 DWORD DLLCALLCONV WDC_DMASyncCpu(_In_ WD_DMA *pDma);
01788
01807 DWORD DLLCALLCONV WDC_DMASyncIo(_In_ WD_DMA *pDma);
01808 #endif
01809 /* -----
01810     Interrupts
01811 ----- */
01812 #if !defined(__KERNEL__)
01949 DWORD DLLCALLCONV WDC_IntEnable(_In_ WDC_DEVICE_HANDLE hDev,
```

```

01950     _In_ WD_TRANSFER *pTransCmds, _In_ DWORD dwNumCmds, _In_ DWORD dwOptions,
01951     _In_ INT_HANDLER funcIntHandler, _In_ PVOID pData, _In_ BOOL fUseKP);
01952
01968 DWORD DLLCALLCONV WDC_IntDisable(_In_ WDC_DEVICE_HANDLE hDev);
01969
01970 #endif
01971
01986 BOOL DLLCALLCONV WDC_IntIsEnabled(_In_ WDC_DEVICE_HANDLE hDev);
01987
02002 const CHAR * DLLCALLCONV WDC_IntType2Str(_In_ DWORD dwIntType);
02003
02008 #if !defined(__KERNEL__)
02064 DWORD DLLCALLCONV WDC_EventRegister(_In_ WDC_DEVICE_HANDLE hDev,
02065     _In_ DWORD dwActions, _In_ EVENT_HANDLER funcEventHandler,
02066     _In_ PVOID pData, _In_ BOOL fUseKP);
02067
02081 DWORD DLLCALLCONV WDC_EventUnregister(_In_ WDC_DEVICE_HANDLE hDev);
02082
02083 #endif
02084
02098 BOOL DLLCALLCONV WDC_EventIsRegistered(_In_ WDC_DEVICE_HANDLE hDev);
02099
02140 DWORD DLLCALLCONV WDC_SetDebugOptions(_In_ WDC_DBG_OPTIONS dbgOptions,
02141     _In_ const CHAR *pcDbgFile);
02142
02154 void DLLCALLCONV WDC_Err(const CHAR *format, ...);
02155
02166 void DLLCALLCONV WDC_Trace(const CHAR *format, ...);
02167
02168 #ifdef __cplusplus
02169 }
02170 #endif
02171
02172 #endif /* _WDC_LIB_H */
02173

```

wds_lib.h File Reference

```
#include "windrvr.h"
#include "wdc_lib.h"
```

Data Structures

- struct [WDS_IPC_SCAN_RESULT](#)
IPC scan processes results.
- struct [WDS_IPC_MSG_RX](#)
IPC message received.

Macros

- #define [WDS_SharedBufferGetGlobalHandle](#)(pKerBuf) ((pKerBuf)->hKerBuf)
Utility macro that returns a kernel buffer global handle that can be used for buffer sharing between multiple processes.

Typedefs

- typedef void(* [IPC_MSG_RX_HANDLER](#)) (_In_ WDS_IPC_MSG_RX *plpcRxMsg, _In_ void *pData)
WinDriver IPC message handler callback.

Functions

- BOOL [DLLCALLCONV WDS_IsIpcRegistered](#) (void)
Enables the application to check if it is already registered with WinDriver IPC.
- DWORD [DLLCALLCONV WDS_IpcRegister](#) (_In_ const CHAR *pcProcessName, _In_ DWORD dwGroupID, _In_ DWORD dwSubGroupID, _In_ DWORD dwAction, _In_ IPC_MSG_RX_HANDLER pFunc, _In_ void *pData)
Registers an application with WinDriver IPC.

- void **DLLCALLCONV WDS_IpcUnRegister** (void)
This function enables the user application to unregister with WinDriver IPC.
- DWORD **DLLCALLCONV WDS_IpcScanProcs** (**_Outptr_ WDS_IPC_SCAN_RESULT** *plpcScanResult)
*Scans and returns information of all registered processes that share the application process groupID (as was given to **WDS_IpcRegister()** or a specific groupID.)*
- DWORD **DLLCALLCONV WDS_IpcUidUnicast** (**_In_** DWORD dwRecipientUID, **_In_** DWORD dwMsgID, **_In_** **UINT64** qwMsgData)
Sends a message to a specific process with WinDriver IPC unique ID.
- DWORD **DLLCALLCONV WDS_IpcSubGroupMulticast** (**_In_** DWORD dwRecipientSubGroupID, **_In_** DWORD dwMsgID, **_In_** **UINT64** qwMsgData)
Sends a message to all processes that registered with the same sub-group ID.
- DWORD **DLLCALLCONV WDS_IpcMulticast** (**_In_** DWORD dwMsgID, **_In_** **UINT64** qwMsgData)
Sends a message to all processes that were registered with the same group ID as the sending process.
- DWORD **DLLCALLCONV WDS_SharedBufferAlloc** (**_In_** **UINT64** qwBytes, **_In_** DWORD dwOptions, **_Outptr_ WD_KERNEL_BUFFER** **ppKerBuf)
Allocates a memory buffer that can be shared between the user mode and the kernel mode ("shared buffer"), and returns user-mode and kernel-mode virtual address space mappings of the allocated buffer.
- DWORD **DLLCALLCONV WDS_SharedBufferGet** (**_In_** DWORD hKerBuf, **_Outptr_ WD_KERNEL_BUFFER** **ppKerBuf)
Retrieves a shared buffer which was allocated by another process.
- DWORD **DLLCALLCONV WDS_SharedBufferFree** (**_In_** WD_KERNEL_BUFFER *pKerBuf)
*Frees a shared buffer that was allocated by a previous call to **WDS_SharedBufferAlloc()**.*
- DWORD **DLLCALLCONV WDS_SharedIntEnable** (**_In_** const CHAR *pcProcessName, **_In_** DWORD dwGroupID, **_In_** DWORD dwSubGroupID, **_In_** DWORD dwAction, **_In_** IPC_MSG_RX_HANDLER pFunc, **_In_** void *pData)
Enables the shared interrupts mechanism of WinDriver.
- DWORD **DLLCALLCONV WDS_SharedIntDisableGlobal** (void)
Disables the Shared Interrupts mechanism of WinDriver for all processes.
- DWORD **DLLCALLCONV WDS_SharedIntDisableLocal** (void)
Disables the Shared Interrupts mechanism of WinDriver for the current process.
- BOOL **DLLCALLCONV WDS_IsSharedIntsEnabledLocally** (void)
Check and returns whether shared interrupts are enabled for the current process.

Macro Definition Documentation

WDS_SharedBufferGetGlobalHandle

```
#define WDS_SharedBufferGetGlobalHandle( pKerBuf ) ((pKerBuf)->hKerBuf)
```

Utility macro that returns a kernel buffer global handle that can be used for buffer sharing between multiple processes.

Parameters

in	<i>pKerBuf</i>	Pointer to a kernel buffer information structure
----	----------------	--

Returns

Returns a Kernel buffer handle of pKerBuf.

Definition at line 287 of file [wds_lib.h](#).

Typedef Documentation**IPC_MSG_RX_HANDLER**

```
typedef void(* IPC_MSG_RX_HANDLER) (_In_ WDS_IPC_MSG_RX *pIpcRxMsg, _In_ void *pData)
```

WinDriver IPC message handler callback.

Parameters

in	<i>plpcRxMsg</i>	Pointer to the received IPC message
in	<i>pData</i>	Application specific data opaque as passed to WDS_IpcRegister()

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 67 of file [wds_lib.h](#).

Function Documentation**WDS_IpcMulticast()**

```
DWORD DLLCALLCONV WDS_IpcMulticast (
    _In_ DWORD dwMsgID,
    _In_ UINT64 qwMsgData )
```

Sends a message to all processes that were registered with the same group ID as the sending process.
The message won't be sent to the sending process.

Parameters

in	<i>dwMsgID</i>	A 32 bit unique number defined by the user application. This number should be known to all user-applications that work under WinDriver IPC and share the same group ID
in	<i>qwMsgData</i>	Optional - 64 bit additional data from the sending user-application

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
DWORD ipc_menu_option; /* Can be MENU_IPC_SEND_UID_UNICAST
                           MENU_IPC_SEND_SUBGROUP_MULTICAST
                           MENU_IPC_SEND_MULTICAST */

DWORD recipientID = 0x1;
DWORD messageID = 0x64;
UINT64 messageData = 0x32;
DWORD dwStatus;
switch (ipc_menu_option)
{
    case MENU_IPC_SEND_UID_UNICAST:
        dwStatus = WDS_IpcUidUnicast(recipientID, messageID, messageData);
        break;
    case MENU_IPC_SEND_SUBGROUP_MULTICAST:
        dwStatus = WDS_IpcSubGroupMulticast(recipientID, messageID,
                                             messageData);
        break;
    case MENU_IPC_SEND_MULTICAST:
        break;
}
```

```

dwStatus = WDS_IpcMulticast(messageID, messageData);
break;
}
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed sending message. Error [0x%lx - %s]\n",
           dwStatus, Stat2Str(dwStatus));
    return;
}
printf("Message sent successfully\n");

```

WDS_IpcRegister()

```

DWORD DLLCALLTYPE WDS_IpcRegister (
    _In_ const CHAR * pcProcessName,
    _In_ DWORD dwGroupID,
    _In_ DWORD dwSubGroupID,
    _In_ DWORD dwAction,
    _In_ IPC_MSG_RX_HANDLER pFunc,
    _In_ void * pData )

```

Registers an application with WinDriver IPC.

Parameters

in	<i>pcProcessName</i>	Optional process name string
in	<i>dwGroupID</i>	A unique group ID represent the specific application. Must be a positive ID
in	<i>dwSubGroupID</i>	Sub-group ID that should identify your user application type in case you have several types that may work simultaneously. Must be a positive ID
in	<i>dwAction</i>	IPC message type to receive, which can consist one of the enumeration values listed below: WD_IPC_UNICAST_MSG: Receive a message to a specific process with WinDriver IPC unique ID WD_IPC_MULTICAST_MSG: Receive a message from all processes that were registered with the same group ID as this process WD_IPC_ALL_MSG: Receive both types of the messages above
in	<i>pFunc</i>	A user-mode IPC message handler callback function, which will be called when a message was received by WinDriver from a different process (see <i>dwActions</i>) occurs. (See IPC_MSG_RX_HANDLER())
in	<i>pData</i>	Data for the user-mode IPC message handler callback routine (<i>pFunc</i>)

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

You should choose your user applications a unique group ID parameter. This is done as a precaution to prevent several applications that use WinDriver with its default driver name (windrvrXXXX) to get mixing messages. We strongly recommend that you rename your driver before distributing it to avoid this issue entirely, among other issue (See Section 15.2 on renaming you driver name). The sub-group id parameter should identify your user application type in case you have several types that may work simultaneously.

```

// WinDriver Callback functions must use DLLCALLTYPE macro
static void DLLCALLTYPE ipc_msg_event_cb(WDS_IPC_MSG_RX *pIpcRxMsg, void *pData)
{
    printf("\n\nReceived an IPC message:\n"
           "msgID [0x%lx], msgData [0x%lx] from process [0x%lx]\n",
           pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData, pIpcRxMsg->dwSenderUID);
    switch (pIpcRxMsg->dwMsgID)
    {
        case A:
        {
            // ...
        }
        break;
        case B:
        {
    
```

```

        // ...
    }
    break;
default:
    printf("Unknown IPC type. msgID [0x%lx], msgData [0x%llx] from "
        "process [0x%lx]\n\n", pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData,
        pIpcRxMsg->dwSenderUID);
}
int main(void)
{
    DWORD dwSubGroupID = 0; // Or any other ID you want to choose
    DWORD dwStatus;
    WDS_IPC_SCAN_RESULT ipcScanResult;
    // Make sure to fully initialize WinDriver!
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        ErrLog("Failed to set the driver name for WDC library.\n");
        return;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    dwStatus = WDS_IpcRegister(DEFAULT_PROCESS_NAME, DEFAULT_PROCESS_GROUP_ID,
        dwSubGroupID, WD_IPC_ALL_MSG, ipc_msg_event_cb, NULL /* Your cb ctx */);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed registering process to IPC. Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    printf("Registration completed successfully\n");
    printf("Scanning for processes...\n");
    dwStatus = WDS_IpcScanProcs(&ipcScanResult);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed scanning registered processes. Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    if (ipcScanResult.dwNumProcs)
    {
        printf("Found %ld processes in current group\n",
            ipcScanResult.dwNumProcs);
        for (i = 0; i < ipcScanResult.dwNumProcs; i++)
        {
            printf(" %lu) Name: %s, SubGroup ID: 0x%lx, UID: 0x%lx\n",
                i + 1,
                ipcScanResult.procInfo[i].cProcessName,
                ipcScanResult.procInfo[i].dwSubGroupID,
                ipcScanResult.procInfo[i].hIpc);
        }
    }
    else
    {
        printf("No processes found in current group\n");
    }
    // ...
    // Rest of your application code
    // ...
    dwStatus = WDS_IpcUnRegister();
    if (dwStatus)
    {
        printf("Failed unregistering IPC Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
Exit:
    WDC_DriverClose();
}

```

WDS_IpcScanProcs()

DWORD **DLLCALLCONV** WDS_IpcScanProcs (
 Outptr WDS_IPC_SCAN_RESULT * *pIpcScanResult*)

Scans and returns information of all registered processes that share the application process groupID (as was given to [WDS_IpcRegister\(\)](#) or a specific groupID.)

Parameters

out	<i>plpcScanResult</i>	Pointer to IpcScanResult struct that will be filled by the function.
-----	-----------------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
// WinDriver Callback functions must use DLLCALLCONV macro
static void DLLCALLCONV ipc_msg_event_cb(WDS_IPC_MSG_RX *pIpcRxMsg, void *pData)
{
    printf("\n\nReceived an IPC message:\n"
        "msgID [0x%lx], msgData [0x%llx] from process [0x%lx]\n",
        pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData, pIpcRxMsg->dwSenderUID);
    switch (pIpcRxMsg->dwMsgID)
    {
        case A:
        {
            // ...
            break;
        }
        case B:
        {
            // ...
            break;
        }
        default:
        {
            printf("Unknown IPC type. msgID [0x%lx], msgData [0x%llx] from "
                "process [0x%lx]\n\n", pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData,
                pIpcRxMsg->dwSenderUID);
        }
    }
}
int main(void)
{
    DWORD dwSubGroupID = 0; // Or any other ID you want to choose
    DWORD dwStatus;
    WDS_IPC_SCAN_RESULT ipcScanResult;
    // Make sure to fully initialize WinDriver!
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        ErrLog("Failed to set the driver name for WDC library.\n");
        return;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
            dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    dwStatus = WDS_IpcRegister(DEFAULT_PROCESS_NAME, DEFAULT_PROCESS_GROUP_ID,
        dwSubGroupID, WD_IPC_ALL_MSG, ipc_msg_event_cb, NULL /* Your cb ctx */);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed registering process to IPC. Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    printf("Registration completed successfully\n");
    printf("Scanning for processes...\n");
    dwStatus = WDS_IpcScanProcs(&ipcScanResult);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed scanning registered processes. Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    if (ipcScanResult.dwNumProcs)
    {
        printf("Found %ld processes in current group\n",
            ipcScanResult.dwNumProcs);
        for (i = 0; i < ipcScanResult.dwNumProcs; i++)
        {
            printf(" %lu) Name: %s, SubGroup ID: 0x%lx, UID: 0x%lx\n",
                i + 1,
                ipcScanResult.procInfo[i].cProcessName,
                ipcScanResult.procInfo[i].dwSubGroupID,
                ipcScanResult.procInfo[i].hIpc);
        }
    }
    else
    {
        printf("No processes found in current group\n");
    }
}
```

```

// ...
// Rest of your application code
// ...
dwStatus = WDS_IpcUnRegister();
if (dwStatus)
{
    printf("Failed unregistering IPC Error [0x%lx - %s]\n",
        dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
Exit:
    WDC_DriverClose();
}

```

WDS_IpcSubGroupMulticast()

DWORD **DLLCALLCONV** WDS_IpcSubGroupMulticast (
 In DWORD dwRecipientSubGroupID,
 In DWORD dwMsgID,
 In **UINT64** qwMsgData)

Sends a message to all processes that registered with the same sub-group ID.

Parameters

in	dwRecipientSubGroupID	Recipient sub-group ID that should identify your user application type in case you have several types that may work simultaneously.
in	dwMsgID	A 32 bit unique number defined by the user application. This number should be known to all user-applications that work under WinDriver IPC and share the same group ID
in	qwMsgData	Optional - 64 bit additional data from the sending user-application

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```

DWORD ipc_menu_option; /* Can be MENU_IPC_SEND_UID_UNICAST
                        MENU_IPC_SEND_SUBGROUP_MULTICAST
                        MENU_IPC_SEND_MULTICAST */

DWORD recipientID = 0x1;
DWORD messageID = 0x64;
UINT64 messageData = 0x32;
DWORD dwStatus;
switch (ipc_menu_option)
{
case MENU_IPC_SEND_UID_UNICAST:
    dwStatus = WDS_IpcUidUnicast(recipientID, messageID, messageData);
    break;
case MENU_IPC_SEND_SUBGROUP_MULTICAST:
    dwStatus = WDS_IpcSubGroupMulticast(recipientID, messageID,
        messageData);
    break;
case MENU_IPC_SEND_MULTICAST:
    dwStatus = WDS_IpcMulticast(messageID, messageData);
    break;
}
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed sending message. Error [0x%lx - %s]\n",
        dwStatus, Stat2Str(dwStatus));
    return;
}
printf("Message sent successfully\n");

```

WDS_IpcUidUnicast()

DWORD **DLLCALLCONV** WDS_IpcUidUnicast (
 In DWORD dwRecipientUID,
 In DWORD dwMsgID,
 In **UINT64** qwMsgData)

Sends a message to a specific process with WinDriver IPC unique ID.

Parameters

in	<i>dwRecipientUID</i>	WinDriver IPC unique ID that should identify one of your user application. The recipient UID can be obtained from the result of WDS_IpcScanProcs() or the sender ID as received in the callback registered in WDS_IpcRegister()
in	<i>dwMsgID</i>	A 32 bit unique number defined by the user application. This number should be known to all user-applications that work under WinDriver IPC and share the same group ID
in	<i>qwMsgData</i>	Optional - 64 bit additional data from the sending user-application

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
DWORD ipc_menu_option; /* Can be MENU_IPC_SEND_UID_UNICAST
                        MENU_IPC_SEND_SUBGROUP_MULTICAST
                        MENU_IPC_SEND_MULTICAST */

DWORD recipientID = 0x1;
DWORD messageID = 0x64;
UINT64 messageData = 0x32;
DWORD dwStatus;
switch (ipc_menu_option)
{
case MENU_IPC_SEND_UID_UNICAST:
    dwStatus = WDS_IpcUidUnicast(recipientID, messageID, messageData);
    break;
case MENU_IPC_SEND_SUBGROUP_MULTICAST:
    dwStatus = WDS_IpcSubGroupMulticast(recipientID, messageID,
                                         messageData);
    break;
case MENU_IPC_SEND_MULTICAST:
    dwStatus = WDS_IpcMulticast(messageID, messageData);
    break;
}
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed sending message. Error [0x%lx - %s]\n",
           dwStatus, Stat2Str(dwStatus));
    return;
}
printf("Message sent successfully\n");
```

WDS_IpcUnRegister()

```
void DLLCALLCONV WDS_IpcUnRegister (
    void )
```

This function enables the user application to unregister with WinDriver IPC.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
// WinDriver Callback functions must use DLLCALLCONV macro
static void DLLCALLCONV ipc_msg_event_cb(WDS_IPC_MSG_RX *pIpcRxMsg, void *pData)
{
    printf("\n\nReceived an IPC message:\n"
           "msgID [0x%lx], msgData [0x%llx] from process [0x%lx]\n",
           pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData, pIpcRxMsg->dwSenderUID);
    switch (pIpcRxMsg->dwMsgID)
    {
    case A:
        {
            // ...
        }
        break;
    case B:
        {
            // ...
        }
        break;
    default:
        printf("Unknown IPC type. msgID [0x%lx], msgData [0x%llx] from "
               "process [0x%lx]\n\n", pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData,
               pIpcRxMsg->dwSenderUID);
    }
}
```

```
}

int main(void)
{
    DWORD dwSubGroupID = 0; // Or any other ID you want to choose
    DWORD dwStatus;
    WDS_IPC_SCAN_RESULT ipcScanResult;
    // Make sure to fully initialize WinDriver!
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        ErrLog("Failed to set the driver name for WDC library.\n");
        return;
    }
    /* Open a handle to the driver and initialize the WDC library */
    dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
               dwStatus, Stat2Str(dwStatus));
        return dwStatus;
    }
    dwStatus = WDS_IpcRegister(DEFAULT_PROCESS_NAME, DEFAULT_PROCESS_GROUP_ID,
                               dwSubGroupID, WD_IPC_ALL_MSG, ipc_msg_event_cb, NULL /* Your cb ctx */);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed registering process to IPC. Error [0x%lx - %s]\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    printf("Registration completed successfully\n");
    printf("Scanning for processes...\n");
    dwStatus = WDS_IpcScanProcs(&ipcScanResult);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed scanning registered processes. Error [0x%lx - %s]\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
    if (ipcScanResult.dwNumProcs)
    {
        printf("Found %ld processes in current group\n",
               ipcScanResult.dwNumProcs);
        for (i = 0; i < ipcScanResult.dwNumProcs; i++)
        {
            printf(" %lu) Name: %s, SubGroup ID: 0x%lx, UID: 0x%lx\n",
                   i + 1,
                   ipcScanResult.procInfo[i].cProcessName,
                   ipcScanResult.procInfo[i].dwSubGroupID,
                   ipcScanResult.procInfo[i].hIpc);
        }
    }
    else
    {
        printf("No processes found in current group\n");
    }
    // ...
    // Rest of your application code
    // ...
    dwStatus = WDS_IpcUnRegister();
    if (dwStatus)
    {
        printf("Failed unregistering IPC Error [0x%lx - %s]\n",
               dwStatus, Stat2Str(dwStatus));
        goto Exit;
    }
Exit:
    WDC_DriverClose();
}
```

WDS_IsIpcRegistered()

```
BOOL DLLCALLCONV WDS_IsIpcRegistered (
    void )
```

Enables the application to check if it is already registered with WinDriver IPC.

Returns

Returns TRUE if successful; otherwise returns FALSE.

WDS_IsSharedIntsEnabledLocally()

```
BOOL DLLCALLCONV WDS_IsSharedIntsEnabledLocally (
    void )
```

Check and returns whether shared interrupts are enabled for the current process.

Returns

TRUE if shared interrupts are enabled, else FALSE

WDS_SharedBufferAlloc()

```
DWORD DLLCALLCONV WDS_SharedBufferAlloc (
    _In_ UINT64 qwBytes,
    _In_ DWORD dwOptions,
    _Outptr_ WD_KERNEL_BUFFER ** ppKerBuf )
```

Allocates a memory buffer that can be shared between the user mode and the kernel mode ("shared buffer"), and returns user-mode and kernel-mode virtual address space mappings of the allocated buffer.

Parameters

in	<i>qwBytes</i>	The size of the buffer to allocate, in bytes
in	<i>dwOptions</i>	<p>Kernel buffer options bit-mask, which can consist of a combination of the enumeration values listed below.</p> <ul style="list-style-type: none"> • KER_BUF_ALLOC_NON_CONTIG: Allocates a non contiguous buffer • KER_BUF_ALLOC_CONTIG: Allocates a physically contiguous buffer • KER_BUF_ALLOC_CACHED: Allocates a cached buffer. This option can be set with KER_BUF_ALLOC_NON_CONTIG or KER_BUF_ALLOC_CONTIG buffer
out	<i>ppKerBuf</i>	Pointer to a WD_KERNEL_BUFFER pointer, to be filled by the function. The caller should use *ppBuf->pUserAddr usermode mapped address. When the buffer is no longer needed, (*ppBuf) should be passed to WDS_SharedBufferFree()

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

This function is currently only supported from the user mode. This function is supported only for Windows and Linux.

```
// Static global pointer is used only for simplicity
static WD_KERNEL_BUFFER *pSharedKerBuf = NULL;
static void IpcKerBufAllocAndShare(void)
{
    DWORD size = 0x100;
    DWORD dwStatus;
    DWORD dwOptions = KER_BUF_ALLOC_CONTIG; // Or any other WD_KER_BUF_OPTION
    /* If kernel buffer was allocated in the past, release it */
    dwStatus = WDS_SharedBufferFree(pSharedKerBuf);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed freeing shared buffer. Error [0x%lx - %s]\n",
            dwStatus, Stat2Str(dwStatus));
```

```

    }
    dwStatus = WDS_SharedBufferAlloc(size, dwOptions, &pSharedKerBuf);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed allocating shared kernel buffer. size [%lu], "
               "Error [0x%lx - %s]\n", size, dwStatus, Stat2Str(dwStatus));
        return;
    }
    printf("Successful kernel buffer allocation. UserAddr [0x%"UPRI"x], "
           "KernelAddr [0x%"KPRI"x], size [%lu]\n", pSharedKerBuf->pUserAddr,
           pSharedKerBuf->pKernelAddr, size);
}

```

WDS_SharedBufferFree()

DWORD **DLLCALLCONV** WDS_SharedBufferFree (
 In WD_KERNEL_BUFFER * pKerBuf)

Frees a shared buffer that was allocated by a previous call to [WDS_SharedBufferAlloc\(\)](#).

Parameters

in	<i>pKerBuf</i>	Pointer to a WD_KERNEL_BUF structure, received within the *ppBuf parameter of a previous call to WDS_SharedBufferAlloc()
----	----------------	--

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

This function is currently only supported from the user mode. This function is supported only for Windows and Linux.

```
dwStatus = WDS_SharedBufferFree(pSharedKerBuf);
```

WDS_SharedBufferGet()

DWORD **DLLCALLCONV** WDS_SharedBufferGet (
 In DWORD hKerBuf,
Outptr WD_KERNEL_BUFFER ** ppKerBuf)

Retrieves a shared buffer which was allocated by another process.

Parameters

in	<i>hKerBuf</i>	Kernel buffer handle.
out	<i>ppKerBuf</i>	Pointer to a pointer to a kernel buffer information structure, which is associated with hKerBuf

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

WDS_SharedIntDisableGlobal()

DWORD **DLLCALLCONV** WDS_SharedIntDisableGlobal (
 void)

Disables the Shared Interrupts mechanism of WinDriver for all processes.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WDS_SharedIntDisableGlobal();
```

WDS_SharedIntDisableLocal()

```
DWORD DLLCALLCONV WDS_SharedIntDisableLocal (
    void )
```

Disables the Shared Interrupts mechanism of WinDriver for the current process.
This function does not disable the Shared Interrupts mechanism for all processes.

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WDS_SharedIntDisableLocal();
```

WDS_SharedIntEnable()

```
DWORD DLLCALLCONV WDS_SharedIntEnable (
    _In_ const CHAR * pcProcessName,
    _In_ DWORD dwGroupID,
    _In_ DWORD dwSubGroupID,
    _In_ DWORD dwAction,
    _In_ IPC_MSG_RX_HANDLER pFunc,
    _In_ void * pData )
```

Enables the shared interrupts mechanism of WinDriver.

If the mechanism is already enabled globally (for all processes) then the mechanism is enabled for the current process.

Parameters

in	<i>pcProcessName</i>	Optional process name string
in	<i>dwGroupID</i>	A unique group ID represent the specific application. Must be a positive ID
in	<i>dwSubGroupID</i>	Sub-group ID that should identify your user application type in case you have several types that may work simultaneously. Must be a positive ID
in	<i>dwAction</i>	IPC message type to receive, which can consist one of the enumeration values listed below: WD_IPC_UNICAST_MSG: Receive a message to a specific process with WinDriver IPC unique ID WD_IPC_MULTICAST_MSG: Receive a message from all processes that were registered with the same group ID as this process WD_IPC_ALL_MSG: Receive both types of the messages above
in	<i>pFunc</i>	A user-mode IPC message handler callback function, which will be called when a message was received by WinDriver from Shared Interrupts IPC process occurs. (See IPC_MSG_RX_HANDLER())
in	<i>pData</i>	Data for the user-mode IPC message handler callback routine (pFunc)

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
static void ipc_shared_int_msg_event_cb(WDS_IPC_MSG_RX *pIpcRxMsg, void *pData)
{
    printf("Shared Interrupt via IPC arrived:\nmsgID [0x%lx], msgData [0x%llx]"
        " from process [0x%lx]\n\n", pIpcRxMsg->dwMsgID, pIpcRxMsg->qwMsgData,
        pIpcRxMsg->dwSenderUID);
}
static DWORD IpcSharedIntsEnable(void)
{
    DWORD dwSubGroupID = 0; // Or any other you choose
    DWORD dwStatus;
```

```
if (WDS_IsSharedIntsEnabledLocally())
{
    printf("%s: Shared interrupts already enabled locally.\n",
           __FUNCTION__);
    return WD_OPERATION_ALREADY_DONE;
}
/* WDS_SharedIntEnable() is called in this sample with pFunc=
ipc_shared_int_msg_event_cb. This will cause a shared interrupt to invoke
both this callback and the callback passed to WDS_IpcRegister().
To disable the "general" IPC callback, pass pFunc=NULL in the above
mentioned call.
Note you can replace pFunc here with your own callback especially designed
to handle interrupts */
dwStatus = WDS_SharedIntEnable(DEFAULT_SHARED_INT_NAME,
                                DEFAULT_PROCESS_GROUP_ID, dwSubGroupID, WD_IPC_ALL_MSG,
                                ipc_shared_int_msg_event_cb, NULL /* Your cb ctx */);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("%s: Failed enabling shared interrupts via IPC. "
           "Error [0x%lx - %s]\n", __FUNCTION__, dwStatus, Stat2Str(dwStatus));
    return 0;
}
printf("Shared interrupts via IPC enabled successfully\n");
return dwSubGroupID;
}
```

wds_lib.h

Go to the documentation of this file.

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WDS_LIB_H_
00004 #define _WDS_LIB_H_
00005
00006 /*****
00007 * File: wds_lib.h - WD Shared (WDS) library header.
00008 *          This file defines the WDS library's high-level
00009 *          interface
00010 *****/
00011
00012 #if defined(__KERNEL__)
00013     #include "kpstdlib.h"
00014 #endif
00015
00016 #include "windrvr.h"
00017 #include "wdc_lib.h"
00018
00019 #ifdef __cplusplus
00020     extern "C" {
00021 #endif
00022
00023 /*****
00024     General definitions
00025 *****/
00026
00027 /* -----
00028     IPC
00029     ----- */
00030 /* IPC API functions are not part of the standard WinDriver API, and not
00031 * included in the standard version of WinDriver. The functions are part of
00032 * "WinDriver for Server" API and require "WinDriver for Server" license.
00033 * Note that "WinDriver for Server" APIs are included in WinDriver evaluation
00034 * version. */
00035
00036 typedef struct {
00037     DWORD         dwNumProcs;
00038     WD_IPC_PROCESS procInfo[WD_IPC_MAX_PROCS];
00039 } WDS_IPC_SCAN_RESULT;
00040
00041
00042 typedef struct {
00043     DWORD         dwSenderUID;
00044     DWORD         dwMsgID;
00045     UINT64        qwMsgData;
00046 } WDS_IPC_MSG_RX;
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064
00065
00066
00067
00068
00069
00070
00071
00072
00073
00074
00075
00076
00077
00078
00079
00080
00081
00082
00083
00084
00085
00086
00087
00088
00089
00090
00091
00092
00093
00094
00095
00096
00097
00098
00099
00100
00101
00102
00103
00104
00105
00106
00107
00108
00109
00110
00111
00112
00113
00114
00115
00116
00117
00118
00119
00120
00121
00122
00123
00124
00125
00126
```

```

00127
00138 void DLLCALLCONV WDS_IpcUnRegister(void);
00139
00155 DWORD DLLCALLCONV WDS_IpcScanProcs(
00156     _Outptr_ WDS_IPC_SCAN_RESULT *pIpcScanResult);
00157
00158
00183 DWORD DLLCALLCONV WDS_IpcUidUnicast(_In_ DWORD dwRecipientUID,
00184     _In_ DWORD dwMsgID, _In_ UINT64 qwMsgData);
00185
00208 DWORD DLLCALLCONV WDS_IpcSubGroupMulticast(_In_ DWORD dwRecipientSubGroupID,
00209     _In_ DWORD dwMsgID, _In_ UINT64 qwMsgData);
00210
00229 DWORD DLLCALLCONV WDS_IpcMulticast(_In_ DWORD dwMsgID, _In_ UINT64 qwMsgData);
00230
00231 /* -----
00232     Shared Buffers (User-Mode <-> Kernel Mode) / (User-Mode <-> User-Mode)
00233 ----- */
00234 /*
00235     * Kernel buffers can be used to share data between:
00236     * 1) User-mode application and a Kernel PlugIn driver.
00237     * 2) Multiple user-mode applications.
00238 */
00239
00273 DWORD DLLCALLCONV WDS_SharedBufferAlloc(_In_ UINT64 qwBytes,
00274     _In_ DWORD dwOptions, _Outptr_ WD_KERNEL_BUFFER **ppKerBuf);
00275
00276
00287 #define WDS_SharedBufferGetGlobalHandle(pKerBuf) ((pKerBuf)->hKerBuf)
00288
00300 DWORD DLLCALLCONV WDS_SharedBufferGet(_In_ DWORD hKerBuf,
00301     _Outptr_ WD_KERNEL_BUFFER **ppKerBuf);
00302
00303
00322 DWORD DLLCALLCONV WDS_SharedBufferFree(_In_ WD_KERNEL_BUFFER *pKerBuf);
00323
00360 DWORD DLLCALLCONV WDS_SharedIntEnable(_In_ const CHAR *pcProcessName,
00361     _In_ DWORD dwGroupID, _In_ DWORD dwSubGroupID, _In_ DWORD dwAction,
00362     _In_ IPC_MSG_RX_HANDLER pFunc, _In_ void *pData);
00363
00373 DWORD DLLCALLCONV WDS_SharedIntDisableGlobal(void);
00374
00375
00389 DWORD DLLCALLCONV WDS_SharedIntDisableLocal(void);
00390
00391
00400 BOOL DLLCALLCONV WDS_IsSharedIntsEnabledLocally(void);
00401
00402 #ifdef __cplusplus
00403 }
00404 #endif
00405
00406 #endif /* _WDS_LIB_H_ */
00407

```

wdu_lib.h File Reference

```
#include "windrvr.h"
```

Data Structures

- struct [WDU_EVENT_TABLE](#)

Typedefs

- typedef PVOID [WDU_DRIVER_HANDLE](#)
- typedef PVOID [WDU_DEVICE_HANDLE](#)
- typedef PVOID [WDU_STREAM_HANDLE](#)
- typedef WORD [WDU_LANGID](#)
- typedef BOOL([DLLCALLCONV](#) * [WDU_ATTACH_CALLBACK](#)) (_In_ [WDU_DEVICE_HANDLE](#) hDevice, _In_ [WDU_DEVICE](#) *pDeviceInfo, _In_ PVOID pUserData)

WindDriver calls this function with any new device that is attached, matches the given criteria, and if WD_← ACKNOWLEDGE was passed to [WDU_Init\(\)](#) in dwOptions - not controlled yet by another driver.

- **typedef void(DLLCALLCONV * WDU_DETACH_CALLBACK) (_In_ WDU_DEVICE_HANDLE hDevice, _In_ PVOID pUserData)**
WinDriver calls this function when a controlled device has been detached from the system.
- **typedef BOOL(DLLCALLCONV * WDU_POWER_CHANGE_CALLBACK) (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPowerState, _In_ PVOID pUserData)**

Functions

- **DWORD DLLCALLCONV WDU_Init (_Outptr_ WDU_DRIVER_HANDLE *phDriver, _In_ WDU_MATCH_TABLE *pMatchTables, _In_ DWORD dwNumMatchTables, _In_ WDU_EVENT_TABLE *pEventTable, _In_ const char *pcLicense, _In_ DWORD dwOptions)**
Starts listening to devices matching a criteria, and registers notification callbacks for those devices.
- **void DLLCALLCONV WDU_Uninit (_In_ WDU_DRIVER_HANDLE hDriver)**
Stops listening to devices matching the criteria, and unregisters the notification callbacks for those devices.
- **DWORD DLLCALLCONV WDU_GetDeviceAddr (_In_ WDU_DEVICE_HANDLE hDevice, _Out_ DWORD *pAddress)**
Gets USB address that the device uses.
- **DWORD DLLCALLCONV WDU_GetDeviceRegistryProperty (_In_ WDU_DEVICE_HANDLE hDevice, _Outptr_ PVOID pBuffer, _Inout_ PDWORD pdwSize, _In_ WD_DEVICE_REGISTRY_PROPERTY property)**
Gets the specified registry property of a given device.
- **DWORD DLLCALLCONV WDU_GetDeviceInfo (_In_ WDU_DEVICE_HANDLE hDevice, _Outptr_ WDU_DEVICE **ppDeviceInfo)**
Gets configuration information from the device including all the descriptors in a [WDU_DEVICE](#) struct.
- **void DLLCALLCONV WDU_PutDeviceInfo (_In_ WDU_DEVICE *pDeviceInfo)**
Receives a device information pointer, allocated with a previous [WDU_GetDeviceInfo\(\)](#) call, in order to perform the necessary cleanup.
- **DWORD DLLCALLCONV WDU_SetInterface (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwInterfaceNum, _In_ DWORD dwAlternateSetting)**
Sets the alternate setting for the specified interface.
- **DWORD DLLCALLCONV WDU_SetConfig (WDU_DEVICE_HANDLE hDevice, DWORD dwConfigNum)**
- **DWORD DLLCALLCONV WDU_ResetPipe (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum)**
Resets a pipe.
- **DWORD DLLCALLCONV WDU_ResetDevice (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwOptions)**
Resets a device (supported only on Windows).
- **DWORD DLLCALLCONV WDU_Wakeup (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwOptions)**
Enables/Disables wakeup feature.
- **DWORD DLLCALLCONV WDU_SelectiveSuspend (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwOptions)**
Submits a request to suspend a given device (selective suspend), or cancels a previous suspend request.
- **DWORD DLLCALLCONV WDU_Transfer (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions, _In_ PVOID pBuffer, _In_ DWORD dwBufferSize, _Outptr_ PDWORD pdwBytesTransferred, _In_ PBYTE pSetupPacket, _In_ DWORD dwTimeout)**
Transfers data to/from a device.
- **DWORD DLLCALLCONV WDU_HaltTransfer (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum)**
Halts the transfer on the specified pipe (only one simultaneous transfer per-pipe is allowed by WinDriver).
- **DWORD DLLCALLCONV WDU_TransferDefaultPipe (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD fRead, _In_ DWORD dwOptions, _In_ PVOID pBuffer, _In_ DWORD dwBufferSize, _Outptr_ PDWORD pdwBytesTransferred, _In_ PBYTE pSetupPacket, _In_ DWORD dwTimeout)**
- **DWORD DLLCALLCONV WDU_TransferBulk (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions, _In_ PVOID pBuffer, _In_ DWORD dwBufferSize, _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout)**

- DWORD **DLLCALLCONV WDU_TransferIsoch** (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions, _In_ PVOID pBuffer, _In_ DWORD dwBufferSize, _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout)
- DWORD **DLLCALLCONV WDU_TransferInterrupt** (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions, _In_ PVOID pBuffer, _In_ DWORD dwBufferSize, _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout)
- DWORD **DLLCALLCONV WDU_GetLangIDs** (_In_ WDU_DEVICE_HANDLE hDevice, _Outptr_ PBYTE pbNumSupportedLangIDs, _Outptr_ WDU_LANGID *pLangIDs, _In_ BYTE bNumLangIDs)

Reads a list of supported language IDs and/or the number of supported language IDs from a device.
- DWORD **DLLCALLCONV WDUGetStringDesc** (_In_ WDU_DEVICE_HANDLE hDevice, _In_ BYTE bStringIndex, _Outptr_ PBYTE pbBuf, _In_ DWORD dwBufSize, _In_ WDU_LANGID langID, _Outptr_ PDWORD pdwDescSize)

Reads a string descriptor from a device by string index.
- DWORD **DLLCALLCONV WDU_StreamOpen** (_In_ WDU_DEVICE_HANDLE hDevice, _In_ DWORD dwPipeNum, _In_ DWORD dwBufferSize, _In_ DWORD dwRxSize, _In_ BOOL fBlocking, _In_ DWORD dwOptions, _In_ DWORD dwRxTxTimeout, _Outptr_ WDU_STREAM_HANDLE *phStream)

Opens a new data stream for the specified pipe.
- DWORD **DLLCALLCONV WDU_StreamClose** (_In_ WDU_STREAM_HANDLE hStream)

Closes an open stream.
- DWORD **DLLCALLCONV WDU_StreamStart** (_In_ WDU_STREAM_HANDLE hStream)

Starts a stream, i.e starts transfers between the stream and the device.
- DWORD **DLLCALLCONV WDU_StreamStop** (_In_ WDU_STREAM_HANDLE hStream)

Stops a stream, i.e stops transfers between the stream and the device.
- DWORD **DLLCALLCONV WDU_StreamFlush** (_In_ WDU_STREAM_HANDLE hStream)

Flushes a stream, i.e writes the entire contents of the stream's data buffer to the device (relevant for a write stream), and blocks until the completion of all pending I/O on the stream.
- DWORD **DLLCALLCONV WDU_StreamRead** (_In_ HANDLE hStream, _Outptr_ PVOID pBuffer, _In_ DWORD bytes, _Outptr_ DWORD *pdwBytesRead)

Reads data from a read stream to the application.
- DWORD **DLLCALLCONV WDU_StreamWrite** (_In_ HANDLE hStream, _In_ const PVOID pBuffer, _In_ DWORD bytes, _Outptr_ DWORD *pdwBytesWritten)

Writes data from the application to a write stream.
- DWORD **DLLCALLCONV WDU_StreamGetStatus** (_In_ WDU_STREAM_HANDLE hStream, _Outptr_ BOOL *pfIsRunning, _Outptr_ DWORD *pdwLastError, _Outptr_ DWORD *pdwBytesInBuffer)

Returns a stream's current status.

Typedef Documentation

WDU_ATTACH_CALLBACK

```
typedef BOOL(DLLCALLCONV * WDU_ATTACH_CALLBACK) (_In_ WDU_DEVICE_HANDLE hDevice, _In_ WDU_DEVICE *pDeviceInfo, _In_ PVOID pUserData)
```

WinDriver calls this function with any new device that is attached, matches the given criteria, and if WD_ACKNOWLEDGE was passed to [WDU_Init\(\)](#) in dwOptions - not controlled yet by another driver.

This callback is called once for each matching interface.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface
in	<i>pDeviceInfo</i>	Pointer to device configuration details. This pointer is valid until the end of the function.
in	<i>pUserData</i>	Pointer to user data that was passed to WDU_Init (in the event table).

Returns

If WD_ACKNOWLEDGE was passed to [Wdu_Init\(\)](#), the implementor should check & return if he wants to control the device.

Definition at line 35 of file [wdu.lib.h](#).

WDU_DETACH_CALLBACK

```
typedef void(DLLCALLCONV * WDU_DETACH_CALLBACK) (_In_ WDU_DEVICE_HANDLE hDevice, _In_ PVOID p←  
UserData)
```

WinDriver calls this function when a controlled device has been detached from the system.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>pUserData</i>	Pointer to user data that was passed to Wdu_Init

Returns

None.

Definition at line 45 of file [wdu.lib.h](#).

WDU_DEVICE_HANDLE

```
typedef PVOID WDU_DEVICE_HANDLE
```

Definition at line 13 of file [wdu.lib.h](#).

WDU_DRIVER_HANDLE

```
typedef PVOID WDU_DRIVER_HANDLE
```

Definition at line 12 of file [wdu.lib.h](#).

WDU_LANGID

```
typedef WORD WDU_LANGID
```

Definition at line 16 of file [wdu.lib.h](#).

WDU_POWER_CHANGE_CALLBACK

```
typedef BOOL(DLLCALLCONV * WDU_POWER_CHANGE_CALLBACK) (_In_ WDU_DEVICE_HANDLE hDevice, _In_  
DWORD dwPowerState, _In_ PVOID pUserData)
```

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwPowerState</i>	Number of the power state selected.
in	<i>pUserData</i>	Pointer to user data that was passed to Wdu_Init (in the event table).

Returns

TRUE/FALSE; Currently there is no significance to the return value.

Definition at line 56 of file [wdu_lib.h](#).

WDU_STREAM_HANDLE

`typedef PVOID WDU_STREAM_HANDLE`

Definition at line 14 of file [wdu_lib.h](#).

Function Documentation

WDU_GetDeviceAddr()

```
DWORD DLLCALLCONV WDU_GetDeviceAddr (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _Out_ DWORD * pAddress )
```

Gets USB address that the device uses.

The address number is written to the caller supplied pAddress.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
out	<i>pAddress</i>	Pointer to DWORD, in which the result will be returned.

Returns

WinDriver Error Code

Note

: This function is supported only on Windows.

```
DWORD dwAddress;
WDU_GetDeviceAddr(hDevice, &dwAddress);
```

WDU_GetDeviceInfo()

```
DWORD DLLCALLCONV WDU_GetDeviceInfo (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _Outptr_ WDU_DEVICE ** ppDeviceInfo )
```

Gets configuration information from the device including all the descriptors in a `WDU_DEVICE` struct.
The caller should free `*ppDeviceInfo` after using it by calling [WDU_PutDeviceInfo\(\)](#).

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
out	<i>ppDeviceInfo</i>	Pointer to a pointer to a buffer containing the device information.

Returns

WinDriver Error Code

```
DWORD dwStatus;
WDU_DEVICE *pDevice = NULL;
dwStatus = WDU_GetDeviceInfo(hDevice, &pDevice);
```

```

if (dwStatus)
{
    printf("WDU_GetDeviceInfo failed. error 0x%lx "
        "(\"%s\")\n", dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
printf("This device has %d configurations:\n",
    pDevice->Descriptor.bNumConfigurations);
/* For full code with usage of the WDU_DEVICE information struct,
   please refer to the USB sample code (samples/c/usb_diag) */
Exit:
if (pDevice)
    free(pDevice);

```

WDU_GetDeviceRegistryProperty()

```

DWORD DLLCALLTYPE WDU_GetDeviceRegistryProperty (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _Outptr_ PVOID pBuffer,
    _Inout_ PDWORD pdwSize,
    _In_ WD_DEVICE_REGISTRY_PROPERTY property )

```

Gets the specified registry property of a given device.

Parameters

in	<i>hDevice</i>	A unique identifier of the device/interface.
out	<i>pBuffer</i>	Pointer to a user allocated buffer to be filled with the requested registry property. The function will fill the buffer only if the buffer size, as indicated in the input value of the pdwSize parameter, is sufficient - i.e >= the property's size, as returned via pdwSize. pBuffer can be set to NULL when using the function only to retrieve the size of the registry property (see pdwSize).
in,out	<i>pdwSize</i>	As input, points to a value indicating the size of the user-supplied buffer (pBuffer); if pBuffer is set to NULL, the input value of this parameter is ignored. As output, points to a value indicating the required buffer size for storing the registry property.
in	<i>property</i>	The ID of the registry property to be retrieved - see the WD_DEVICE_REGISTRY_PROPERTY enumeration in windrvr.h . Note: String registry properties are in WCHAR format.

Returns

WinDriver Error Code.

Note

When the size of the provided user buffer (pBuffer) - *pdwSize (input) is not sufficient to hold the requested registry property, the function returns WD_INVALID_PARAMETER.

```

DWORD dwStatus = 0;
DWORD dwSize;
#ifdef WIN32
    WCHAR cProperty[256];
#else
    CHAR cProperty[256];
#endif
dwStatus = WDU_GetDeviceRegistryProperty(hDevice, cProperty, &dwSize,
    WdDevicePropertyManufacturer);
if (WD_STATUS_SUCCESS == dwStatus)
{
    printf("%-46s: ", propertyNames[i]);
    /* In windows, device property string is encoded as wchar_t(WCHAR) */
#ifdef WIN32
    printf("%ws\n", cProperty);
#else
    printf("%s\n", cProperty);
#endif
}

```

WDU_GetLangIDs()

```
DWORD DLLCALLCONV WDU_GetLangIDs (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _Outptr_ PBYTE pbNumSupportedLangIDs,
    _Outptr_ WDU_LANGID * pLangIDs,
    _In_ BYTE bNumLangIDs )
```

Reads a list of supported language IDs and/or the number of supported language IDs from a device.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
out	<i>pbNumSupportedLangIDs</i>	Pointer to the number of supported language IDs, to be filled by the function. Can be NULL if bNumLangIDs is not 0 and pLangIDs is not NULL. If NULL, the function will not return the number of supported language IDs for the device.
out	<i>pLangIDs</i>	Array of language IDs. If bNumLangIDs is not 0 the function will fill this array with the supported language IDs for the device. If bNumLangIDs < the number of supported language IDs for the device, only the first bNumLangIDs supported language IDs will be read from the device and returned in the pLangIDs array.
in	<i>bNumLangIDs</i>	Number of IDs in pLangIDs array. If 0, the function will only return the number of supported language IDs.

Returns

WinDriver Error Code

Note

If no language IDs are supported for the device (*pbNumSupportedLangIDs == 0) the function returns WD_STATUS_SUCCESS.

WDUGetStringDesc()

```
DWORD DLLCALLCONV WDUGetStringDesc (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ BYTE bStrIndex,
    _Outptr_ PBYTE pbBuf,
    _In_ DWORD dwBufSize,
    _In_ WDU_LANGID langID,
    _Outptr_ PDWORD pdwDescSize )
```

Reads a string descriptor from a device by string index.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>bStrIndex</i>	Index of the string descriptor to read.
out	<i>pbBuf</i>	Pointer to a buffer to be filled with the string descriptor that is read from the device. If the buffer is smaller than the string descriptor (dwBufSize < *pdwDescSize), the returned descriptor will be truncated to dwBufSize bytes.
in	<i>dwBufSize</i>	The size of the pbBuf buffer, in bytes.
in	<i>langID</i>	The language ID to be used in the get string descriptor request that is sent to the device. If langID is 0, the request will use the first supported language ID returned by the device.
out	<i>pdwDescSize</i>	An optional DWORD pointer to be filled with the size of the string descriptor read from the device. If this parameter is NULL, the funWDU_StreamReadction will not return the size of the string descriptor.

Returns

WinDriver Error Code

```
WDU_DEVICE *pDevice;
DWORD dwStatus;
BYTE bSerialNum[0x100];
DWORD dwSerialDescSize = 0;
dwStatus = WDU_GetDeviceInfo(hDevice, &pDevice);
if (dwStatus)
{
    return;
}
if (!pDevice->Descriptor.iSerialNumber)
{
    printf("Serial number is not available\n");
    goto Exit;
}
WDUGetStringDesc(hDevice, pDevice->Descriptor.iSerialNumber,
    bSerialNum, sizeof(bSerialNum), 0, &dwSerialDescSize);
Exit:
if (pDevice)
    free(pDevice);
```

WDU_HaltTransfer()

```
DWORD DLLCALLCONV WDU_HaltTransfer (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum )
```

Halts the transfer on the specified pipe (only one simultaneous transfer per-pipe is allowed by WinDriver).

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwPipeNum</i>	Pipe number.

Returns

WinDriver Error Code

```
WDU_HaltTransfer(hDevice, 0);
```

WDU_Init()

```
DWORD DLLCALLCONV WDU_Init (
    _Outptr_ WDU_DRIVER_HANDLE * phDriver,
    _In_ WDU_MATCH_TABLE * pMatchTables,
    _In_ DWORD dwNumMatchTables,
    _In_ WDU_EVENT_TABLE * pEventTable,
    _In_ const char * pcLicense,
    _In_ DWORD dwOptions )
```

Starts listening to devices matching a criteria, and registers notification callbacks for those devices.

Parameters

out	<i>phDriver</i>	Handle to this registration of events & criteria.
in	<i>pMatchTables</i>	Array of match tables defining the devices-criteria.
in	<i>dwNumMatchTables</i>	Number of elements in pMatchTables.
in	<i>pEventTable</i>	Notification callbacks when the device's status changes.
in	<i>pcLicense</i>	WinDriver's license string.
in	<i>dwOptions</i>	Can be 0 or: WD_ACKNOWLEDGE - The user can seize control over the device in WDU_ATTACH_CALLBACK return value.

Returns

WinDriver Error Code

WDU_PutDeviceInfo()

```
void DLLCALLCONV WDU_PutDeviceInfo (
    _In_ WDU_DEVICE * pDeviceInfo )
```

Receives a device information pointer, allocated with a previous [WDU_GetDeviceInfo\(\)](#) call, in order to perform the necessary cleanup.

Parameters

in	<i>pDeviceInfo</i>	Pointer to a buffer containing the device information, as returned by a previous call to WDU_GetDeviceInfo() .
----	--------------------	--

Returns

None

WDU_ResetDevice()

```
DWORD DLLCALLCONV WDU_ResetDevice (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwOptions )
```

Resets a device (supported only on Windows).

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface
in	<i>dwOptions</i>	Can be 0 or: WD_USB_HARD_RESET - will reset the device even if it is not disabled. After using this option it is advised to set the interface of the device (WDU_SetInterface()). WD_USB_CYCLE_PORT - will simulate unplugging and replugging the device, prompting the operating system to enumerate the device without resetting it. This option is available only on Windows.

Returns

WinDriver Error Code The CYCLE_PORT option is supported only on Windows

WDU_ResetPipe()

```
DWORD DLLCALLCONV WDU_ResetPipe (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum )
```

Resets a pipe.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwPipeNum</i>	Pipe number.

Returns

WinDriver Error Code

```
WDU_ResetPipe(hDevice, dwPipeNum);
```

WDU_SelectiveSuspend()

```
DWORD DLLCALLCONV WDU_SelectiveSuspend (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwOptions )
```

Submits a request to suspend a given device (selective suspend), or cancels a previous suspend request.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwOptions</i>	Can be set to either of the following WDU_SELECTIVE_SUSPEND_OPTIONS enumeration values: WDU_SELECTIVE_SUSPEND_SUBMIT - submit a request to suspend the device. WDU_SELECTIVE_SUSPEND_CANCEL - cancel a previous suspend request for the device.

Returns

WinDriver Error Code. If a suspend request is received while the device is busy, the function returns WD_OPERATION_FAILED.

Note

This function is supported only on Windows.

```
WDU_SelectiveSuspend(hDevice, WDU_SELECTIVE_SUSPEND_SUBMIT);
```

WDU_SetConfig()

```
DWORD DLLCALLCONV WDU_SetConfig (
    WDU_DEVICE_HANDLE hDevice,
    DWORD dwConfigNum )
```

WDU_SetInterface()

```
DWORD DLLCALLCONV WDU_SetInterface (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwInterfaceNum,
    _In_ DWORD dwAlternateSetting )
```

Sets the alternate setting for the specified interface.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwInterfaceNum</i>	The interface's number.
in	<i>dwAlternateSetting</i>	The desired alternate setting value.

Returns

WinDriver Error Code

```
WDU_SetInterface(hDevice, dwInterfaceNumber, dwAlternateSetting);
```

WDU_StreamClose()

```
DWORD DLLCALLCONV WDU_StreamClose (
    _In_ WDU_STREAM_HANDLE hStream )
```

Closes an open stream.

The function stops the stream, including flushing its data to the device (in the case of a write stream), before closing it.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
----	----------------	---

Returns

WinDriver Error Code

```
DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
                           dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
        "error 0x%lx (%"s"\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (%"s"\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
                               &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
                               &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;
    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
                                   &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);
```

WDU_StreamFlush()

```
DWORD DLLCALLCONV WDU_StreamFlush (
    _In_ WDU_STREAM_HANDLE hStream )
```

Flushes a stream, i.e writes the entire contents of the stream's data buffer to the device (relevant for a write stream), and blocks until the completion of all pending I/O on the stream.

This function can be called for both blocking and non-blocking streams.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
----	----------------	---

Returns

WinDriver Error Code

```
WDU_StreamFlush(stream);
```

WDU_StreamGetStatus()

```
DWORD DLLCALLTYPE WDU_StreamGetStatus (
    _In_ WDU_STREAM_HANDLE hStream,
    _Outptr_ BOOL * pfIsRunning,
    _Outptr_ DWORD * pdwLastError,
    _Outptr_ DWORD * pdwBytesInBuffer )
```

Returns a stream's current status.

This function can be called for both blocking and non-blocking streams.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
out	<i>pfIsRunning</i>	Pointer to a boolean value indicating the stream's current state: TRUE - the stream is currently running; FALSE - the stream is currently stopped.
out	<i>pdwLastError</i>	Pointer to the last error associated with the stream. Note: Calling this function also resets the stream's last error.
out	<i>pdwBytesInBuffer</i>	Pointer to the current bytes count in the stream's data buffer.

Returns

WinDriver Error Code

```
DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
                         dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
        "error 0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
                             &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
                             &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;
    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
                                   &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);
```

WDU_StreamOpen()

```
DWORD DLLCALLCONV WDU_StreamOpen (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum,
    _In_ DWORD dwBufferSize,
    _In_ DWORD dwRxSize,
    _In_ BOOL fBlocking,
    _In_ DWORD dwOptions,
    _In_ DWORD dwRxTxTimeout,
    _Outptr_ WDU_STREAM_HANDLE * phStream )
```

Opens a new data stream for the specified pipe.

Note

The streaming functions are currently supported only on Windows. A stream can be associated with any pipe except the control pipe (Pipe 0). The stream's data transfer direction – read/write – is derived from the direction of its pipe.

Parameters

in	hDevice	A unique identifier for the device/interface.
in	dwPipeNum	The number of the pipe for which to open the stream
in	dwBufferSize	The size, in bytes, of the stream's data buffer.
in	dwRxSize	The size, in bytes, of the data blocks that the stream reads from the device. This parameter is relevant only for read streams and must be <= dwBufferSize.
in	fBlocking	TRUE for a blocking stream (performs blocking I/O); FALSE for a non-blocking stream (non-blocking I/O).
in	dwOptions	Can be a bit-mask of any of the following options: USB_ISOCH_NOASAP - For isochronous data transfers. Setting this option instructs the lower driver (usbd.sys) to use a preset frame number (instead of the next available frame) while performing the data transfer. Use this flag if you notice unused frames during the transfer, on low-speed or full-speed devices (USB 1.1 only) and only on Windows. USB_ISOCH_RESET - Resets the isochronous pipe before the data transfer. It also resets the pipe after minor errors, consequently allowing the transfer to continue. USB_ISOCH_FULL_PACKETS_ONLY - When set, do not transfer less than packet size on isochronous pipes. USB_BULK_INT_URB_SIZE_OVERRIDE_128K - Limits the size of the USB Request Block (URB) to 128KB. USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL - When there is not enough free space in a read stream's data buffer to complete the transfer, overwrite old data in the buffer. (Applicable only to read streams).
in	dwRxTxTimeout	Maximum time, in milliseconds, for the completion of a data transfer between the stream and the device. Zero = infinite wait.
out	phStream	Pointer to a unique identifier for the stream, to be returned by the function and passed to the other WDU_StreamXXX() functions.

Returns**WinDriver Error Code**

```
DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
                           dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
```

```

        "error 0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;
    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
        &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);

```

WDU_StreamRead()

```

DWORD DLLCALLTYPE WDU_StreamRead (
    _In_ HANDLE hStream,
    _Outptr_ PVOID pBuffer,
    _In_ DWORD bytes,
    _Outptr_ DWORD * pdwBytesRead )

```

Reads data from a read stream to the application.

For a blocking stream (fBlocking=TRUE - see [WDU_StreamOpen\(\)](#)), the call to this function is blocked until the specified amount of data is read, or until the stream's attempt to read from the device times out (i.e the timeout period for transfers between the stream and the device, as set in the dwRxTxTimeout [WDU_StreamOpen\(\)](#) parameter, expires). For a non-blocking stream, the function transfers to the application as much of the requested data as possible, subject to the amount of data currently available in the stream's data buffer, and returns immediately. For both blocking and non-blocking transfers, the function returns the amount of bytes that were actually read from the stream within the pdwBytesRead parameter.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
out	<i>pBuffer</i>	Pointer to a data buffer to be filled with the data read from the stream.
in	<i>bytes</i>	Number of bytes to read from the stream.
out	<i>pdwBytesRead</i>	Pointer to a value indicating the number of bytes actually read from the stream.

Returns

WinDriver Error Code

```

DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
    dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
        "error 0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}

```

```

}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;
    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
        &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);

```

WDU_StreamStart()

DWORD **DLLCALLCONV** WDU_StreamStart (
 In WDU_STREAM_HANDLE hStream)

Starts a stream, i.e starts transfers between the stream and the device.

Data will be transferred according to the stream's direction - read/write.

Parameters

in	hStream	A unique identifier for the stream, as returned by WDU_StreamOpen() .
-----------	----------------	---

Returns

WinDriver Error Code

```

DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
    dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
        "error 0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;

```

```

    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
        &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);

```

WDU_StreamStop()

```

DWORD DLLCALLCONV WDU_StreamStop (
    _In_ WDU_STREAM_HANDLE hStream )

```

Stops a stream, i.e stops transfers between the stream and the device.

In the case of a write stream, the function flushes the stream - i.e writes its contents to the device - before stopping it.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
----	----------------	---

Returns

WinDriver Error Code

WDU_StreamWrite()

```

DWORD DLLCALLCONV WDU_StreamWrite (
    _In_ HANDLE hStream,
    _In_ const PVOID pBuffer,
    _In_ DWORD bytes,
    _Outptr_ DWORD * pdwBytesWritten )

```

Writes data from the application to a write stream.

For a blocking stream (fBlocking=TRUE - see [WDU_StreamOpen\(\)](#)), the call to this function is blocked until the entire data (*pBuffer) is written to the stream's data buffer, or until the stream's attempt to write to the device times out (i.e the timeout period for transfers between the stream and the device, as set in the dwRxTxTimeout [WDU_StreamOpen\(\)](#) parameter, expires). For a non-blocking stream (fBlocking=FALSE), the function writes as much of the write data as currently possible to the stream's data buffer, and returns immediately. For both blocking and non-blocking transfers, the function returns the amount of bytes that were actually written to the stream within the pdwBytesWritten parameter.

Parameters

in	<i>hStream</i>	A unique identifier for the stream, as returned by WDU_StreamOpen() .
in	<i>pBuffer</i>	Pointer to a data buffer containing the data to write to the stream.
in	<i>bytes</i>	Number of bytes to write to the stream.
out	<i>pdwBytesWritten</i>	Pointer to a value indicating the number of bytes actually written to the stream.

Returns

WinDriver Error Code

```

DWORD dwStatus;
DWORD dwPipeNum, dwSize, dwBytesTransferred, cmd = MENU_RW_READ_PIPE;
VOID *pBuffer = NULL;
WDU_STREAM_HANDLE stream;
DWORD dwBufferSize = 0x20000;
dwStatus = WDU_StreamOpen(hDevice, dwPipeNum, dwBufferSize,
    dwSize, TRUE, 0, TRANSFER_TIMEOUT, &stream);
if (dwStatus)

```

```

{
    ERR("ReadWritePipesMenu: WDU_StreamOpen() failed. "
        "error 0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
dwStatus = WDU_StreamStart(stream);
if (dwStatus)
{
    ERR("ReadWritePipesMenu: WDU_StreamStart() failed. error "
        "0x%lx (\\"%s\\")\n", dwStatus, Stat2Str(dwStatus));
    goto End_transfer;
}
if (cmd == MENU_RW_READ_PIPE)
{
    dwStatus = WDU_StreamRead(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
else
{
    dwStatus = WDU_StreamWrite(stream, pBuffer, dwSize,
        &dwBytesTransferred);
}
if (dwStatus)
{
    BOOL fIsRunning;
    DWORD dwLastError;
    DWORD dwBytesInBuffer;
    dwStatus = WDU_StreamGetStatus(stream, &fIsRunning,
        &dwLastError, &dwBytesInBuffer);
    if (!dwStatus)
        dwStatus = dwLastError;
}
WDU_StreamClose(stream);

```

WDU_Transfer()

```

DWORD DLLCALLCONV WDU_Transfer (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum,
    _In_ DWORD fRead,
    _In_ DWORD dwOptions,
    _In_ PVOID pBuffer,
    _In_ DWORD dwBufferSize,
    _Outptr_ PDWORD pdwBytesTransferred,
    _In_ PBYTE pSetupPacket,
    _In_ DWORD dwTimeout )

```

Transfers data to/from a device.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface.
in	<i>dwPipeNum</i>	The number of the pipe through which the data is transferred.
in	<i>fRead</i>	TRUE for read, FALSE for write.
in	<i>dwOptions</i>	Can be a bit-mask of any of the following options: USB_ISOCH_NOASAP - For isochronous data transfers. Setting this option instructs the lower driver (usbd.sys) to use a preset frame number (instead of the next available frame) while performing the data transfer. Use this flag if you notice unused frames during the transfer, on low-speed or full-speed devices (USB 1.1 only) and on Windows only. USB_ISOCH_RESET - Resets the isochronous pipe before the data transfer. It also resets the pipe after minor errors, consequently allowing the transfer to continue. USB_ISOCH_FULL_PACKETS_ONLY - When set, do not transfer less than packet size on isochronous pipes. USB_BULK_INT_URB_SIZE_OVERRIDE_128K - Limits the size of the USB Request Block (URB) to 128KB.
in	<i>pBuffer</i>	location of the data buffer
in	<i>dwBufferSize</i>	Number of the bytes to transfer.
out	<i>pdwBytesTransferred</i>	Number of bytes actually transferred.

Parameters

in	<i>pSetupPacket</i>	8-bytes packet to transfer to control pipes.
in	<i>dwTimeout</i>	Maximum time, in milliseconds, to complete a transfer. Zero = infinite wait.

Returns

WinDriver Error Code

```
DWORD dwSize = 0x100, dwPipeNum = 0;
DWORD dwBytesTransferred;
PVOID pBuffer = malloc(dwSize);
BYTE SetupPacket[8];
WDU_Transfer(hDevice, dwPipeNum, MENU_RW_READ_PIPE, 0, pBuffer, dwSize,
    &dwBytesTransferred, SetupPacket, TRANSFER_TIMEOUT);
```

WDU_TransferBulk()

```
DWORD DLLCALLTYPE WDU_TransferBulk (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum,
    _In_ DWORD fRead,
    _In_ DWORD dwOptions,
    _In_ PVOID pBuffer,
    _In_ DWORD dwBufferSize,
    _Outptr_ PDWORD pdwBytesTransferred,
    _In_ DWORD dwTimeout )
```

WDU_TransferDefaultPipe()

```
DWORD DLLCALLTYPE WDU_TransferDefaultPipe (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD fRead,
    _In_ DWORD dwOptions,
    _In_ PVOID pBuffer,
    _In_ DWORD dwBufferSize,
    _Outptr_ PDWORD pdwBytesTransferred,
    _In_ PBYTE pSetupPacket,
    _In_ DWORD dwTimeout )
```

WDU_TransferInterrupt()

```
DWORD DLLCALLTYPE WDU_TransferInterrupt (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum,
    _In_ DWORD fRead,
    _In_ DWORD dwOptions,
    _In_ PVOID pBuffer,
    _In_ DWORD dwBufferSize,
    _Outptr_ PDWORD pdwBytesTransferred,
    _In_ DWORD dwTimeout )
```

WDU_TransferIsoch()

```
DWORD DLLCALLTYPE WDU_TransferIsoch (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwPipeNum,
```

```

    _In_ DWORD fRead,
    _In_ DWORD dwOptions,
    _In_ PVOID pBuffer,
    _In_ DWORD dwBufferSize,
    _Outptr_ PDWORD pdwBytesTransferred,
    _In_ DWORD dwTimeout )

```

WDU_Uninit()

```
void DLLCALLCONV WDU_Uninit (
    _In_ WDU_DRIVER_HANDLE hDriver )
```

Stops listening to devices matching the criteria, and unregisters the notification callbacks for those devices.

Parameters

in	<i>hDriver</i>	Handle to the registration, received from WDU_Init.
----	----------------	---

Returns

None

WDU_Wakeup()

```
DWORD DLLCALLCONV WDU_Wakeup (
    _In_ WDU_DEVICE_HANDLE hDevice,
    _In_ DWORD dwOptions )
```

Enables/Disables wakeup feature.

Parameters

in	<i>hDevice</i>	A unique identifier for the device/interface
in	<i>dwOptions</i>	Can be set to either of the following options: WDU_WAKEUP_ENABLE - will enable wakeup. or: WDU_WAKEUP_DISABLE - will disable wakeup.

Returns

WinDriver Error Code

Note

This function is supported only on Windows.

wdu_lib.h

[Go to the documentation of this file.](#)

```

00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WD_USB_H_
00004 #define _WD_USB_H_
00005
00006 #include "windrvr.h"
00007
00008 #if defined(__cplusplus)
00009     extern "C" {
00010 #endif
00011
00012 typedef PVOID WDU_DRIVER_HANDLE;
00013 typedef PVOID WDU_DEVICE_HANDLE;
00014 typedef PVOID WDU_STREAM_HANDLE;
00015

```

```
00016 typedef WORD WDU_LANGID;
00017
00018 /*
00019 * User Callbacks
00020 */
00021
00022 #define _IN_WDU_DEVICE _In_ WDU_DEVICE
00023 #define _IN_WDU_EVENT_TABLE _In_ WDU_EVENT_TABLE
00024
00025 #define _IN_WDU_DRIVER_HANDLE _In_ WDU_DRIVER_HANDLE
00026
00027 #define _IN_WDU_MATCH_TABLE _In_ WDU_MATCH_TABLE
00028
00029 #define _IN_WDU_POWER_CHANGE_CALLBACK _In_ WDU_POWER_CHANGE_CALLBACK
00030
00031 #define _IN_WDU_TRANSFER_CALLBACK _In_ WDU_TRANSFER_CALLBACK
00032
00033 #define _IN_WDU_WAKEUP_CALLBACK _In_ WDU_WAKEUP_CALLBACK
00034
00035 #define _IN_WDU_ATTACH_CALLBACK _In_ WDU_ATTACH_CALLBACK
00036
00037 #define _IN_WDU_DETACH_CALLBACK _In_ WDU_DETACH_CALLBACK
00038
00039 #define _IN_WDU_POWER_CHANGE_CALLBACK _In_ WDU_POWER_CHANGE_CALLBACK
00040
00041 #define _IN_WDU_RESETPIPE_CALLBACK _In_ WDU_RESETPIPE_CALLBACK
00042
00043 #define _IN_WDU_SETINTERFACE_CALLBACK _In_ WDU_SETINTERFACE_CALLBACK
00044
00045 #define _IN_WDU_RESETDEVICE_CALLBACK _In_ WDU_RESETDEVICE_CALLBACK
00046
00047 #define _IN_WDU_HALTTRANSFER_CALLBACK _In_ WDU_HALTTRANSFER_CALLBACK
00048
00049 #define _IN_WDU_SELECTIVESUSPEND_CALLBACK _In_ WDU_SELECTIVESUSPEND_CALLBACK
00050
00051 #define _IN_WDU_TRANSFER_CALLBACK _In_ WDU_TRANSFER_CALLBACK
00052
00053 #define _IN_WDU_TRANSFERDEFAULTPIPE_CALLBACK _In_ WDU_TRANSFERDEFAULTPIPE_CALLBACK
00054
00055 #define _IN_WDU_PUTDEVICEINFO_CALLBACK _In_ WDU_PUTDEVICEINFO_CALLBACK
00056
00057 #define _IN_WDU_SETCONFIG_CALLBACK _In_ WDU_SETCONFIG_CALLBACK
00058
00059 #define _IN_WDU_WAKEUP_CALLBACK _In_ WDU_WAKEUP_CALLBACK
00060
00061 /*
00062 * struct definitions
00063 */
00064
00065 struct WDU_EVENT_TABLE {
00066     WDU_ATTACH_CALLBACK pfDeviceAttach;
00067     WDU_DETACH_CALLBACK pfDeviceDetach;
00068     WDU_POWER_CHANGE_CALLBACK pfPowerChange;
00069     PVOID pUserData; /* pointer to pass in each callback */
00070 };
00071
00072 /*
00073 * API Functions
00074 */
00075
00076 DWORD DLLCALLCONV WDU_Init(_Outptr_ WDU_DRIVER_HANDLE *phDriver,
00077     _In_ WDU_MATCH_TABLE *pMatchTables, _In_ DWORD dwNumMatchTables,
00078     _In_ WDU_EVENT_TABLE *pEventTable, _In_ const char *pcLicense,
00079     _In_ DWORD dwOptions);
00080
00081 void DLLCALLCONV WDU_Uninit(_In_ WDU_DRIVER_HANDLE hDriver);
00082
00083 DWORD DLLCALLCONV WDU_GetDeviceAddr(_In_ WDU_DEVICE_HANDLE hDevice,
00084     _Out_ DWORD *pAddress);
00085
00086 DWORD DLLCALLCONV WDU_GetDeviceRegistryProperty(_In_ WDU_DEVICE_HANDLE hDevice,
00087     _Outptr_ PVOID pBuffer, _Inout_ PDWORD pdwSize,
00088     _In_ WD_DEVICE_REGISTRY_PROPERTY property);
00089
00090 DWORD DLLCALLCONV WDU_GetDeviceInfo(_In_ WDU_DEVICE_HANDLE hDevice,
00091     _Outptr_ WDU_DEVICE **ppDeviceInfo);
00092
00093 void DLLCALLCONV WDU_PutDeviceInfo(_In_ WDU_DEVICE *pDeviceInfo);
00094
00095 DWORD DLLCALLCONV WDU_SetInterface(_In_ WDU_DEVICE_HANDLE hDevice,
00096     _In_ DWORD dwInterfaceNum, _In_ DWORD dwAlternateSetting);
00097
00098 /*
00099 * NOT IMPLEMENTED YET */
00100 DWORD DLLCALLCONV WDU_SetConfig(WDU_DEVICE_HANDLE hDevice, DWORD dwConfigNum);
00101
00102 /*
00103 * NOT IMPLEMENTED YET */
00104
00105 DWORD DLLCALLCONV WDU_ResetPipe(_In_ WDU_DEVICE_HANDLE hDevice,
00106     _In_ DWORD dwPipeNum);
00107
00108 DWORD DLLCALLCONV WDU_ResetDevice(_In_ WDU_DEVICE_HANDLE hDevice,
00109     _In_ DWORD dwOptions);
00110
00111 DWORD DLLCALLCONV WDU_Wakeup(_In_ WDU_DEVICE_HANDLE hDevice,
00112     _In_ DWORD dwOptions);
00113
00114 DWORD DLLCALLCONV WDU_SelectiveSuspend(_In_ WDU_DEVICE_HANDLE hDevice,
00115     _In_ DWORD dwOptions);
00116
00117 DWORD DLLCALLCONV WDU_Transfer(_In_ WDU_DEVICE_HANDLE hDevice,
00118     _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions,
00119     _In_ PVOID pBuffer, _In_ DWORD dwBufferSize,
00120     _Outptr_ PDWORD pdwBytesTransferred, _In_ PBYTE pSetupPacket,
00121     _In_ DWORD dwTimeout);
00122
00123 DWORD DLLCALLCONV WDU_HaltTransfer(_In_ WDU_DEVICE_HANDLE hDevice,
00124     _In_ DWORD dwPipeNum);
00125
00126 /*
00127 * Simplified transfers - for a specific pipe
00128 */
00129
00130 DWORD DLLCALLCONV WDU_TransferDefaultPipe(_In_ WDU_DEVICE_HANDLE hDevice,
00131     _In_ DWORD fRead, _In_ DWORD dwOptions,
00132     _In_ PVOID pBuffer, _In_ DWORD dwBufferSize,
00133     _Outptr_ PDWORD pdwBytesTransferred,
00134     _In_ PBYTE pSetupPacket, _In_ DWORD dwTimeout);
```

```

00298
00299 DWORD DLLCALLCONV WDU_TransferBulk(_In_ WDU_DEVICE_HANDLE hDevice,
00300     _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions,
00301     _In_ PVOID pBuffer, _In_ DWORD dwBufferSize,
00302     _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout);
00303
00304 DWORD DLLCALLCONV WDU_TransferIsoch(_In_ WDU_DEVICE_HANDLE hDevice,
00305     _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions,
00306     _In_ PVOID pBuffer, _In_ DWORD dwBufferSize,
00307     _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout);
00308
00309 DWORD DLLCALLCONV WDU_TransferInterrupt(_In_ WDU_DEVICE_HANDLE hDevice,
00310     _In_ DWORD dwPipeNum, _In_ DWORD fRead, _In_ DWORD dwOptions,
00311     _In_ PVOID pBuffer, _In_ DWORD dwBufferSize,
00312     _Outptr_ PDWORD pdwBytesTransferred, _In_ DWORD dwTimeout);
00313
00314 DWORD DLLCALLCONV WDU_GetLangIDs(_In_ WDU_DEVICE_HANDLE hDevice,
00315     _Outptr_ PBYTE pbNumSupportedLangIDs, _Outptr_ WDU_LANGID *pLangIDs,
00316     _In_ BYTE bNumLangIDs);
00317
00318 DWORD DLLCALLCONV WDUGetStringDesc(_In_ WDU_DEVICE_HANDLE hDevice,
00319     _In_ BYTE bStrIndex, _Outptr_ PBYTE pbBuf, _In_ DWORD dwBufSize,
00320     _In_ WDU_LANGID langID, _Outptr_ PDWORD pdwDescSize);
00321
00322 /*
00323  * Streaming Functions
00324 */
00325
00326 DWORD DLLCALLCONV WDU_StreamOpen(_In_ WDU_DEVICE_HANDLE hDevice,
00327     _In_ DWORD dwPipeNum, _In_ DWORD dwBufferSize, _In_ DWORD dwRxSize,
00328     _In_ BOOL fBlocking, _In_ DWORD dwOptions, _In_ DWORD dwRxTxTimeout,
00329     _Outptr_ WDU_STREAM_HANDLE *phStream);
00330
00331 DWORD DLLCALLCONV WDU_StreamClose(_In_ WDU_STREAM_HANDLE hStream);
00332
00333 DWORD DLLCALLCONV WDU_StreamStart(_In_ WDU_STREAM_HANDLE hStream);
00334
00335 DWORD DLLCALLCONV WDU_StreamStop(_In_ WDU_STREAM_HANDLE hStream);
00336
00337 DWORD DLLCALLCONV WDU_StreamFlush(_In_ WDU_STREAM_HANDLE hStream);
00338
00339 DWORD DLLCALLCONV WDU_StreamRead(_In_ HANDLE hStream, _Outptr_ PVOID pBuffer,
00340     _In_ DWORD bytes, _Outptr_ DWORD *pdwBytesRead);
00341
00342 DWORD DLLCALLCONV WDU_StreamWrite(_In_ HANDLE hStream,
00343     _In_ const PVOID pBuffer, _In_ DWORD bytes,
00344     _Outptr_ DWORD *pdwBytesWritten);
00345
00346 DWORD DLLCALLCONV WDU_StreamGetStatus(_In_ WDU_STREAM_HANDLE hStream,
00347     _Outptr_ BOOL *pfIsRunning, _Outptr_ DWORD *pdwLastError,
00348     _Outptr_ DWORD *pdwBytesInBuffer);
00349
00350 #ifdef __cplusplus
00351 }
00352 #endif
00353
00354 #endif /* _WD_USB_H_ */
00355

```

windrivr.h File Reference

```
#include "wd_ver.h"
#include <windows.h>
#include <winiocrtl.h>
#include <stdarg.h>
#include "windrvr_usb.h"
```

Data Structures

- struct [WD_DMA_PAGE](#)
- struct [WD_DMA](#)
- struct [WD_KERNEL_BUFFER](#)
- struct [WD_TRANSFER](#)
- struct [WD_KERNEL_PLUGIN_CALL](#)
- struct [WD_INTERRUPT](#)

- struct [WD_VERSION](#)
- struct [WD_LICENSE](#)
- struct [WD_BUS](#)
- struct [WD_ITEMS](#)
- struct [WD_CARD](#)
- struct [WD_CARD_REGISTER](#)
- struct [WD_IPC_PROCESS](#)
- struct [WD_IPC_REGISTER](#)
- struct [WD_IPC_SCAN_PROCS](#)
- struct [WD_IPC_SEND](#)
- struct [WD_CARD_CLEANUP](#)
- struct [WD_PCI_SLOT](#)
- struct [WD_PCI_ID](#)
- struct [WD_PCI_SCAN_CARDS](#)
- struct [WD_PCI_CAP](#)
- struct [WD_PCI_SCAN_CAPS](#)
- struct [WD_PCI_SRIOV](#)
- struct [WD_PCI_CARD_INFO](#)
- struct [WD_PCI_CONFIG_DUMP](#)
- struct [WD_SLEEP](#)
- struct [WD_DEBUG](#)
- struct [WD_DEBUG_DUMP](#)
- struct [WD_DEBUG_ADD](#)
- struct [WD_KERNEL_PLUGIN](#)
- struct [WD_GET_DEVICE_PROPERTY](#)
- struct [WD_EVENT](#)
- struct [WD_USAGE](#)
- struct [WD_OS_INFO](#)

Macros

- #define DLLCALLCONV
- #define _In_
- #define _Inout_
- #define _Out_
- #define _Outptr_
- #define WD_DRIVER_NAME_PREFIX ""
- #define __FUNCTION__ func
- #define WD_DEFAULT_DRIVER_NAME_BASE "windrvr" WD_VER_Itoa
- #define WD_DEFAULT_DRIVER_NAME WD_DRIVER_NAME_PREFIX WD_DEFAULT_DRIVER_NAME_BASE
- #define WD_MAX_DRIVER_NAME_LENGTH 128
- #define WD_MAX_KP_NAME_LENGTH 128
- #define WD_VERSION_STR_LENGTH 128
- #define WD_DRIVER_NAME WD_DriverName(NULL)
Get driver name.
- #define WD_PROD_NAME "WinDriver"
- #define WD_CPU_SPEC "X86"
- #define WD_DATA_MODEL "32bit"
- #define WD_VER_STR
- #define WIN32
- #define stricmp _stricmp
- #define va_copy(ap2, ap1) (ap2)=(ap1)
- #define WINAPI
- #define PRI64 "I64"

formatting for printing a 64bit variable

- `#define KPRI ""`
formatting for printing a kernel pointer
- `#define UPRI "I"`
- `#define DMA_BIT_MASK(n) (((n) == 64) ? ~0ULL : ((1ULL<<(n))-1))`
- `#define DMA_ADDRESS_WIDTH_MASK 0x7f000000`
- `#define DMA_OPTIONS_ALL`
- `#define DMA_DIRECTION_MASK DMA_TO_FROM_DEVICE`
- `#define DMA_READ_FROM_DEVICE DMA_FROM_DEVICE`

Macros for backward compatibility.

- `#define DMA_WRITE_TO_DEVICE DMA_TO_DEVICE`
- `#define DMA_OPTIONS_ADDRESS_WIDTH_SHIFT`
- `#define PAD_TO_64(pName)`
- `#define WD_LICENSE_LENGTH 3072`
- `#define WD_PROCESS_NAME_LENGTH 128`
- `#define DEBUG_USER_BUF_LEN 2048`
- `#define WD_IPC_ALL_MSG (WD_IPC_UNICAST_MSG | WD_IPC_MULTICAST_MSG)`
- `#define WD_ACTIONS_POWER`
- `#define WD_ACTIONS_ALL (WD_ACTIONS_POWER | WD_INSERT | WD_REMOVE)`
- `#define BZERO(buf) memset(&(buf), 0, sizeof(buf))`
- `#define INVALID_HANDLE_VALUE ((HANDLE)(-1))`
- `#define CTL_CODE(DeviceType, Function, Method, Access)`
- `#define METHOD_BUFFERED 0`
- `#define METHOD_IN_DIRECT 1`
- `#define METHOD_OUT_DIRECT 2`
- `#define METHOD_NEITHER 3`
- `#define FILE_ANY_ACCESS 0`
- `#define FILE_READ_ACCESS 1`

file & pipe

- `#define FILE_WRITE_ACCESS 2`

file & pipe

- `#define WD_TYPE 38200`

Device type.

- `#define FUNC_MASK 0x0`
- `#define WD_CTL_CODE(wFuncNum)`
- `#define WD_CTL_DECODE_FUNC(IoControlCode) ((IoControlCode >> 2) & 0xffff)`
- `#define WD_CTL_DECODE_TYPE(IoControlCode) DEVICE_TYPE_FROM_CTL_CODE(IoControlCode)`
- `#define WD_CTL_IS_64BIT_AWARE(IoControlCode) (WD_CTL_DECODE_FUNC(IoControlCode) & FUNC_MASK)`
- `#define IOCTL_WD_KERNEL_BUF_LOCK WD_CTL_CODE(0x9f3)`
- `#define IOCTL_WD_KERNEL_BUF_UNLOCK WD_CTL_CODE(0x9f4)`
- `#define IOCTL_WD_DMA_LOCK WD_CTL_CODE(0x9be)`
- `#define IOCTL_WD_DMA_UNLOCK WD_CTL_CODE(0x902)`
- `#define IOCTL_WD_TRANSFER WD_CTL_CODE(0x98c)`
- `#define IOCTL_WD_MULTI_TRANSFER WD_CTL_CODE(0x98d)`
- `#define IOCTL_WD_PCI_SCAN_CARDS WD_CTL_CODE(0x9fa)`
- `#define IOCTL_WD_PCI_GET_CARD_INFO WD_CTL_CODE(0x9e8)`
- `#define IOCTL_WD_VERSION WD_CTL_CODE(0x910)`
- `#define IOCTL_WD_PCI_CONFIG_DUMP WD_CTL_CODE(0x91a)`
- `#define IOCTL_WD_KERNEL_PLUGIN_OPEN WD_CTL_CODE(0x91b)`
- `#define IOCTL_WD_KERNEL_PLUGIN_CLOSE WD_CTL_CODE(0x91c)`
- `#define IOCTL_WD_KERNEL_PLUGIN_CALL WD_CTL_CODE(0x91d)`
- `#define IOCTL_WD_INT_ENABLE WD_CTL_CODE(0x9b6)`
- `#define IOCTL_WD_INT_DISABLE WD_CTL_CODE(0x9bb)`

- #define IOCTL_WD_INT_COUNT WD_CTL_CODE(0x9ba)
- #define IOCTL_WD_SLEEP WD_CTL_CODE(0x927)
- #define IOCTL_WD_DEBUG WD_CTL_CODE(0x928)
- #define IOCTL_WD_DEBUG_DUMP WD_CTL_CODE(0x929)
- #define IOCTL_WD_CARD_UNREGISTER WD_CTL_CODE(0x9e7)
- #define IOCTL_WD_CARD_REGISTER WD_CTL_CODE(0x9e6)
- #define IOCTL_WD_INT_WAIT WD_CTL_CODE(0x9b9)
- #define IOCTL_WD_LICENSE WD_CTL_CODE(0x9f9)
- #define IOCTL_WD_EVENT_REGISTER WD_CTL_CODE(0x9ef)
- #define IOCTL_WD_EVENT_UNREGISTER WD_CTL_CODE(0x9f0)
- #define IOCTL_WD_EVENT_PULL WD_CTL_CODE(0x9f1)
- #define IOCTL_WD_EVENT_SEND WD_CTL_CODE(0x9f2)
- #define IOCTL_WD_DEBUG_ADD WD_CTL_CODE(0x964)
- #define IOCTL_WD_USAGE WD_CTL_CODE(0x976)
- #define IOCTL_WDU_GET_DEVICE_DATA WD_CTL_CODE(0x9a7)
- #define IOCTL_WDU_SET_INTERFACE WD_CTL_CODE(0x981)
- #define IOCTL_WDU_RESET_PIPE WD_CTL_CODE(0x982)
- #define IOCTL_WDU_TRANSFER WD_CTL_CODE(0x983)
- #define IOCTL_WDU_HALT_TRANSFER WD_CTL_CODE(0x985)
- #define IOCTL_WDU_WAKEUP WD_CTL_CODE(0x98a)
- #define IOCTL_WDU_RESET_DEVICE WD_CTL_CODE(0x98b)
- #define IOCTL_WD_GET_DEVICE_PROPERTY WD_CTL_CODE(0x990)
- #define IOCTL_WD_CARD_CLEANUP_SETUP WD_CTL_CODE(0x995)
- #define IOCTL_WD_DMA_SYNC_CPU WD_CTL_CODE(0x99f)
- #define IOCTL_WD_DMA_SYNC_IO WD_CTL_CODE(0x9a0)
- #define IOCTL_WDU_STREAM_OPEN WD_CTL_CODE(0x9a8)
- #define IOCTL_WDU_STREAM_CLOSE WD_CTL_CODE(0x9a9)
- #define IOCTL_WDU_STREAM_START WD_CTL_CODE(0x9af)
- #define IOCTL_WDU_STREAM_STOP WD_CTL_CODE(0x9b0)
- #define IOCTL_WDU_STREAM_FLUSH WD_CTL_CODE(0x9aa)
- #define IOCTL_WDU_STREAM_GET_STATUS WD_CTL_CODE(0x9b5)
- #define IOCTL_WDU_SELECTIVE_SUSPEND WD_CTL_CODE(0x9ae)
- #define IOCTL_WD_PCI_SCAN_CAPS WD_CTL_CODE(0x9e5)
- #define IOCTL_WD_IPC_REGISTER WD_CTL_CODE(0x9eb)
- #define IOCTL_WD_IPC_UNREGISTER WD_CTL_CODE(0x9ec)
- #define IOCTL_WD_IPC_SCAN_PROCS WD_CTL_CODE(0x9ed)
- #define IOCTL_WD_IPC_SEND WD_CTL_CODE(0x9ee)
- #define IOCTL_WD_PCI_SRIOV_ENABLE WD_CTL_CODE(0x9f5)
- #define IOCTL_WD_PCI_SRIOV_DISABLE WD_CTL_CODE(0x9f6)
- #define IOCTL_WD_PCI_SRIOV_GET_NUMVFS WD_CTL_CODE(0x9f7)
- #define IOCTL_WD_IPC_SHARED_INT_ENABLE WD_CTL_CODE(0x9fc)
- #define IOCTL_WD_IPC_SHARED_INT_DISABLE WD_CTL_CODE(0x9fd)
- #define IOCTL_WD_DMA_TRANSACTION_INIT WD_CTL_CODE(0x9fe)
- #define IOCTL_WD_DMA_TRANSACTION_EXECUTE WD_CTL_CODE(0x9ff)
- #define IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK WD_CTL_CODE(0xa00)
- #define IOCTL_WD_DMA_TRANSACTION_RELEASE WD_CTL_CODE(0xa01)
- #define REGKEY_BUFSIZE 256
- #define OS_CAN_NOT_DETECT_TEXT "OS CAN NOT DETECT"
- #define INSTALLATION_TYPE_NOT_DETECT_TEXT "unknown"
- #define WD_CloseLocal(h) CloseHandle(h)
- #define WD_UStreamRead(hFile, pBuffer, dwNumberOfBytesToRead, dwNumberOfBytesRead)
- #define WD_UStreamWrite(hFile, pBuffer, dwNumberOfBytesToWrite, dwNumberOfBytesWritten)
- #define WD_OpenLocal()
- #define WD_OpenStreamLocal(read, sync)
- #define WD_FUNCTION WD_FUNCTION_LOCAL

- #define WD_Close WD_CloseLocal
- #define WD_Open WD_OpenLocal
- #define WD_StreamOpen WD_OpenStreamLocal
- #define WD_StreamClose WD_CloseLocal
- #define SIZE_OF_WD_DMA(pDma)
- #define SIZE_OF_WD_EVENT(pEvent)
- #define WD_Debug(h, pDebug) WD_FUNCTION(IOCTL_WD_DEBUG, h, pDebug, sizeof(WD_DEBUG), FALSE)
 - Sets debugging level for collecting debug messages.*
- #define WD_DebugDump(h, pDebugDump)
 - Retrieves debug messages buffer.*
- #define WD_DebugAdd(h, pDebugAdd) WD_FUNCTION(IOCTL_WD_DEBUG_ADD, h, pDebugAdd, sizeof(WD_DEBUG_ADD), FALSE)
 - Sends debug messages to the debug log.*
- #define WD_Transfer(h, pTransfer) WD_FUNCTION(IOCTL_WD_TRANSFER, h, pTransfer, sizeof(WD_TRANSFER), FALSE)
 - Executes a single read/write instruction to an I/O port or to a memory address.*
- #define WD_MultiTransfer(h, pTransferArray, dwNumTransfers)
 - Executes multiple read/write instructions to I/O ports and/or memory addresses.*
- #define WD_KernelBufLock(h, pKerBuf)
 - Allocates a contiguous or non-contiguous non-paged kernel buffer, and maps it to user address space.*
- #define WD_KernelBufUnlock(h, pKerBuf)
 - Frees kernel buffer.*
- #define WD_DMALock(h, pDma) WD_FUNCTION(IOCTL_WD_DMA_LOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
 - Enables contiguous-buffer or Scatter/Gather DMA.*
- #define WD_DMAUnlock(h, pDma) WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
 - Unlocks a DMA buffer.*
- #define WD_DMATransactionInit(h, pDma)
 - Initializes the transaction, allocates a DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.*
- #define WD_DMATransactionExecute(h, pDma)
 - Begins the execution of a specified DMA transaction.*
- #define WD_DMATransferCompletedAndCheck(h, pDma)
 - Notifies WinDriver that a device's DMA transfer operation is completed.*
- #define WD_DMATransactionRelease(h, pDma)
 - Terminates a specified DMA transaction without deleting the associated WD_DMA transaction structure.*
- #define WD_DMATransactionUninit(h, pDma) WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
 - Unlocks and frees the memory allocated for a DMA buffer transaction by a previous call to WD_DMATransactionInit()*
- #define WD_DMASyncCpu(h, pDma) WD_FUNCTION(IOCTL_WD_DMA_SYNC_CPU, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
 - Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.*
- #define WD_DMASyncIo(h, pDma) WD_FUNCTION(IOCTL_WD_DMA_SYNC_IO, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
 - Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.*
- #define WD_CardRegister(h, pCard)
 - Card registration function.*
- #define WD_CardUnregister(h, pCard)
 - Unregisters a device and frees the resources allocated to it.*

- `#define WD_IpcRegister(h, plpcRegister)`
Registers an application with WinDriver IPC.
- `#define WD_IpcUnRegister(h, pProclInfo)`
This function unregisters the user application from WinDriver IPC.
- `#define WD_IpcScanProcs(h, plpcScanProcs)`
Scans and returns information of all WinDriver IPC registered processes that share the application process groupID.
- `#define WD_IpcSend(h, plpcSend) WD_FUNCTION(IOCTL_WD_IPC_SEND, h, plpcSend, sizeof(WD_IPC_SEND), FALSE)`
Sends a message to other processes, depending on the content of the plpcSend struct.
- `#define WD_SharedIntEnable(h, plpcRegister)`
Enables the shared interrupts mechanism of WinDriver.
- `#define WD_SharedIntDisable(h) WD_FUNCTION(IOCTL_WD_IPC_SHARED_INT_DISABLE, h, 0, 0, FALSE)`
Disables the Shared Interrupts mechanism of WinDriver for all processes.
- `#define WD_PciSriovEnable(h, pPciSRIOV)`
Enables SR-IOV for a supported device.
- `#define WD_PciSriovDisable(h, pPciSRIOV)`
Disables SR-IOV for a supported device and removes all the assigned VFs.
- `#define WD_PciSriovGetNumVFs(h, pPciSRIOV)`
Gets the number of virtual functions assigned to a supported device.
- `#define WD_CardCleanupSetup(h, pCardCleanup)`
*Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:
The application exits abnormally.*
- `#define WD_PciScanCards(h, pPciScan)`
Detects PCI devices installed on the PCI bus, which conform to the input criteria (vendor ID and/or card ID), and returns the number and location (bus, slot and function) of the detected devices.
- `#define WD_PciScanCaps(h, pPciScanCaps)`
Scans the specified PCI capabilities group of the given PCI slot for the specified capability (or for all capabilities).
- `#define WD_PciGetCardInfo(h, pPciCard)`
Retrieves PCI device's resource information (i.e., Memory ranges, I/O ranges, Interrupt lines).
- `#define WD_PciConfigDump(h, pPciConfigDump)`
Reads/writes from/to the PCI configuration space of a selected PCI card or the extended configuration space of a selected PCI Express card.
- `#define WD_Version(h, pVerInfo) WD_FUNCTION(IOCTL_WD_VERSION, h, pVerInfo, sizeof(WD_VERSION), FALSE)`
Returns the version number of the WinDriver kernel module currently running.
- `#define WD_License(h, pLicense) WD_FUNCTION(IOCTL_WD_LICENSE, h, pLicense, sizeof(WD_LICENSE), FALSE)`
Transfers the license string to the WinDriver kernel module When using the high-level WDC library APIs, described in the WinDriver PCI Manual, the license registration is done via the `WDC_DriverOpen()` function, so you do not need to call `WD_License()` directly.
- `#define WD_KernelPlugInOpen(h, pKernelPlugIn)`
Obtain a valid handle to the Kernel PlugIn.
- `#define WD_KernelPlugInClose(h, pKernelPlugIn)`
Closes the WinDriver Kernel PlugIn handle obtained from `WD_KernelPlugInOpen()`
- `#define WD_KernelPlugInCall(h, pKernelPlugInCall)`
Calls a routine in the Kernel PlugIn to be executed.
- `#define WD_IntEnable(h, plInterrupt) WD_FUNCTION(IOCTL_WD_INT_ENABLE, h, plInterrupt, sizeof(WD_INTERRUPT), FALSE)`
Registers an interrupt service routine (ISR) to be called upon interrupt.
- `#define WD_IntDisable(h, plInterrupt)`
Disables interrupt processing.

- #define WD_IntCount(h, plInterrupt) WD_FUNCTION(IOCTL_WD_INT_COUNT, h, plInterrupt, sizeof(WD_INTERRUPT), FALSE)
Retrieves the interrupts count since the call to WD_IntEnable()
- #define WD_IntWait(h, plInterrupt) WD_FUNCTION(IOCTL_WD_INT_WAIT, h, plInterrupt, sizeof(WD_INTERRUPT), TRUE)
Waits for an interrupt.
- #define WD_Sleep(h, pSleep) WD_FUNCTION(IOCTL_WD_SLEEP, h, pSleep, sizeof(WD_SLEEP), FALSE)
Delays execution for a specific duration of time.
- #define WD_EventRegister(h, pEvent)
- #define WD_EventUnregister(h, pEvent)
- #define WD_EventPull(h, pEvent) WD_FUNCTION(IOCTL_WD_EVENT_PULL, h, pEvent, SIZE_OF_WD_EVENT(pEvent), FALSE)
- #define WD_EventSend(h, pEvent) WD_FUNCTION(IOCTL_WD_EVENT_SEND, h, pEvent, SIZE_OF_WD_EVENT(pEvent), FALSE)
- #define WD_Usage(h, pStop) WD_FUNCTION(IOCTL_WD_USAGE, h, pStop, sizeof(WD_USAGE), FALSE)
- #define WD_UGetDeviceData(h, pGetDevData)
- #define WD_GetDeviceProperty(h, pGetDevProperty)
- #define WD_USetInterface(h, pSetIfc)
- #define WD_UResetPipe(h, pResetPipe)
- #define WD_UTransfer(h, pTrans) WD_FUNCTION(IOCTL_WDU_TRANSFER, h, pTrans, sizeof(WDU_TRANSFER), TRUE);
- #define WD_UHaltTransfer(h, pHaltTrans)
- #define WD_UWakeup(h, pWakeup) WD_FUNCTION(IOCTL_WDU_WAKEUP, h, pWakeup, sizeof(WDU_WAKEUP), FALSE);
- #define WD_USelectiveSuspend(h, pSelectiveSuspend)
- #define WD_UResetDevice(h, pResetDevice)
- #define WD_UStreamOpen(h, pStream) WD_FUNCTION(IOCTL_WDU_STREAM_OPEN, h, pStream, sizeof(WDU_STREAM), FALSE);
- #define WD_UStreamClose(h, pStream) WD_FUNCTION(IOCTL_WDU_STREAM_CLOSE, h, pStream, sizeof(WDU_STREAM), FALSE);
- #define WD_UStreamStart(h, pStream) WD_FUNCTION(IOCTL_WDU_STREAM_START, h, pStream, sizeof(WDU_STREAM), FALSE);
- #define WD_UStreamStop(h, pStream) WD_FUNCTION(IOCTL_WDU_STREAM_STOP, h, pStream, sizeof(WDU_STREAM), FALSE);
- #define WD_UStreamFlush(h, pStream) WD_FUNCTION(IOCTL_WDU_STREAM_FLUSH, h, pStream, sizeof(WDU_STREAM), FALSE);
- #define WD_UStreamGetStatus(h, pStreamStatus)
- #define __ALIGN_DOWN(val, alignment) ((val) & ~((alignment) - 1))
- #define __ALIGN_UP(val, alignment) ((val) + (alignment) - 1) & ~((alignment) - 1))
- #define MIN(a, b) ((a) > (b) ? (b) : (a))
- #define MAX(a, b) ((a) > (b) ? (a) : (b))
- #define SAFE_STRING(s) ((s) ? (s) : "")
- #define UNUSED_VAR(x) (void)x

Typedefs

- typedef unsigned __int64 UINT64
- typedef unsigned char BYTE
- typedef unsigned short int WORD
- typedef unsigned int UINT32
- typedef UINT32 KPTR
- typedef size_t UPTR
- typedef UINT64 DMA_ADDR
- typedef UINT64 PHYS_ADDR

- typedef struct `WD_DMA_PAGE` `WD_DMA_PAGE_V80`
- typedef void(`DLLCALLCONV` * `DMA_TRANSACTION_CALLBACK`) (`PVOID` `pData`)
- typedef struct `WD_DMA` `WD_DMA_V80`
- typedef struct `WD_KERNEL_BUFFER` `WD_KERNEL_BUFFER_V121`
- typedef struct `WD_TRANSFER` `WD_TRANSFER_V61`
- typedef struct `WD_KERNEL_PLUGIN_CALL` `WD_KERNEL_PLUGIN_CALL_V40`
- typedef struct `WD_INTERRUPT` `WD_INTERRUPT_V91`
- typedef struct `WD_VERSION` `WD_VERSION_V30`
- typedef struct `WD_LICENSE` `WD_LICENSE_V122`
- typedef `DWORD` `WD_BUS_TYPE`
- typedef struct `WD_BUS` `WD_BUS_V30`
- typedef struct `WD_ITEMS` `WD_ITEMS_V118`
- typedef struct `WD_CARD` `WD_CARD_V118`
- typedef struct `WD_CARD_REGISTER` `WD_CARD_REGISTER_V118`
- typedef struct `WD_IPC_PROCESS` `WD_IPC_PROCESS_V121`
- typedef struct `WD_IPC_REGISTER` `WD_IPC_REGISTER_V121`
- typedef struct `WD_IPC_SCAN_PROCS` `WD_IPC_SCAN_PROCS_V121`
- typedef struct `WD_IPC_SEND` `WD_IPC_SEND_V121`
- typedef struct `WD_PCI_SCAN_CARDS` `WD_PCI_SCAN_CARDS_V124`
- typedef struct `WD_PCI_SCAN_CAPS` `WD_PCI_SCAN_CAPS_V118`
- typedef struct `WD_PCI_SRIOV` `WD_PCI_SRIOV_V122`
- typedef struct `WD_PCI_CARD_INFO` `WD_PCI_CARD_INFO_V118`
- typedef struct `WD_PCI_CONFIG_DUMP` `WD_PCI_CONFIG_DUMP_V30`
- typedef struct `WD_SLEEP` `WD_SLEEP_V40`
- typedef struct `WD_DEBUG` `WD_DEBUG_V40`
- typedef struct `WD_DEBUG_DUMP` `WD_DEBUG_DUMP_V40`
- typedef struct `WD_DEBUG_ADD` `WD_DEBUG_ADD_V503`
- typedef struct `WD_KERNEL_PLUGIN` `WD_KERNEL_PLUGIN_V40`
- typedef `DWORD` `WD_EVENT_TYPE`
- typedef struct `WD_EVENT` `WD_EVENT_V121`

Enumerations

- enum `WD_TRANSFER_CMD` {
 `CMD_NONE` = 0 , `CMD_END` = 1 , `CMD_MASK` = 2 , `RP_BYT`E = 10 ,
 `RP_WORD` = 11 , `RP_DWORD` = 12 , `WP_BYT`E = 13 , `WP_WORD` = 14 ,
 `WP_DWORD` = 15 , `RP_QWORD` = 16 , `WP_QWORD` = 17 , `RP_SBYT`E = 20 ,
 `RP_SWOR`D = 21 , `RP_SDWORD` = 22 , `WP_SBYT`E = 23 , `WP_SWOR`D = 24 ,
 `WP_SDWORD` = 25 , `RP_SQWORD` = 26 , `WP_SQWORD` = 27 , `RM_BYT`E = 30 ,
 `RM_WORD` = 31 , `RM_DWORD` = 32 , `WM_BYT`E = 33 , `WM_WORD` = 34 ,
 `WM_DWORD` = 35 , `RM_QWORD` = 36 , `WM_QWORD` = 37 , `RM_SBYT`E = 40 ,
 `RM_SWOR`D = 41 , `RM_SDWORD` = 42 , `WM_SBYT`E = 43 , `WM_SWOR`D = 44 ,
 `WM_SDWORD` = 45 , `RM_SQWORD` = 46 , `WM_SQWORD` = 47 }
- IN WD_TRANSFER_CMD and WD_Transfer() DWORD stands for 32 bits and QWORD is 64 bit.*
- enum { `WD_DMA_PAGES` = 256 }
- enum `WD_DMA_OPTIONS` {
 `DMA_KERNEL_BUFFER_ALLOC` = 0x1 , `DMA_KBUF_BELOW_16M` = 0x2 , `DMA_LARGE_BUFFER` = 0x4 ,
 `DMA_ALLOW_CACHE` = 0x8 ,
 `DMA_KERNEL_ONLY_MAP` = 0x10 , `DMA_FROM_DEVICE` = 0x20 , `DMA_TO_DEVICE` = 0x40 ,
 `DMA_TO_FROM_DEVICE` = (`DMA_FROM_DEVICE` | `DMA_TO_DEVICE`) ,
 `DMA_ALLOW_64BIT_ADDRESS` = 0x80 , `DMA_ALLOW_NO_HCARD` = 0x100 , `DMA_GET_EXISTING_BUF` = 0x200 , `DMA_RESERVED_MEM` = 0x400 ,
 `DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH` = 0x800 , `DMA_GET_PREALLOCATED_BUFFERS_ONLY` = 0x1000 , `DMA_TRANSACTION` = 0x2000 , `DMA_GPUDIRECT` = 0x4000 ,
 `DMA_DISABLE_MERGE_ADJACENT_PAGES` = 0x8000 }
- enum { `WD_MATCH_EXCLUDE` = 0x1 }

- enum `WD_KER_BUF_OPTION` { `KER_BUF_ALLOC_NON_CONTIG` = 0x0001 , `KER_BUF_ALLOC_CONTIG` = 0x0002 , `KER_BUF_ALLOC_CACHED` = 0x0004 , `KER_BUF_GET_EXISTING_BUF` = 0x0008 }
- enum {
 - `INTERRUPT_LATCHED` = 0x00 , `INTERRUPT_LEVEL_SENSITIVE` = 0x01 , `INTERRUPT_CMD_COPY` = 0x02 , `INTERRUPT_CE_INT_ID` = 0x04 ,
 - `INTERRUPT_CMD_RETURN_VALUE` = 0x08 , `INTERRUPT_MESSAGE` = 0x10 , `INTERRUPT_MESSAGE_X` = 0x20 , `INTERRUPT_DONT_GET_MSI_MESSAGE` = 0x40 }
- enum `WD_INTERRUPT_WAIT_RESULT` { `INTERRUPT_RECEIVED` = 0 , `INTERRUPT_STOPPED` , `INTERRUPT_INTERRUPTED` }
- enum {
 - `WD_BUS_USB` = (int)0xffffffff , `WD_BUS_UNKNOWN` = 0 , `WD_BUS_ISA` = 1 , `WD_BUS_EISA` = 2 ,
 - `WD_BUS_PCI` = 5 }
- enum `ITEM_TYPE` {
 - `ITEM_NONE` = 0 , `ITEM_INTERRUPT` = 1 , `ITEM_MEMORY` = 2 , `ITEM_IO` = 3 ,
 - `ITEM_BUS` = 5 }
- enum `WD_ITEM_MEM_OPTIONS` { `WD_ITEM_MEM_DO_NOT_MAP_KERNEL` = 0x1 , `WD_ITEM_MEM_ALLOW_CACHE` = 0x2 , `WD_ITEM_MEM_USER_MAP` = 0x4 }
- enum { `WD_CARD_ITEMS` = 128 }
- enum { `WD_IPC_MAX_PROCS` = 0x40 }
- enum { `WD_IPC_UID_UNICAST` = 0x1 , `WD_IPC_SUBGROUP_MULTICAST` = 0x2 , `WD_IPC_MULTICAST` = 0x4 }
- enum { `WD_FORCE_CLEANUP` = 0x1 }
- enum { `WD_PCI_CARDS` = 256 }
- enum `WD_PCI_SCAN_OPTIONS` { `WD_PCI_SCAN_DEFAULT` = 0x0 , `WD_PCI_SCAN_BY_TOPOLOGY` = 0x1 , `WD_PCI_SCAN_REGISTERED` = 0x2 , `WD_PCI_SCAN_INCLUDE_DOMAINS` = 0x4 }
- enum { `WD_PCI_MAX_CAPS` = 50 }
- enum { `WD_PCI_CAP_ID_ALL` = 0x0 }
- enum `WD_PCI_SCAN_CAPS_OPTIONS` { `WD_PCI_SCAN_CAPS_BASIC` = 0x1 , `WD_PCI_SCAN_CAPS_EXTENDED` = 0x2 }
- enum `PCI_ACCESS_RESULT` { `PCI_ACCESS_OK` = 0 , `PCI_ACCESS_ERROR` = 1 , `PCI_BAD_BUS` = 2 , `PCI_BAD_SLOT` = 3 }
- enum { `SLEEP_BUSY` = 0 , `SLEEP_NON_BUSY` = 1 }
- enum `DEBUG_LEVEL` {
 - `D_OFF` = 0 , `D_ERROR` = 1 , `D_WARN` = 2 , `D_INFO` = 3 ,
 - `D_TRACE` = 4 }
- enum `DEBUG_SECTION` {
 - `S_ALL` = (int)0xffffffff , `S_IO` = 0x00000008 , `S_MEM` = 0x00000010 , `S_INT` = 0x00000020 ,
 - `S_PCI` = 0x00000040 , `S_DMA` = 0x00000080 , `S_MISC` = 0x00000100 , `S_LICENSE` = 0x00000200 ,
 - `S_PNP` = 0x00001000 , `S_CARD_REG` = 0x00002000 , `S_KER_DRV` = 0x00004000 , `S_USB` = 0x00008000
 - ,
 - `S_KER_PLUG` = 0x00010000 , `S_EVENT` = 0x00020000 , `S_IPC` = 0x00040000 , `S_KER_BUF` = 0x00080000 }
- enum `DEBUG_COMMAND` {
 - `DEBUG_STATUS` = 1 , `DEBUG_SET_FILTER` = 2 , `DEBUG_SET_BUFFER` = 3 , `DEBUG_CLEAR_BUFFER` = 4 ,
 - `DEBUG_DUMP_SEC_ON` = 5 , `DEBUG_DUMP_SEC_OFF` = 6 , `KERNEL_DEBUGGER_ON` = 7 ,
 - `KERNEL_DEBUGGER_OFF` = 8 ,
 - `DEBUG_DUMP_CLOCK_ON` = 9 , `DEBUG_DUMP_CLOCK_OFF` = 10 , `DEBUG_CLOCK_RESET` = 11 }
- enum `WD_GET_DEVICE_PROPERTY_OPTION` { `WD_DEVICE_PCI` = 0x1 , `WD_DEVICE_USB` = 0x2 }
 - IOCTL Structures.*
- enum `WD_ERROR_CODES` {
 - `WD_STATUS_SUCCESS` = 0 , `WD_STATUS_INVALID_WD_HANDLE` = (int)0xffffffff , `WD_WINDRIVER_STATUS_ERROR` = 0x20000000L , `WD_INVALID_HANDLE` = 0x20000001L ,
 - `WD_INVALID_PIPE_NUMBER` = 0x20000002L , `WD_READ_WRITE_CONFLICT` = 0x20000003L ,
 - `WD_ZERO_PACKET_SIZE` = 0x20000004L , `WD_INSUFFICIENT_RESOURCES` = 0x20000005L ,
 - `WD_UNKNOWN_PIPE_TYPE` = 0x20000006L , `WD_SYSTEM_INTERNAL_ERROR` = 0x20000007L ,
 - `WD_DATA_MISMATCH` = 0x20000008L , `WD_NO_LICENSE` = 0x20000009L ,

```
WD_NOT_IMPLEMENTED = 0x20000000aL, WD_KERPLUG_FAILURE = 0x2000000bL, WD_FAILED_ENABLING_INTERRUPT
= 0x2000000cL, WD_INTERRUPT_NOT_ENABLED = 0x2000000dL,
WD_RESOURCE_OVERLAP = 0x2000000eL, WD_DEVICE_NOT_FOUND = 0x2000000fL, WD_WRONG_UNIQUE_ID
= 0x20000010L, WD_OPERATION_ALREADY_DONE = 0x20000011L,
WD_USB_DESCRIPTOR_ERROR = 0x20000012L, WD_SET_CONFIGURATION FAILED = 0x20000013L
, WD_CANT_OBTAIN_PDO = 0x20000014L, WD_TIME_OUT_EXPIRED = 0x20000015L,
WD_IRP_CANCELED = 0x20000016L, WD_FAILED_USER_MAPPING = 0x20000017L, WD_FAILED_KERNEL_MAPPING
= 0x20000018L, WD_NO_RESOURCES_ON_DEVICE = 0x20000019L,
WD_NO_EVENTS = 0x2000001aL, WD_INVALID_PARAMETER = 0x2000001bL, WD_INCORRECT_VERSION
= 0x2000001cL, WD_TRY AGAIN = 0x2000001dL,
WD_WINDRIVER_NOT_FOUND = 0x2000001eL, WD_INVALID_IOCTL = 0x2000001fL, WD_OPERATION_FAILED
= 0x20000020L, WD_INVALID_32BIT_APP = 0x20000021L,
WD_TOO_MANY_HANDLES = 0x20000022L, WD_NO_DEVICE_OBJECT = 0x20000023L, WD_MORE_PROCESSING_REQUIRED
= (int)0xC0000016L, WD_USBD_STATUS_SUCCESS = 0x00000000L,
WD_USBD_STATUS_PENDING = 0x40000000L, WD_USBD_STATUS_ERROR = (int)0x80000000L ,
WD_USBD_STATUS_HALTED = (int)0xC0000000L, WD_USBD_STATUS_CRC = (int)0xC0000001L ,
WD_USBD_STATUS_BTSTUFF = (int)0xC0000002L, WD_USBD_STATUS_DATA_TOGGLE_MISMATCH =
(int)0xC0000003L, WD_USBD_STATUS_STALL_PID = (int)0xC0000004L, WD_USBD_STATUS_DEV_NOT_RESPONDING
= (int)0xC0000005L ,
WD_USBD_STATUS_PID_CHECK_FAILURE = (int)0xC0000006L, WD_USBD_STATUS_UNEXPECTED_PID
= (int)0xC0000007L, WD_USBD_STATUS_DATA_OVERRUN = (int)0xC0000008L, WD_USBD_STATUS_DATA_UNDERUN
= (int)0xC0000009L ,
WD_USBD_STATUS_RESERVED1 = (int)0xC000000AL, WD_USBD_STATUS_RESERVED2 = (int)0xC000000BL,
WD_USBD_STATUS_BUFFER_OVERRUN = (int)0xC000000CL, WD_USBD_STATUS_BUFFER_UNDERUN
= (int)0xC000000DL ,
WD_USBD_STATUS_NOT_ACCESSED = (int)0xC000000FL, WD_USBD_STATUS_FIFO = (int)0xC0000010L,
WD_USBD_STATUS_XACT_ERROR = (int)0xC0000011L, WD_USBD_STATUS_BABBLE_DETECTED
= (int)0xC0000012L ,
WD_USBD_STATUS_DATA_BUFFER_ERROR = (int)0xC0000013L, WD_USBD_STATUS_CANCELED =
(int)0xC0010000L, WD_USBD_STATUS_ENDPOINT_HALTED = (int)0xC0000030L, WD_USBD_STATUS_NO_MEMORY
= (int)0x80000100L ,
WD_USBD_STATUS_INVALID_URB_FUNCTION = (int)0x80000200L, WD_USBD_STATUS_INVALID_PARAMETER
= (int)0x80000300L, WD_USBD_STATUS_ERROR_BUSY = (int)0x80000400L, WD_USBD_STATUS_REQUEST_FAILED
= (int)0x80000500L ,
WD_USBD_STATUS_INVALID_PIPE_HANDLE = (int)0x80000600L, WD_USBD_STATUS_NO_BANDWIDTH
= (int)0x80000700L, WD_USBD_STATUS_INTERNAL_HC_ERROR = (int)0x80000800L, WD_USBD_STATUS_ERROR_SHOT
= (int)0x80000900L ,
WD_USBD_STATUS_BAD_START_FRAME = (int)0xC000A00L, WD_USBD_STATUS_ISOCH_REQUEST_FAILED
= (int)0xC000B00L , WD_USBD_STATUS_FRAME_CONTROL OWNED = (int)0xC000C00L ,
WD_USBD_STATUS_FRAME_CONTROL_NOT OWNED = (int)0xC000D00L ,
WD_USBD_STATUS_NOT_SUPPORTED = (int)0xC000E00L, WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR
= (int)0xC000F00L , WD_USBD_STATUS_INSUFFICIENT_RESOURCES = (int)0xC001000L ,
WD_USBD_STATUS_SET_CONFIG_FAILED = (int)0xC0002000L ,
WD_USBD_STATUS_BUFFER_TOO_SMALL = (int)0xC0003000L, WD_USBD_STATUS_INTERFACE_NOT_FOUND
= (int)0xC0004000L, WD_USBD_STATUS_INAVLID_PIPE_FLAGS = (int)0xC0005000L, WD_USBD_STATUS_TIMEOUT
= (int)0xC0006000L ,
WD_USBD_STATUS_DEVICE_GONE = (int)0xC0007000L, WD_USBD_STATUS_STATUS_NOT_MAPPED
= (int)0xC0008000L , WD_USBD_STATUS_ISO_NOT_ACCESSED_BY_HW = (int)0xC0020000L ,
WD_USBD_STATUS_ISO_TD_ERROR = (int)0xC0030000L ,
WD_USBD_STATUS_ISO_NA_LATE_USBPORT = (int)0xC0040000L, WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE
= (int)0xC0050000L }
• enum WD_EVENT_ACTION {
WD_INSERT = 0x1 , WD_REMOVE = 0x2 , WD_OBSOLETE = 0x8 , WD_POWER_CHANGED_D0 = 0x10 ,
WD_POWER_CHANGED_D1 = 0x20 , WD_POWER_CHANGED_D2 = 0x40 , WD_POWER_CHANGED_D3
= 0x80 , WD_POWER_SYSTEM_WORKING = 0x100 ,
WD_POWER_SYSTEM_SLEEPING1 = 0x200 , WD_POWER_SYSTEM_SLEEPING2 = 0x400 ,
WD_POWER_SYSTEM_SLEEPING3 = 0x800 , WD_POWER_SYSTEM_HIBERNATE = 0x1000 ,
WD_POWER_SYSTEM_SHUTDOWN = 0x2000 , WD_IPC_UNICAST_MSG = 0x4000 , WD_IPC_MULTICAST_MSG
```

- = 0x8000 }
- enum `WD_EVENT_OPTION` { `WD_ACKNOWLEDGE` = 0x1 , `WD_ACCEPT_CONTROL` = 0x2 }
- enum { `WD_EVENT_TYPE_UNKNOWN` = 0 , `WD_EVENT_TYPE_PCI` = 1 , `WD_EVENT_TYPE_USB` = 3 ,
`WD_EVENT_TYPE_IPC` = 4 }
- enum { `WD_USB_HARD_RESET` = 1 , `WD_USB_CYCLE_PORT` = 2 }

Functions

- const char *`DLLCALLCONV WD_DriverName` (const char *`sName`)
Sets the name of the WinDriver kernel module, which will be used by the calling application.
- `WD_OS_INFO DLLCALLCONV get_os_type` (void)
Retrieves the type of the operating system in section.
- DWORD `DLLCALLCONV check_secureBoot_enabled` (void)
Checks whether the Secure Boot feature is enabled on the system.

Macro Definition Documentation

__ALIGN_DOWN

```
#define __ALIGN_DOWN(
    val,
    alignment) ( (val) & ~((alignment) - 1) )
```

Definition at line 2707 of file `windrvr.h`.

__ALIGN_UP

```
#define __ALIGN_UP(
    val,
    alignment) ( ((val) + (alignment) - 1) & ~((alignment) - 1) )
```

Definition at line 2708 of file `windrvr.h`.

__FUNCTION__

```
#define __FUNCTION__ __func__
```

Definition at line 52 of file `windrvr.h`.

In

```
#define _In_
Definition at line 37 of file windrvr.h.
```

Inout

```
#define _Inout_
Definition at line 38 of file windrvr.h.
```

Out

```
#define _Out_
Definition at line 39 of file windrvr.h.
```

Outptr

```
#define _Outptr_
Definition at line 40 of file windrvr.h.
```

BZERO

```
#define BZERO(
    buf ) memset (&(buf), 0, sizeof(buf))
Definition at line 1528 of file windrvr.h.
```

CTL_CODE

```
#define CTL_CODE(
    DeviceType,
    Function,
    Method,
    Access )
Value:
( \
((DeviceType)«16) | ((Access)«14) | ((Function)«2) | (Method) \
)
Definition at line 1536 of file windrvr.h.
```

DEBUG_USER_BUF_LEN

```
#define DEBUG_USER_BUF_LEN 2048
Definition at line 992 of file windrvr.h.
```

DLLCALLCONV

```
#define DLLCALLCONV
Definition at line 32 of file windrvr.h.
```

DMA_ADDRESS_WIDTH_MASK

```
#define DMA_ADDRESS_WIDTH_MASK 0x7f000000
Definition at line 471 of file windrvr.h.
```

DMA_BIT_MASK

```
#define DMA_BIT_MASK(
    n ) (((n) == 64) ? ~0ULL : ((1ULL<<(n))-1))
Definition at line 413 of file windrvr.h.
```

DMA_DIRECTION_MASK

```
#define DMA_DIRECTION_MASK DMA_TO_FROM_DEVICE
Definition at line 480 of file windrvr.h.
```

DMA_OPTIONS_ADDRESS_WIDTH_SHIFT

```
#define DMA_OPTIONS_ADDRESS_WIDTH_SHIFT
```

Value:

24

Definition at line 486 of file [windrvr.h](#).

DMA_OPTIONS_ALL

```
#define DMA_OPTIONS_ALL
```

Value:

```
(DMA_KERNEL_BUFFER_ALLOC | DMA_KBUF_BELOW_16M | DMA_LARGE_BUFFER \
| DMA_ALLOW_CACHE | DMA_KERNEL_ONLY_MAP | DMA_FROM_DEVICE | DMA_TO_DEVICE \
| DMA_ALLOW_64BIT_ADDRESS | DMA_ALLOW_NO_HCARD | DMA_GET_EXISTING_BUF \
| DMA_RESERVED_MEM | DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH \
| DMA_ADDRESS_WIDTH_MASK)
```

Definition at line 473 of file [windrvr.h](#).

DMA_READ_FROM_DEVICE

```
#define DMA_READ_FROM_DEVICE DMA_FROM_DEVICE
```

Macros for backward compatibility.

Definition at line 483 of file [windrvr.h](#).

DMA_WRITE_TO_DEVICE

```
#define DMA_WRITE_TO_DEVICE DMA_TO_DEVICE
```

Definition at line 484 of file [windrvr.h](#).

FILE_ANY_ACCESS

```
#define FILE_ANY_ACCESS 0
```

Definition at line 1544 of file [windrvr.h](#).

FILE_READ_ACCESS

```
#define FILE_READ_ACCESS 1
```

file & pipe

Definition at line 1545 of file [windrvr.h](#).

FILE_WRITE_ACCESS

```
#define FILE_WRITE_ACCESS 2
```

file & pipe

Definition at line 1546 of file [windrvr.h](#).

FUNC_MASK

```
#define FUNC_MASK 0x0
```

Definition at line 1566 of file [windrvr.h](#).

INSTALLATION_TYPE_NOT_DETECT_TEXT

```
#define INSTALLATION_TYPE_NOT_DETECT_TEXT "unknown"  
Definition at line 1672 of file windrvr.h.
```

INVALID_HANDLE_VALUE

```
#define INVALID_HANDLE_VALUE ((HANDLE)(-1))  
Definition at line 1532 of file windrvr.h.
```

IOCTL_WD_CARD_CLEANUP_SETUP

```
#define IOCTL_WD_CARD_CLEANUP_SETUP WD_CTL_CODE(0x995)  
Definition at line 1625 of file windrvr.h.
```

IOCTL_WD_CARD_REGISTER

```
#define IOCTL_WD_CARD_REGISTER WD_CTL_CODE(0x9e6)  
Definition at line 1608 of file windrvr.h.
```

IOCTL_WD_CARD_UNREGISTER

```
#define IOCTL_WD_CARD_UNREGISTER WD_CTL_CODE(0x9e7)  
Definition at line 1607 of file windrvr.h.
```

IOCTL_WD_DEBUG

```
#define IOCTL_WD_DEBUG WD_CTL_CODE(0x928)  
Definition at line 1605 of file windrvr.h.
```

IOCTL_WD_DEBUG_ADD

```
#define IOCTL_WD_DEBUG_ADD WD_CTL_CODE(0x964)  
Definition at line 1615 of file windrvr.h.
```

IOCTL_WD_DEBUG_DUMP

```
#define IOCTL_WD_DEBUG_DUMP WD_CTL_CODE(0x929)  
Definition at line 1606 of file windrvr.h.
```

IOCTL_WD_DMA_LOCK

```
#define IOCTL_WD_DMA_LOCK WD_CTL_CODE(0x9be)  
Definition at line 1590 of file windrvr.h.
```

IOCTL_WD_DMA_SYNC_CPU

```
#define IOCTL_WD_DMA_SYNC_CPU WD_CTL_CODE(0x99f)  
Definition at line 1626 of file windrvr.h.
```

IOCTL_WD_DMA_SYNC_IO

```
#define IOCTL_WD_DMA_SYNC_IO WD_CTL_CODE(0x9a0)
```

Definition at line 1627 of file [windrvr.h](#).

IOCTL_WD_DMA_TRANSACTION_EXECUTE

```
#define IOCTL_WD_DMA_TRANSACTION_EXECUTE WD_CTL_CODE(0x9ff)
```

Definition at line 1646 of file [windrvr.h](#).

IOCTL_WD_DMA_TRANSACTION_INIT

```
#define IOCTL_WD_DMA_TRANSACTION_INIT WD_CTL_CODE(0x9fe)
```

Definition at line 1645 of file [windrvr.h](#).

IOCTL_WD_DMA_TRANSACTION_RELEASE

```
#define IOCTL_WD_DMA_TRANSACTION_RELEASE WD_CTL_CODE(0xa01)
```

Definition at line 1648 of file [windrvr.h](#).

IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK

```
#define IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK WD_CTL_CODE(0xa00)
```

Definition at line 1647 of file [windrvr.h](#).

IOCTL_WD_DMA_UNLOCK

```
#define IOCTL_WD_DMA_UNLOCK WD_CTL_CODE(0x902)
```

Definition at line 1591 of file [windrvr.h](#).

IOCTL_WD_EVENT_PULL

```
#define IOCTL_WD_EVENT_PULL WD_CTL_CODE(0x9f1)
```

Definition at line 1613 of file [windrvr.h](#).

IOCTL_WD_EVENT_REGISTER

```
#define IOCTL_WD_EVENT_REGISTER WD_CTL_CODE(0x9ef)
```

Definition at line 1611 of file [windrvr.h](#).

IOCTL_WD_EVENT_SEND

```
#define IOCTL_WD_EVENT_SEND WD_CTL_CODE(0x9f2)
```

Definition at line 1614 of file [windrvr.h](#).

IOCTL_WD_EVENT_UNREGISTER

```
#define IOCTL_WD_EVENT_UNREGISTER WD_CTL_CODE(0x9f0)
```

Definition at line 1612 of file [windrvr.h](#).

IOCTL_WD_GET_DEVICE_PROPERTY

```
#define IOCTL_WD_GET_DEVICE_PROPERTY WD_CTL_CODE (0x990)
```

Definition at line 1624 of file [windrvr.h](#).

IOCTL_WD_INT_COUNT

```
#define IOCTL_WD_INT_COUNT WD_CTL_CODE (0x9ba)
```

Definition at line 1603 of file [windrvr.h](#).

IOCTL_WD_INT_DISABLE

```
#define IOCTL_WD_INT_DISABLE WD_CTL_CODE (0x9bb)
```

Definition at line 1602 of file [windrvr.h](#).

IOCTL_WD_INT_ENABLE

```
#define IOCTL_WD_INT_ENABLE WD_CTL_CODE (0x9b6)
```

Definition at line 1601 of file [windrvr.h](#).

IOCTL_WD_INT_WAIT

```
#define IOCTL_WD_INT_WAIT WD_CTL_CODE (0x9b9)
```

Definition at line 1609 of file [windrvr.h](#).

IOCTL_WD_IPC_REGISTER

```
#define IOCTL_WD_IPC_REGISTER WD_CTL_CODE (0x9eb)
```

Definition at line 1636 of file [windrvr.h](#).

IOCTL_WD_IPC_SCAN_PROCS

```
#define IOCTL_WD_IPC_SCAN_PROCS WD_CTL_CODE (0x9ed)
```

Definition at line 1638 of file [windrvr.h](#).

IOCTL_WD_IPC_SEND

```
#define IOCTL_WD_IPC_SEND WD_CTL_CODE (0x9ee)
```

Definition at line 1639 of file [windrvr.h](#).

IOCTL_WD_IPC_SHARED_INT_DISABLE

```
#define IOCTL_WD_IPC_SHARED_INT_DISABLE WD_CTL_CODE (0x9fd)
```

Definition at line 1644 of file [windrvr.h](#).

IOCTL_WD_IPC_SHARED_INT_ENABLE

```
#define IOCTL_WD_IPC_SHARED_INT_ENABLE WD_CTL_CODE (0x9fc)
```

Definition at line 1643 of file [windrvr.h](#).

IOCTL_WD_IPC_UNREGISTER

```
#define IOCTL_WD_IPC_UNREGISTER WD_CTL_CODE(0x9ec)
```

Definition at line 1637 of file [windrvr.h](#).

IOCTL_WD_KERNEL_BUF_LOCK

```
#define IOCTL_WD_KERNEL_BUF_LOCK WD_CTL_CODE(0x9f3)
```

Definition at line 1588 of file [windrvr.h](#).

IOCTL_WD_KERNEL_BUF_UNLOCK

```
#define IOCTL_WD_KERNEL_BUF_UNLOCK WD_CTL_CODE(0x9f4)
```

Definition at line 1589 of file [windrvr.h](#).

IOCTL_WD_KERNEL_PLUGIN_CALL

```
#define IOCTL_WD_KERNEL_PLUGIN_CALL WD_CTL_CODE(0x91d)
```

Definition at line 1600 of file [windrvr.h](#).

IOCTL_WD_KERNEL_PLUGIN_CLOSE

```
#define IOCTL_WD_KERNEL_PLUGIN_CLOSE WD_CTL_CODE(0x91c)
```

Definition at line 1599 of file [windrvr.h](#).

IOCTL_WD_KERNEL_PLUGIN_OPEN

```
#define IOCTL_WD_KERNEL_PLUGIN_OPEN WD_CTL_CODE(0x91b)
```

Definition at line 1598 of file [windrvr.h](#).

IOCTL_WD_LICENSE

```
#define IOCTL_WD_LICENSE WD_CTL_CODE(0x9f9)
```

Definition at line 1610 of file [windrvr.h](#).

IOCTL_WD_MULTI_TRANSFER

```
#define IOCTL_WD_MULTI_TRANSFER WD_CTL_CODE(0x98d)
```

Definition at line 1593 of file [windrvr.h](#).

IOCTL_WD_PCI_CONFIG_DUMP

```
#define IOCTL_WD_PCI_CONFIG_DUMP WD_CTL_CODE(0x91a)
```

Definition at line 1597 of file [windrvr.h](#).

IOCTL_WD_PCI_GET_CARD_INFO

```
#define IOCTL_WD_PCI_GET_CARD_INFO WD_CTL_CODE(0x9e8)
```

Definition at line 1595 of file [windrvr.h](#).

IOCTL_WD_PCI_SCAN_CAPS

```
#define IOCTL_WD_PCI_SCAN_CAPS WD_CTL_CODE(0x9e5)
```

Definition at line 1635 of file [windrvr.h](#).

IOCTL_WD_PCI_SCAN_CARDS

```
#define IOCTL_WD_PCI_SCAN_CARDS WD_CTL_CODE(0x9fa)
```

Definition at line 1594 of file [windrvr.h](#).

IOCTL_WD_PCI_SRIOV_DISABLE

```
#define IOCTL_WD_PCI_SRIOV_DISABLE WD_CTL_CODE(0x9f6)
```

Definition at line 1641 of file [windrvr.h](#).

IOCTL_WD_PCI_SRIOV_ENABLE

```
#define IOCTL_WD_PCI_SRIOV_ENABLE WD_CTL_CODE(0x9f5)
```

Definition at line 1640 of file [windrvr.h](#).

IOCTL_WD_PCI_SRIOV_GET_NUMVFS

```
#define IOCTL_WD_PCI_SRIOV_GET_NUMVFS WD_CTL_CODE(0x9f7)
```

Definition at line 1642 of file [windrvr.h](#).

IOCTL_WD_SLEEP

```
#define IOCTL_WD_SLEEP WD_CTL_CODE(0x927)
```

Definition at line 1604 of file [windrvr.h](#).

IOCTL_WD_TRANSFER

```
#define IOCTL_WD_TRANSFER WD_CTL_CODE(0x98c)
```

Definition at line 1592 of file [windrvr.h](#).

IOCTL_WD_USAGE

```
#define IOCTL_WD_USAGE WD_CTL_CODE(0x976)
```

Definition at line 1616 of file [windrvr.h](#).

IOCTL_WD_VERSION

```
#define IOCTL_WD_VERSION WD_CTL_CODE(0x910)
```

Definition at line 1596 of file [windrvr.h](#).

IOCTL_WDU_GET_DEVICE_DATA

```
#define IOCTL_WDU_GET_DEVICE_DATA WD_CTL_CODE(0x9a7)
```

Definition at line 1617 of file [windrvr.h](#).

IOCTL_WDU_HALT_TRANSFER

```
#define IOCTL_WDU_HALT_TRANSFER WD_CTL_CODE(0x985)
```

Definition at line 1621 of file [windrvr.h](#).

IOCTL_WDU_RESET_DEVICE

```
#define IOCTL_WDU_RESET_DEVICE WD_CTL_CODE(0x98b)
```

Definition at line 1623 of file [windrvr.h](#).

IOCTL_WDU_RESET_PIPE

```
#define IOCTL_WDU_RESET_PIPE WD_CTL_CODE(0x982)
```

Definition at line 1619 of file [windrvr.h](#).

IOCTL_WDU_SELECTIVE_SUSPEND

```
#define IOCTL_WDU_SELECTIVE_SUSPEND WD_CTL_CODE(0x9ae)
```

Definition at line 1634 of file [windrvr.h](#).

IOCTL_WDU_SET_INTERFACE

```
#define IOCTL_WDU_SET_INTERFACE WD_CTL_CODE(0x981)
```

Definition at line 1618 of file [windrvr.h](#).

IOCTL_WDU_STREAM_CLOSE

```
#define IOCTL_WDU_STREAM_CLOSE WD_CTL_CODE(0x9a9)
```

Definition at line 1629 of file [windrvr.h](#).

IOCTL_WDU_STREAM_FLUSH

```
#define IOCTL_WDU_STREAM_FLUSH WD_CTL_CODE(0x9aa)
```

Definition at line 1632 of file [windrvr.h](#).

IOCTL_WDU_STREAM_GET_STATUS

```
#define IOCTL_WDU_STREAM_GET_STATUS WD_CTL_CODE(0x9b5)
```

Definition at line 1633 of file [windrvr.h](#).

IOCTL_WDU_STREAM_OPEN

```
#define IOCTL_WDU_STREAM_OPEN WD_CTL_CODE(0x9a8)
```

Definition at line 1628 of file [windrvr.h](#).

IOCTL_WDU_STREAM_START

```
#define IOCTL_WDU_STREAM_START WD_CTL_CODE(0x9af)
```

Definition at line 1630 of file [windrvr.h](#).

IOCTL_WDU_STREAM_STOP

```
#define IOCTL_WDU_STREAM_STOP WD_CTL_CODE(0x9b0)
```

Definition at line 1631 of file [windrvr.h](#).

IOCTL_WDU_TRANSFER

```
#define IOCTL_WDU_TRANSFER WD_CTL_CODE(0x983)
```

Definition at line 1620 of file [windrvr.h](#).

IOCTL_WDU_WAKEUP

```
#define IOCTL_WDU_WAKEUP WD_CTL_CODE(0x98a)
```

Definition at line 1622 of file [windrvr.h](#).

KPRI

```
#define KPRI ""
```

formatting for printing a kernel pointer
Definition at line 340 of file [windrvr.h](#).

MAX

```
#define MAX(
```

a,
 b) ((a) > (b) ? (a) : (b))

Definition at line 2719 of file [windrvr.h](#).

METHOD_BUFFERED

```
#define METHOD_BUFFERED 0
```

Definition at line 1540 of file [windrvr.h](#).

METHOD_IN_DIRECT

```
#define METHOD_IN_DIRECT 1
```

Definition at line 1541 of file [windrvr.h](#).

METHOD_NEITHER

```
#define METHOD_NEITHER 3
```

Definition at line 1543 of file [windrvr.h](#).

METHOD_OUT_DIRECT

```
#define METHOD_OUT_DIRECT 2
```

Definition at line 1542 of file [windrvr.h](#).

MIN

```
#define MIN(  
    a,  
    b ) ((a) > (b) ? (b) : (a))
```

Definition at line 2716 of file [windrvr.h](#).

OS_CAN_NOT_DETECT_TEXT

```
#define OS_CAN_NOT_DETECT_TEXT "OS CAN NOT DETECT"  
Definition at line 1671 of file windrvr.h.
```

PAD_TO_64

```
#define PAD_TO_64(  
    pName )
```

Definition at line 496 of file [windrvr.h](#).

PRI64

```
#define PRI64 "I64"  
formatting for printing a 64bit variable  
Definition at line 328 of file windrvr.h.
```

REGKEY_BUFSIZE

```
#define REGKEY_BUFSIZE 256  
Definition at line 1670 of file windrvr.h.
```

SAFE_STRING

```
#define SAFE_STRING(  
    s ) ((s) ? (s) : "")
```

Definition at line 2721 of file [windrvr.h](#).

SIZE_OF_WD_DMA

```
#define SIZE_OF_WD_DMA(  
    pDma )
```

Value:

```
((DWORD)(sizeof(WD_DMA) + ((pDma)->dwPages <= WD_DMA_PAGES ? \  
0 : ((pDma)->dwPages - WD_DMA_PAGES) * sizeof(WD_DMA_PAGE))))
```

Definition at line 1819 of file [windrvr.h](#).

SIZE_OF_WD_EVENT

```
#define SIZE_OF_WD_EVENT(  
    pEvent )
```

Value:

```
((DWORD)(sizeof(WD_EVENT) + ((pEvent)->dwNumMatchTables > 0 ? \  
sizeof(WDU_MATCH_TABLE) * ((pEvent)->dwNumMatchTables - 1) : 0)))
```

Definition at line 1822 of file [windrvr.h](#).

strcmp

```
#define strcmp _strcmp
```

Definition at line 296 of file [windrvr.h](#).

UNUSED_VAR

```
#define UNUSED_VAR(
```

```
    x ) (void)x
```

Definition at line 2723 of file [windrvr.h](#).

UPRI

```
#define UPRI "1"
```

Definition at line 341 of file [windrvr.h](#).

va_copy

```
#define va_copy(
```

```
    ap2,
```

```
    ap1 ) (ap2)=(ap1)
```

Definition at line 303 of file [windrvr.h](#).

WD_ACTIONS_ALL

```
#define WD_ACTIONS_ALL (WD_ACTIONS_POWER | WD_INSERT | WD_REMOVE)
```

Definition at line 1466 of file [windrvr.h](#).

WD_ACTIONS_POWER

```
#define WD_ACTIONS_POWER
```

Value:

```
(WD_POWER_CHANGED_D0 | WD_POWER_CHANGED_D1 | \
WD_POWER_CHANGED_D2 | WD_POWER_CHANGED_D3 | WD_POWER_SYSTEM_WORKING | \
WD_POWER_SYSTEM_SLEEPING1 | WD_POWER_SYSTEM_SLEEPING3 | \
WD_POWER_SYSTEM_HIBERNATE | WD_POWER_SYSTEM_SHUTDOWN)
```

Definition at line 1462 of file [windrvr.h](#).

WD_CardCleanupSetup

```
#define WD_CardCleanupSetup(
```

```
    h,
```

```
    pCardCleanup )
```

Value:

```
WD_FUNCTION(IOCTL_WD_CARD_CLEANUP_SETUP, h, pCardCleanup, \
sizeof(WD_CARD_CLEANUP), FALSE)
```

Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:
The application exits abnormally.
The application exits normally but without unregistering 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 unregistered.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pCardCleanup</i>	Pointer to a card clean-up information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

You should call this function right after calling [WD_CardRegister\(\)](#)

```
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));
}
```

Definition at line 2395 of file [windrvr.h](#).

WD_CardRegister

```
#define WD_CardRegister(
    h,
    pCard )
```

Value:

```
WD_FUNCTION(IOCTL_WD_CARD_REGISTER, h, pCard, sizeof(WD_CARD_REGISTER), \
    FALSE)
```

Card registration function.

The function: Maps the physical memory ranges to be accessed by kernel-mode processes and user-mode applications.

Verifies that none of the registered device resources (set in pCardReg->Card.Item) are already locked for exclusive use. A resource can be locked for exclusive use by setting the fNotSharable field of its [WD_ITEMS](#) structure (pCardReg->Card.Item[i]) to 1, before calling [WD_CardRegister\(\)](#).

Saves data regarding the interrupt request (IRQ) number and the interrupt type in internal data structures; this data will later be used by [InterruptEnable\(\)](#) and/or [WD_IntEnable\(\)](#)

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in, out	<i>pCard</i>	Pointer to a card registration information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

1. For PCI the cardReg.Card input resources information should be retrieved from the Plug-and-Play Manager via [WD_PciGetCardInfo\(\)](#)

2. If your card has a large memory range that cannot be fully mapped to the kernel address space, you can set the WD_ITEM_MEM_DO_NOT_MAP_KERNEL flag in the I.Mem.dwOptions field of the relevant [WD_ITEMS](#) memory resource structure that you pass to the card registration function (pCardReg->Card.Item[i].I.Mem.dwOptions). This flag instructs the function to map the memory range only to the user-mode virtual address space, and not to the kernel address space. (For PCI devices, you can modify the relevant item in the card information structure (pCard) that you received from [WD_PciGetCardInfo\(\)](#) before passing this structure to [WD_CardRegister\(\)](#).)

Note that if you select to set the WD_ITEM_MEM_DO_NOT_MAP_KERNEL flag, [WD_CardRegister\(\)](#) will not update the item's pTransAddr field with a kernel mapping of the memory's base address, and you will therefore not be able to rely on this mapping in calls to WinDriver APIs namely interrupt handling APIs or any API called from a Kernel PlugIn driver

3. [WD_CardRegister\(\)](#) enables the user to map the card memory resources into virtual memory and access them as regular pointers.

```
WD_CARD_REGISTER cardReg;
BZERO(cardReg);
cardReg.Card.dwItems = 1;
cardReg.Card.Item[0].item = ITEM_IO;
cardReg.Card.Item[0].fNotSharable = 1;
cardReg.Card.Item[0].I.IO.pAddr = 0x378;
cardReg.Card.Item[0].I.IO.dwBytes = 8;
WD_CardRegister(hWD, &cardReg);
if (cardReg.hCard == 0)
{
    printf("Failed locking device\n");
    return FALSE;
}
```

Definition at line 2191 of file [windrvr.h](#).

WD_CardUnregister

```
#define WD_CardUnregister(
    h,
    pCard )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_CARD_UNREGISTER, h, pCard, sizeof(WD_CARD_REGISTER), \
        FALSE)
```

Unregisters a device and frees the resources allocated to it.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pCard</i>	Pointer to a card registration information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_CardUnregister(hWD, &cardReg);
```

Definition at line 2207 of file [windrvr.h](#).

WD_Close

```
#define WD_Close WD_CloseLocal
```

Definition at line 1813 of file [windrvr.h](#).

WD_CloseLocal

```
#define WD_CloseLocal(
    h ) CloseHandle(h)
```

Definition at line 1762 of file [windrvr.h](#).

WD_CPU_SPEC

```
#define WD_CPU_SPEC "X86"
```

Definition at line 131 of file [windrvr.h](#).

WD_CTL_CODE

```
#define WD_CTL_CODE (
    wFuncNum )
```

Value:

```
    CTL_CODE(WD_TYPE, (wFuncNum | FUNC_MASK), \
```

```
METHOD_NEITHER, FILE_ANY_ACCESS)
```

Definition at line 1568 of file [windrvr.h](#).

WD_CTL_DECODE_FUNC

```
#define WD_CTL_DECODE_FUNC(
```

$$\text{IoControlCode} \text{ } ((\text{IoControlCode} >> 2) \& 0xffff)$$

Definition at line 1570 of file [windrvr.h](#).

WD_CTL_DECODE_TYPE

```
#define WD_CTL_DECODE_TYPE(
```

$$\text{IoControlCode} \text{ } \text{DEVICE_TYPE_FROM_CTL_CODE}(\text{IoControlCode})$$

Definition at line 1571 of file [windrvr.h](#).

WD_CTL_IS_64BIT_AWARE

```
#define WD_CTL_IS_64BIT_AWARE(
```

$$\text{IoControlCode} \text{ } (\text{WD_CTL_DECODE_FUNC}(\text{IoControlCode}) \& \text{FUNC_MASK})$$

Definition at line 1581 of file [windrvr.h](#).

WD_DATA_MODEL

```
#define WD_DATA_MODEL "32bit"
```

Definition at line 145 of file [windrvr.h](#).

WD_Debug

```
#define WD_Debug(
```

$$\text{h},$$

```
\text{pDebug} \text{ } \text{WD\_FUNCTION}(\text{IOCTL_WD_DEBUG}, \text{h}, \text{pDebug}, \text{sizeof(WD_DEBUG)}, \text{FALSE})

Sets debugging level for collecting debug messages.


```

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDebug</i>	Pointer to a debug information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_DEBUG dbg;
```

$$\text{BZERO}(\text{dbg});$$

```
dbg.dwCmd = DEBUG_SET_FILTER;
```

$$\text{dbg.dwLevel} = \text{D_ERROR};$$

```
dbg.dwSection = S_ALL;
```

$$\text{dbg.dwLevelMessageBox} = \text{D_ERROR};$$

```
WD_Debug(hWD, &dbg);
```

Definition at line 1837 of file [windrvr.h](#).

WD_DebugAdd

```
#define WD_DebugAdd(
```

$$\text{h},$$

```
pDebugAdd )  WD_FUNCTION(IOCTL_WD_DEBUG_ADD, h, pDebugAdd, sizeof(WD_DEBUG_ADD), \
FALSE)
```

Sends debug messages to the debug log.

Used by the driver code.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDebugAdd</i>	Pointer to an additional debug information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
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);
```

Definition at line 1866 of file [windrvr.h](#).

WD_DebugDump

```
#define WD_DebugDump (
    h,
    pDebugDump )
```

Value:

```
WD_FUNCTION(IOCTL_WD_DEBUG_DUMP, h, pDebugDump, sizeof(WD_DEBUG_DUMP), \
FALSE)
```

Retrieves debug messages buffer.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDebugDump</i>	Pointer to a debug information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
char buffer[1024];
WD_DEBUG_DUMP dump;
dump.pcBuffer=buffer;
dump.dwSize = sizeof(buffer);
WD_DebugDump(hWD, &dump);
```

Definition at line 1851 of file [windrvr.h](#).

WD_DEFAULT_DRIVER_NAME

```
#define WD_DEFAULT_DRIVER_NAME WD_DRIVER_NAME_PREFIX WD_DEFAULT_DRIVER_NAME_BASE
```

Definition at line 56 of file [windrvr.h](#).

WD_DEFAULT_DRIVER_NAME_BASE

```
#define WD_DEFAULT_DRIVER_NAME_BASE "windrvr" WD_VER_ITOA
```

Definition at line 55 of file [windrvr.h](#).

WD_DMALock

```
#define WD_DMALock(
    h,
    pDma )  WD_FUNCTION(IOCTL_WD_DMA_LOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
```

Enables contiguous-buffer or Scatter/Gather DMA.

For contiguous-buffer DMA, the function allocates a DMA buffer and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.

For Scatter/Gather DMA, the function receives the address of a data buffer allocated in the usermode, locks it for DMA, and returns the corresponding physical mappings of the locked DMA pages. On Windows the function also returns a kernel-mode mapping of the buffer.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in, out	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

WinDriver supports both Scatter/Gather and contiguous-buffer DMA on Windows and Linux. 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).

You should NOT use the physical memory address returned by the function (dma.Page[i].pPhysicalAddr) directly in order to access the DMA buffer from your driver. To access the memory directly from a user-mode process, use the user-mode virtual mapping of the DMA buffer dma.pUserAddr. To access the memory in the kernel, either directly from within a Kernel PlugIn driver (see the WinDriver PCI Manual) or when calling [WD_Transfer\(\)](#) / [WD_MultiTransfer\(\)](#), use the kernel mapping of the DMA buffer. For contiguous-buffer DMA (dma.dwOptions | DMA_KERNEL_BUFFER_ALLOC) and for Scatter/Gather DMA on Windows, this mapping is returned by [WD_DMALock\(\)](#) within the dma.pKernelAddr field. For Scatter/Gather DMA on other platforms, you can acquire a kernel mapping of the buffer by calling [WD_CardRegister\(\)](#) with a card structure that contains a memory item defined with the physical DMA buffer address returned from [WD_DMALock\(\)](#) (dma.Page[i].pPhysicalAddr). [WD_CardRegister\(\)](#) will return a kernel mapping of the physical buffer within the pCardReg->Card.Item[i].lMem.pTransAddr field.

On Windows x86 and x86_64 platforms, you should normally set the DMA_ALLOW_CACHE flag in the DMA options bitmask parameter (pDma->dwOptions).

If the device supports 64-bit DMA addresses, it is recommended to set the DMA_ALLOW_64BIT_ADDRESS flag in pDma->dwOptions. Otherwise, when the physical memory on the target platform is larger than 4GB, the operating system may only allow allocation of relatively small 32-bit DMA buffers (such as 1MB buffers, or even smaller).

When using the DMA_LARGE_BUFFER flag, dwPages is an input/output parameter. As input to [WD_DMALock\(\)](#), dwPages should be set to the maximum number of pages that can be used for the DMA buffer (normally this would be the number of elements in the dma.Page array). As an output value of [WD_DMALock\(\)](#), dwPages holds the number of actual physical blocks allocated for the DMA buffer. The returned dwPages may be smaller than the input value because adjacent pages are returned as one block.

```
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_FROM_DEVICE : DMA_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. */
}

*****
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_FROM_DEVICE : DMA_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. */
}

```

Definition at line 1998 of file [windrvr.h](#).

WD_DMASyncCpu

```
#define WD_DMASyncCpu(
    h,
    pDma )  WD_FUNCTION(IOCTL_WD_DMA_SYNC_CPU, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
```

Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WD_DMALock()

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

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 upto-date than the copy in memory.

```
WD_DMASyncCpu(hWD, &dma);
```

Definition at line 2113 of file [windrvr.h](#).

WD_DMASyncIo

```
#define WD_DMASyncIo(
    h,
    pDma )  WD_FUNCTION(IOCTL_WD_DMA_SYNC_IO, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
```

Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pDma</i>	Pointer to a DMA information structure, received from a previous call to WD_DMALock()

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

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.

`WD_DMASyncIo(hWD, &dma);`

Definition at line 2140 of file [windrvr.h](#).

WD_DMATransactionExecute

```
#define WD_DMATransactionExecute(
    h,
    pDma )
```

Value:

```
WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_EXECUTE, h, pDma, \
    SIZE_OF_WD_DMA(pDma), FALSE)
```

Begins the execution of a specified DMA transaction.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2043 of file [windrvr.h](#).

WD_DMATransactionInit

```
#define WD_DMATransactionInit(
    h,
    pDma )
```

Value:

```
WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_INIT, h, pDma, SIZE_OF_WD_DMA(pDma), \
    FALSE)
```

Initializes the transaction, allocates a DMA buffer, locks it in physical memory, and returns mappings of the allocated buffer to physical address space and to user-mode and kernel virtual address spaces.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in, out	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2029 of file [windrvr.h](#).

WD_DMATransactionRelease

```
#define WD_DMATransactionRelease( h, pDma )  
    WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_RELEASE, h, pDma, \  
    SIZE_OF_WD_DMA(pDma), FALSE)
```

Terminates a specified DMA transaction without deleting the associated [WD_DMA](#) transaction structure.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2072 of file [windrvr.h](#).

WD_DMATransactionUninit

```
#define WD_DMATransactionUninit( h, pDma )  WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
```

Unlocks and frees the memory allocated for a DMA buffer transaction by a previous call to [WD_DMATransactionInit\(\)](#)

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2087 of file [windrvr.h](#).

WD_DMATransferCompletedAndCheck

```
#define WD_DMATransferCompletedAndCheck( h, pDma )  
    WD_FUNCTION(IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK, h, pDma, \  
    SIZE_OF_WD_DMA(pDma), FALSE)
```

Notifies WinDriver that a device's DMA transfer operation is completed.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2057 of file [windrvr.h](#).

WD_DMAUnlock

```
#define WD_DMAUnlock( h, pDma ) WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
```

Unlocks a DMA buffer.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pDma</i>	Pointer to a DMA information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

[WD_DMAUnlock](#) (hWD, &dma);

Definition at line 2013 of file [windrvr.h](#).

WD_DRIVER_NAME

```
#define WD_DRIVER_NAME WD_DriverName(NULL)
```

Get driver name.

Definition at line 105 of file [windrvr.h](#).

WD_DRIVER_NAME_PREFIX

```
#define WD_DRIVER_NAME_PREFIX ""
```

Definition at line 48 of file [windrvr.h](#).

WD_EventPull

```
#define WD_EventPull( h, pEvent ) WD_FUNCTION(IOCTL_WD_EVENT_PULL, h, pEvent, SIZE_OF_WD_EVENT(pEvent), FALSE)
```

Definition at line 2661 of file [windrvr.h](#).

WD_EventRegister

```
#define WD_EventRegister( h, pEvent )
```

Value:

```
WD_FUNCTION(IOCTL_WD_EVENT_REGISTER, h, pEvent, SIZE_OF_WD_EVENT(pEvent), \
```

FALSE)

Definition at line 2655 of file [windrvr.h](#).

WD_EventSend

```
#define WD_EventSend(
    h,
    pEvent )  WD_FUNCTION(IOCTL_WD_EVENT_SEND, h, pEvent, SIZE_OF_WD_EVENT(pEvent),
FALSE)
```

Definition at line 2663 of file [windrvr.h](#).

WD_EventUnregister

```
#define WD_EventUnregister(
    h,
    pEvent )

Value:
    WD_FUNCTION(IOCTL_WD_EVENT_UNREGISTER, h, pEvent, \
    SIZE_OF_WD_EVENT(pEvent), FALSE)
```

Definition at line 2658 of file [windrvr.h](#).

WD_FUNCTION

```
#define WD_FUNCTION WD_FUNCTION_LOCAL
Definition at line 1812 of file windrvr.h.
```

WD_GetDeviceProperty

```
#define WD_GetDeviceProperty(
    h,
    pGetDevProperty )

Value:
    WD_FUNCTION(IOCTL_WD_GET_DEVICE_PROPERTY, h, pGetDevProperty, \
    sizeof(WD_GET_DEVICE_PROPERTY), FALSE);
```

Definition at line 2671 of file [windrvr.h](#).

WD_IntCount

```
#define WD_IntCount(
    h,
    pInterrupt )  WD_FUNCTION(IOCTL_WD_INT_COUNT, h, pInterrupt, sizeof(WD_INTERRUPT),
FALSE)
```

Retrieves the interrupts count since the call to [WD_IntEnable\(\)](#)

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in, out	<i>pInterrupt</i>	Pointer to an interrupt information structure

Returns

Returns [WD_STATUS_SUCCESS](#) (0) on success, or an appropriate error code otherwise

```
DWORD dwNumInterrupts;
WD_IntCount(hWD, &intrp);
dwNumInterrupts = intrp.dwCounter;
```

Definition at line 2615 of file [windrvr.h](#).

WD_IntDisable

```
#define WD_IntDisable(
```

```
    h,
    pInterrupt )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_INT_DISABLE, h, pInterrupt, sizeof(WD_INTERRUPT), \
    FALSE)
```

Disables interrupt processing.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pInterrupt</i>	Pointer to an interrupt information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_IntDisable(hWD, &intrp);
```

Definition at line 2600 of file [windrvr.h](#).

WD_IntEnable

```
#define WD_IntEnable(
    h,
    pInterrupt )  WD_FUNCTION(IOCTL_WD_INT_ENABLE, h, pInterrupt, sizeof(WD_INTERRUPT),
FALSE)
```

Registers an interrupt service routine (ISR) to be called upon interrupt.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in, out	<i>pInterrupt</i>	Pointer to an interrupt information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

For more information regarding interrupt handling please refer to the Interrupts section in the WinDriver PCI Manual.

KpCall is relevant for Kernel Plugin implementation.

WinDriver must be registered with the OS as the driver of the device before enabling interrupts. For Plug-and-Play hardware (PCI/PCI Express) on Windows platforms, this association is made by installing an INF file for the device. If the INF file is not installed, [WD_IntEnable\(\)](#) will fail with a WD_NO_DEVICE_OBJECT error.

```
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 = 1;
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");
}
```

Definition at line 2586 of file [windrvr.h](#).

WD_IntWait

```
#define WD_IntWait(
    h,
    pInterrupt )  WD_FUNCTION(IOCTL_WD_INT_WAIT, h, pInterrupt, sizeof(WD_INTERRUPT),
TRUE)
```

Waits for an interrupt.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in,out	<i>pInterrupt</i>	Pointer to an interrupt information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

The INTERRUPT_INTERRUPTED status (set in pInterrupt->fStopped) can occur on Linux if the application that waits on the interrupt is stopped (e.g., by pressing CTRL+Z).

```
for (;;)
{
    WD_IntWait(hWD, &intrp);
    if (intrp.fStopped)
        break;
    ProcessInterrupt(intrp.dwCounter);
}
```

Definition at line 2634 of file [windrvr.h](#).

WD_IPC_ALL_MSG

```
#define WD_IPC_ALL_MSG (WD_IPC_UNICAST_MSG | WD_IPC_MULTICAST_MSG)
```

Definition at line 1454 of file [windrvr.h](#).

WD_IpcRegister

```
#define WD_IpcRegister(
    h,
    pIpcRegister )
```

Value:

```
WD_FUNCTION(IOCTL_WD_IPC_REGISTER, h, pIpcRegister, \
sizeof(WD_IPC_REGISTER), FALSE)
```

Registers an application with WinDriver IPC.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in,out	<i>pIpcRegister</i>	Pointer to the IPC information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

You should choose your user applications a unique group ID parameter. This is done as a precaution to prevent several applications that use WinDriver with its default driver name (windrvrXXXX) to get mixing messages. We strongly recommend that you rename your driver before distributing it to avoid this issue entirely, among other issue (See Section 15.2 on renaming your driver name). The sub-group id parameter should identify your user application type in case you have several types that may work simultaneously.

Definition at line 2232 of file [windrvr.h](#).

WD_IpcScanProcs

```
#define WD_IpcScanProcs( h,  
                        pIpcScanProcs )  
WD_FUNCTION(IOCTL_WD_IPC_SCAN_PROCS, h, pIpcScanProcs, \  
           sizeof(WD_IPC_SCAN_PROCS), FALSE)
```

Value:

```
WD_FUNCTION(IOCTL_WD_IPC_SCAN_PROCS, h, pIpcScanProcs, \  
           sizeof(WD_IPC_SCAN_PROCS), FALSE)
```

Scans and returns information of all WinDriver IPC registered processes that share the application process groupID.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>plpcScanProcs</i>	Pointer to the processes info struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2265 of file [windrvr.h](#).

WD_IpcSend

```
#define WD_IpcSend( h,  
                  pIpcSend )  WD_FUNCTION(IOCTL_WD_IPC_SEND, h, pIpcSend, sizeof(WD_IPC_SEND),  
                  FALSE)
```

Sends a message to other processes, depending on the content of the plpcSend struct.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>plpcSend</i>	Pointer to the IPC message info struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2282 of file [windrvr.h](#).

WD_IpcUnRegister

```
#define WD_IpcUnRegister(
```

```
    h,
    pProcInfo )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_IPC_UNREGISTER, h, pProcInfo, sizeof(WD_IPC_PROCESS), \
        FALSE)
```

This function unregisters the user application from WinDriver IPC.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pProcInfo</i>	Pointer to the process information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2248 of file [windrvr.h](#).

WD_KernelBufLock

```
#define WD_KernelBufLock(
    h,
    pKerBuf )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_KERNEL_BUF_LOCK, h, pKerBuf, \
        sizeof(WD_KERNEL_BUFFER), FALSE)
```

Allocates a contiguous or non-contiguous non-paged kernel buffer, and maps it to user address space. This buffer should be used ONLY for shared buffer purposes (The buffer should NOT be used for DMA).

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in,out	<i>pKerBuf</i>	Pointer to a WD_KERNEL_BUFFER information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_KERNEL_BUFFER buf;
DWORD dwStatus;
BZERO(buf);
buf.qwBytes = 200000;
buf.dwOptions = ALLOCATE_CONTIG_BUFFER | ALLOCATE_CACHED_BUFFER;
dwStatus = WD_KernelBufLock(hWD, &buf);
if (dwStatus)
{
    printf("Could not lock down buffer\n");
}
else
{
    /* To access the memory from your user mode
       application, use buf.pUserAddr. */
}
```

Definition at line 1917 of file [windrvr.h](#).

WD_KernelBufUnlock

```
#define WD_KernelBufUnlock(
    h,
    pKerBuf )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_KERNEL_BUF_UNLOCK, h, pKerBuf, \
        sizeof(WD_KERNEL_BUFFER), FALSE)
```

Frees kernel buffer.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pKerBuf</i>	Pointer to a WD_KERNEL_BUFFER information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_KernelBufUnlock(hWD, &buf);
```

Definition at line 1932 of file [windrvr.h](#).

WD_KernelPlugInCall

```
#define WD_KernelPlugInCall(
    h,
    pKernelPlugInCall )
```

Value:

```
WD_FUNCTION(IOCTL_WD_KERNEL_PLUGIN_CALL, h, pKernelPlugInCall, \
    sizeof(WD_KERNEL_PLUGIN_CALL), FALSE)
```

Calls a routine in the Kernel PlugIn to be executed.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in,out	<i>pKernelPlugInCall</i>	Pointer to Kernel PlugIn information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

Calling the [WD_KernelPlugInCall\(\)](#) function in the user mode will call your KP_Call callback function in the kernel. The KP_Call Kernel PlugIn function will determine what routine to execute according to the message passed to it in the [WD_KERNEL_PLUGIN_CALL](#) structure.

```
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));
}
```

Definition at line 2559 of file [windrvr.h](#).

WD_KernelPlugInClose

```
#define WD_KernelPlugInClose(
    h,
    pKernelPlugIn )
```

Value:

```
WD_FUNCTION( IOCTL_WD_KERNEL_PLUGIN_CLOSE, h, pKernelPlugIn, \
    sizeof(WD_KERNEL_PLUGIN), FALSE)
```

Closes the WinDriver Kernel PlugIn handle obtained from [WD_KernelPlugInOpen\(\)](#)

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pKernelPlugin</i>	Pointer to Kernel PlugIn information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_KernelPlugInClose(hWD, &kernelPlugIn);
```

Definition at line 2536 of file [windrvr.h](#).

WD_KernelPlugInOpen

```
#define WD_KernelPlugInOpen(
    h,
    pKernelPlugin )
```

Value:

```
WD_FUNCTION(IOCTL_WD_KERNEL_PLUGIN_OPEN, h, pKernelPlugin, \
    sizeof(WD_KERNEL_PLUGIN), FALSE)
```

Obtain a valid handle to the Kernel PlugIn.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in, out	<i>pKernelPlugin</i>	Pointer to Kernel PlugIn information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_KERNEL_PLUGIN kernelPlugIn;
HANDLE hWD;
hWD = WD_Open();
if (hWD == INVALID_HANDLE_VALUE)
{
    goto Error; /* Handle error */
}
BZERO(kernelPlugIn);
/* Tells WinDriver which driver to open */
kernelPlugIn.pcDriverName = "KDDriver";
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);
}
```

Definition at line 2520 of file [windrvr.h](#).

WD_License

```
#define WD_License(
    h,
    pLicense )  WD_FUNCTION(IOCTL_WD_LICENSE, h, pLicense, sizeof(WD_LICENSE), FALSE)
```

Transfers the license string to the WinDriver kernel module When using the high-level WDC library APIs, described in the WinDriver PCI Manual, the license registration is done via the [WDC_DriverOpen\(\)](#) function, so you do not need to call [WD_License\(\)](#) directly.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pLicense</i>	Pointer to a WinDriver license information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

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.

```
/* Use the returned handle when calling WinDriver API functions */
HANDLE WinDriverOpenAndRegister(void)
{
    HANDLE hWD;
    WD_LICENSE lic;
    DWORD dwStatus;
    hWD = WD_Open();
    if (hWD != INVALID_HANDLE_VALUE)
    {
        BZERO(lic);
        /* Replace the following string with your license string: */
        strcpy(lic.license, "12345abcde12345.CompanyName");
        dwStatus = WD_License(hWD, &lic);
        if (dwStatus != WD_STATUS_SUCCESS)
        {
            WD_Close(hWD);
            hWD = INVALID_HANDLE_VALUE;
        }
    }
    return hWD;
}
```

Definition at line 2504 of file [windrvr.h](#).

WD_LICENSE_LENGTH

```
#define WD_LICENSE_LENGTH 3072
```

Definition at line 647 of file [windrvr.h](#).

WD_MAX_DRIVER_NAME_LENGTH

```
#define WD_MAX_DRIVER_NAME_LENGTH 128
```

Definition at line 59 of file [windrvr.h](#).

WD_MAX_KP_NAME_LENGTH

```
#define WD_MAX_KP_NAME_LENGTH 128
```

Definition at line 60 of file [windrvr.h](#).

WD_MultiTransfer

```
#define WD_MultiTransfer(
    h,
    pTransferArray,
    dwNumTransfers )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_MULTI_TRANSFER, h, pTransferArray, \
        sizeof(WD_TRANSFER) * (dwNumTransfers), FALSE)
```

Executes multiple read/write instructions to I/O ports and/or memory addresses.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pTransferArray</i>	Pointer to a beginning of an array of transfer information structures
in,out	<i>dwNumTransfers</i>	The number of transfers to perform (the <i>pTransferArray</i> array should contain at least <i>dwNumTransfers</i> elements)

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_TRANSFER Trans[4];
DWORD dwResult;
char *cData = "Message to send\n";
BZERO(Trans);
Trans[0].cmdTrans = WP_WORD; /* Write Port WORD */
Trans[0].pPort = 0x1e0;
Trans[0].Data.Word = 0x1023;
Trans[1].cmdTrans = WP_WORD;
Trans[1].pPort = 0x1e0;
Trans[1].Data.Word = 0x1022;
Trans[2].cmdTrans = WP_SBYTE; /* Write Port String BYTE */
Trans[2].pPort = 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].pPort = 0x1e4;
WD_MultiTransfer(hWD, &Trans, 4);
dwResult = Trans[3].Data.Dword;
```

Definition at line 1900 of file [windrvr.h](#).

WD_Open

```
#define WD_Open WD_OpenLocal
```

Definition at line 1814 of file [windrvr.h](#).

WD_OpenLocal

```
#define WD_OpenLocal( )
```

Value:

```
CreateFileA(\n    WD_DRIVER_NAME,\n    GENERIC_READ,\n    FILE_SHARE_READ | FILE_SHARE_WRITE,\n    NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL)
```

Definition at line 1777 of file [windrvr.h](#).

WD_OpenStreamLocal

```
#define WD_OpenStreamLocal (\n    read,\n    sync )
```

Value:

```
CreateFileA(\n    WD_DRIVER_NAME,\n    (read) ? GENERIC_READ : GENERIC_WRITE,\n    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, \\\n    (sync) ? 0 : FILE_FLAG_OVERLAPPED, NULL)
```

Definition at line 1784 of file [windrvr.h](#).

WD_PciConfigDump

```
#define WD_PciConfigDump(
    h,
    pPciConfigDump )
```

Value:

```
WD_FUNCTION(IOCTL_WD_PCI_CONFIG_DUMP, h, pPciConfigDump, \
    sizeof(WD_PCI_CONFIG_DUMP), FALSE)
```

Reads/writes from/to the PCI configuration space of a selected PCI card or the extended configuration space of a selected PCI Express card.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in, out	<i>pPciConfigDump</i>	Pointer to a PCI configuration space information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_PCI_CONFIG_DUMP pciConfig;
DWORD dwStatus;
WORD aBuffer[2];
BZERO(pciConfig);
pciConfig.pciSlot.dwDomain = 0;
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 Domain 0, Bus 0, Slot 3, Funcion 0 has Vendor ID %x "
        "Device ID %x\n", aBuffer[0], aBuffer[1]);
}
```

Definition at line 2465 of file [windrvr.h](#).

WD_PciGetCardInfo

```
#define WD_PciGetCardInfo(
    h,
    pPciCard )
```

Value:

```
WD_FUNCTION(IOCTL_WD_PCI_GET_CARD_INFO, h, pPciCard, \
    sizeof(WD_PCI_CARD_INFO), FALSE)
```

Retrieves PCI device's resource information (i.e., Memory ranges, I/O ranges, Interrupt lines).

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in, out	<i>pPciCard</i>	Pointer to a PCI card information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
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");
}

```

Definition at line 2447 of file [windrvr.h](#).

WD_PciScanCaps

```
#define WD_PciScanCaps (
    h,
    pPciScanCaps )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_PCI_SCAN_CAPS, h, pPciScanCaps, \
        sizeof(WD_PCI_SCAN_CAPS), FALSE)
```

Scans the specified PCI capabilities group of the given PCI slot for the specified capability (or for all capabilities).

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in,out	<i>pPciScanCaps</i>	Pointer to a PCI capabilities scan structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```

WD_PCI_SCAN_CAPS pciScanCaps;
WD_PCI_CAP pciCap;
BZERO(pciScanCaps);
pciScanCaps.pciSlot = pciSlot; /* pciSlot = a WD_PCI_SLOT struct returned by
                               WD_PciScanCards() in pPciScan->cardSlot */
pciScanCaps.dwCapId = 0x05; /* Search for the MSI capability */
pciScanCaps.dwOptions = WD_PCI_SCAN_CAPS_BASIC; /* Scan the basic PCI
                                               capabilities */
WD_PciScanCaps(hWD, &pciScanCaps);
if (pciScanCaps.dwNumCaps > 0) /* Found a matching capability */
{
    pciCap = pciScanCaps.pciCaps[0]; /* Use the first capability found */
}
else
{
    printf("PCI capability 0x%lx not found in the basic PCI capabilities\n",
    pciScanCaps.dwCapId);
}

```

Definition at line 2430 of file [windrvr.h](#).

WD_PciScanCards

```
#define WD_PciScanCards (
    h,
    pPciScan )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_PCI_SCAN_CARDS, h, pPciScan, \
        sizeof(WD_PCI_SCAN_CARDS), FALSE)
```

Detects PCI devices installed on the PCI bus, which conform to the input criteria (vendor ID and/or card ID), and returns the number and location (bus, slot and function) of the detected devices.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in,out	<i>pPciScan</i>	Pointer to a PCI bus scan information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_PCI_SCAN_CARDS pciScan;
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");
Definition at line 2413 of file windrvr.h.
```

WD_PciSriovDisable

```
#define WD_PciSriovDisable(
    h,
    pPciSRIOV )
```

Value:

```
WD_FUNCTION(IOCTL_WD_PCI_SRIOV_DISABLE, h, pPciSRIOV, \
            sizeof(WD_PCI_SRIOV), FALSE)
```

Disables SR-IOV for a supported device and removes all the assigned VFs.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pPciSRIOV</i>	Pointer to SR-IOV information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

Only supported in Linux

Definition at line 2350 of file [windrvr.h](#).

WD_PciSriovEnable

```
#define WD_PciSriovEnable(
    h,
    pPciSRIOV )
```

Value:

```
WD_FUNCTION(IOCTL_WD_PCI_SRIOV_ENABLE, h, pPciSRIOV, \
            sizeof(WD_PCI_SRIOV), FALSE)
```

Enables SR-IOV for a supported device.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in	<i>pPciSRIOV</i>	Pointer to SR-IOV information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

Only supported in Linux

Definition at line 2333 of file [windrvr.h](#).

WD_PciSriovGetNumVFs

```
#define WD_PciSriovGetNumVFs ( \
    h, \
    pPciSRIOV )
```

Value:
`WD_FUNCTION(IOCTL_WD_PCI_SRIOV_GET_NUMVFS, h, pPciSRIOV, \
 sizeof(WD_PCI_SRIOV), FALSE)`

Gets the number of virtual functions assigned to a supported device.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in,out	<i>pPciSRIOV</i>	Pointer to SR-IOV information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

The number of virtual functions will be stored in the pciSlot->dwNumVFs field. Only supported in Linux.

Definition at line 2369 of file [windrvr.h](#).

WD_PROCESS_NAME_LENGTH

```
#define WD_PROCESS_NAME_LENGTH 128
```

Definition at line 763 of file [windrvr.h](#).

WD_PROD_NAME

```
#define WD_PROD_NAME "WinDriver"
```

Definition at line 110 of file [windrvr.h](#).

WD_SharedIntDisable

```
#define WD_SharedIntDisable( \
    h )  WD_FUNCTION(IOCTL_WD_IPC_SHARED_INT_DISABLE, h, 0, 0, FALSE)
```

Disables the Shared Interrupts mechanism of WinDriver for all processes.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
----	----------	---

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Definition at line 2317 of file [windrvr.h](#).

WD_SharedIntEnable

```
#define WD_SharedIntEnable(
    h,
    pIpcRegister )
```

Value:

```
    WD_FUNCTION(IOCTL_WD_IPC_SHARED_INT_ENABLE, h, pIpcRegister, \
        sizeof(WD_IPC_REGISTER), FALSE)
```

Enables the shared interrupts mechanism of WinDriver.

If the mechanism is already enabled globally (for all processes) then the mechanism is also enabled for the current process.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pIpcRegister</i>	Pointer to the IPC information struct

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

This function is currently only supported from the user mode. This function is supported only for Windows and Linux.

Definition at line 2302 of file [windrvr.h](#).

WD_Sleep

```
#define WD_Sleep(
    h,
    pSleep )  WD_FUNCTION(IOCTL_WD_SLEEP, h, pSleep, sizeof(WD_SLEEP), FALSE)
```

Delays execution for a specific duration of time.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
in	<i>pSleep</i>	Pointer to a sleep information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

Remarks

Example usage: to access slow response hardware.

```
WD_Sleep slp;
BZERO(slp);
slp.dwMicroSeconds = 200;
WD_Sleep(hWD, &slp)
```

Definition at line 2651 of file [windrvr.h](#).

WD_StreamClose

```
#define WD_StreamClose WD_CloseLocal
```

Definition at line 1816 of file [windrvr.h](#).

WD_StreamOpen

```
#define WD_StreamOpen WD_OpenStreamLocal
Definition at line 1815 of file windrvr.h.
```

WD_Transfer

```
#define WD_Transfer(
    h,
    pTransfer )  WD_FUNCTION(IOCTL_WD_TRANSFER, h, pTransfer, sizeof(WD_TRANSFER),
FALSE)
```

Executes a single read/write instruction to an I/O port or to a memory address.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open() .
in,out	<i>pTransfer</i>	Pointer to a transfer information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
WD_TRANSFER Trans;
BYTE read_data;
BZERO(Trans);
Trans.cmdTrans = RP_BYTE; /* Read Port BYTE */
Trans.pPort = 0x210;
WD_Transfer(hWD, &Trans);
read_data = Trans.Data.Byte;
```

Definition at line 1881 of file [windrvr.h](#).

WD_TYPE

```
#define WD_TYPE 38200
Device type.
Definition at line 1562 of file windrvr.h.
```

WD_UGetDeviceData

```
#define WD_UGetDeviceData(
    h,
    pGetDevData )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_GET_DEVICE_DATA, h, pGetDevData, \
        sizeof(WDU_GET_DEVICE_DATA), FALSE);
```

Definition at line 2668 of file [windrvr.h](#).

WD_UHaltTransfer

```
#define WD_UHaltTransfer(
    h,
    pHaltTrans )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_HALT_TRANSFER, h, pHaltTrans, \
        sizeof(WDU_HALT_TRANSFER), FALSE);
```

Definition at line 2682 of file [windrvr.h](#).

WD_UResetDevice

```
#define WD_UResetDevice( h,  
                        pResetDevice )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_RESET_DEVICE, h, pResetDevice, \  
               sizeof(WDU_RESET_DEVICE), FALSE);
```

Definition at line 2690 of file [windrvr.h](#).

WD_UResetPipe

```
#define WD_UResetPipe( h,  
                      pResetPipe )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_RESET_PIPE, h, pResetPipe, sizeof(WDU_RESET_PIPE), \  
               FALSE);
```

Definition at line 2677 of file [windrvr.h](#).

WD_Usage

```
#define WD_Usage( h,  
                pStop )  WD_FUNCTION(IOCTL_WD_USAGE, h, pStop, sizeof(WD_USAGE), FALSE)
```

Definition at line 2665 of file [windrvr.h](#).

WD_USelectiveSuspend

```
#define WD_USelectiveSuspend( h,  
                            pSelectiveSuspend )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_SELECTIVE_SUSPEND, h, pSelectiveSuspend, \  
               sizeof(WDU_SELECTIVE_SUSPEND), FALSE);
```

Definition at line 2687 of file [windrvr.h](#).

WD_USetInterface

```
#define WD_USetInterface( h,  
                        pSetIfc )
```

Value:

```
    WD_FUNCTION(IOCTL_WDU_SET_INTERFACE, h, pSetIfc, \  
               sizeof(WDU_SET_INTERFACE), FALSE);
```

Definition at line 2674 of file [windrvr.h](#).

WD_UStreamClose

```
#define WD_UStreamClose( h,  
                        pStream )  WD_FUNCTION(IOCTL_WDU_STREAM_CLOSE, h, pStream, sizeof(WDU_STREAM),  
                           FALSE);
```

Definition at line 2695 of file [windrvr.h](#).

WD_UStreamFlush

```
#define WD_UStreamFlush(
    h,
    pStream )  WD_FUNCTION(IOCTL_WDU_STREAM_FLUSH, h, pStream, sizeof(WDU_STREAM),
FALSE);
```

Definition at line 2701 of file [windrvr.h](#).

WD_UStreamGetStatus

```
#define WD_UStreamGetStatus(
    h,
    pStreamStatus )

Value:
    WD_FUNCTION(IOCTL_WDU_STREAM_GET_STATUS, h, pStreamStatus, \
    sizeof(WDU_STREAM_STATUS), FALSE);
```

Definition at line 2703 of file [windrvr.h](#).

WD_UStreamOpen

```
#define WD_UStreamOpen(
    h,
    pStream )  WD_FUNCTION(IOCTL_WDU_STREAM_OPEN, h, pStream, sizeof(WDU_STREAM),
FALSE);
```

Definition at line 2693 of file [windrvr.h](#).

WD_UStreamRead

```
#define WD_UStreamRead(
    hFile,
    pBuffer,
    dwNumberOfBytesToRead,
    dwNumberOfBytesRead )

Value:
    ReadFile(hFile, pBuffer, dwNumberOfBytesToRead, \
    dwNumberOfBytesRead, NULL) ? WD_STATUS_SUCCESS : \
    WD_OPERATION_FAILED
```

Definition at line 1764 of file [windrvr.h](#).

WD_UStreamStart

```
#define WD_UStreamStart(
    h,
    pStream )  WD_FUNCTION(IOCTL_WDU_STREAM_START, h, pStream, sizeof(WDU_STREAM),
FALSE);
```

Definition at line 2697 of file [windrvr.h](#).

WD_UStreamStop

```
#define WD_UStreamStop(
    h,
    pStream )  WD_FUNCTION(IOCTL_WDU_STREAM_STOP, h, pStream, sizeof(WDU_STREAM),
FALSE);
```

Definition at line 2699 of file [windrvr.h](#).

WD_UStreamWrite

```
#define WD_UStreamWrite(
    hFile,
    pBuffer,
    dwNumberOfBytesToWrite,
    dwNumberOfBytesWritten )
```

Value:

```
WriteFile(hFile, pBuffer, dwNumberOfBytesToWrite, \
dwNumberOfBytesWritten, NULL) ? WD_STATUS_SUCCESS : \
WD_OPERATION_FAILED
```

Definition at line 1770 of file [windrvr.h](#).

WD_UTransfer

```
#define WD_UTransfer(
    h,
    pTrans )  WD_FUNCTION(IOCTL_WDU_TRANSFER, h, pTrans, sizeof(WDU_TRANSFER), TRUE);
```

Definition at line 2680 of file [windrvr.h](#).

WD_UWakeup

```
#define WD_UWakeup(
    h,
    pWakeup )  WD_FUNCTION(IOCTL_WDU_WAKEUP, h, pWakeup, sizeof(WDU_WAKEUP), FALSE);
```

Definition at line 2685 of file [windrvr.h](#).

WD_VER_STR

```
#define WD_VER_STR
```

Value:

```
WD_PROD_NAME " v" WD_VERSION_STR \
" Jungs Connectivity (c) 1997 - " COPYRIGHTS_YEAR_STR \
" Build Date: " __DATE__ \
WD_CPU_SPEC WD_DATA_MODEL WD_FILE_FORMAT
```

Definition at line 148 of file [windrvr.h](#).

WD_Version

```
#define WD_Version(
    h,
    pVerInfo )  WD_FUNCTION(IOCTL_WD_VERSION, h, pVerInfo, sizeof(WD_VERSION), FALSE)
```

Returns the version number of the WinDriver kernel module currently running.

Parameters

in	<i>h</i>	Handle to WinDriver's kernel-mode driver as received from WD_Open()
out	<i>pVerInfo</i>	Pointer to a WinDriver version information structure

Returns

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise

```
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");
```

}

Definition at line 2481 of file [windrvr.h](#).

WD_VERSION_STR_LENGTH

```
#define WD_VERSION_STR_LENGTH 128
```

Definition at line 61 of file [windrvr.h](#).

WIN32

```
#define WIN32
```

Definition at line 166 of file [windrvr.h](#).

WINAPI

```
#define WINAPI
```

Definition at line 311 of file [windrvr.h](#).

Typedef Documentation

BYTE

```
typedef unsigned char BYTE
```

Definition at line 315 of file [windrvr.h](#).

DMA_ADDR

```
typedef UINT64 DMA_ADDR
```

Definition at line 360 of file [windrvr.h](#).

DMA_TRANSACTION_CALLBACK

```
typedef void(DLLCALLCONV * DMA_TRANSACTION_CALLBACK) (PVOID pData)
```

Definition at line 506 of file [windrvr.h](#).

KPTR

```
typedef UINT32 KPTR
```

Definition at line 351 of file [windrvr.h](#).

PHYS_ADDR

```
typedef UINT64 PHYS_ADDR
```

Definition at line 361 of file [windrvr.h](#).

UINT32

```
typedef unsigned int UINT32
```

Definition at line 320 of file [windrvr.h](#).

UINT64

```
typedef unsigned __int64 UINT64  
Definition at line 297 of file windrvr.h.
```

UPTR

```
typedef size_t UPTR  
Definition at line 357 of file windrvr.h.
```

WD_BUS_TYPE

```
typedef DWORD WD_BUS_TYPE  
Definition at line 661 of file windrvr.h.
```

WD_BUS_V30

```
typedef struct WD_BUS WD_BUS_V30
```

WD_CARD_REGISTER_V118

```
typedef struct WD_CARD_REGISTER WD_CARD_REGISTER_V118
```

WD_CARD_V118

```
typedef struct WD_CARD WD_CARD_V118
```

WD_DEBUG_ADD_V503

```
typedef struct WD_DEBUG_ADD WD_DEBUG_ADD_V503
```

WD_DEBUG_DUMP_V40

```
typedef struct WD_DEBUG_DUMP WD_DEBUG_DUMP_V40
```

WD_DEBUG_V40

```
typedef struct WD_DEBUG WD_DEBUG_V40
```

WD_DMA_PAGE_V80

```
typedef struct WD_DMA_PAGE WD_DMA_PAGE_V80
```

WD_DMA_V80

```
typedef struct WD_DMA WD_DMA_V80
```

WD_EVENT_TYPE

```
typedef DWORD WD_EVENT_TYPE  
Definition at line 1475 of file windrvr.h.
```

WD_EVENT_V121

```
typedef struct WD_EVENT WD_EVENT_V121
```

WD_INTERRUPT_V91

```
typedef struct WD_INTERRUPT WD_INTERRUPT_V91
```

WD_IPC_PROCESS_V121

```
typedef struct WD_IPC_PROCESS WD_IPC_PROCESS_V121
```

WD_IPC_REGISTER_V121

```
typedef struct WD_IPC_REGISTER WD_IPC_REGISTER_V121
```

WD_IPC_SCAN_PROCS_V121

```
typedef struct WD_IPC_SCAN_PROCS WD_IPC_SCAN_PROCS_V121
```

WD_IPC_SEND_V121

```
typedef struct WD_IPC_SEND WD_IPC_SEND_V121
```

WD_ITEMS_V118

```
typedef struct WD_ITEMS WD_ITEMS_V118
```

WD_KERNEL_BUFFER_V121

```
typedef struct WD_KERNEL_BUFFER WD_KERNEL_BUFFER_V121
```

WD_KERNEL_PLUGIN_CALL_V40

```
typedef struct WD_KERNEL_PLUGIN_CALL WD_KERNEL_PLUGIN_CALL_V40
```

WD_KERNEL_PLUGIN_V40

```
typedef struct WD_KERNEL_PLUGIN WD_KERNEL_PLUGIN_V40
```

WD_LICENSE_V122

```
typedef struct WD_LICENSE WD_LICENSE_V122
```

WD_PCI_CARD_INFO_V118

```
typedef struct WD_PCI_CARD_INFO WD_PCI_CARD_INFO_V118
```

WD_PCI_CONFIG_DUMP_V30

```
typedef struct WD_PCI_CONFIG_DUMP WD_PCI_CONFIG_DUMP_V30
```

WD_PCI_SCAN_CAPS_V118

```
typedef struct WD_PCI_SCAN_CAPS WD_PCI_SCAN_CAPS_V118
```

WD_PCI_SCAN_CARDS_V124

```
typedef struct WD_PCI_SCAN_CARDS WD_PCI_SCAN_CARDS_V124
```

WD_PCI_SRIOV_V122

```
typedef struct WD_PCI_SRIOV WD_PCI_SRIOV_V122
```

WD_SLEEP_V40

```
typedef struct WD_SLEEP WD_SLEEP_V40
```

WD_TRANSFER_V61

```
typedef struct WD_TRANSFER WD_TRANSFER_V61
```

WD_VERSION_V30

```
typedef struct WD_VERSION WD_VERSION_V30
```

WORD

typedef unsigned short int WORD

Definition at line 316 of file [windrvr.h](#).

Enumeration Type Documentation

anonymous enum

anonymous enum

Enumerator

WD_DMA_PAGES	
--------------	--

Definition at line 410 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_MATCH_EXCLUDE	Exclude if there is a match.
------------------	------------------------------

Definition at line 488 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_PCI_CAP_ID_ALL	
-------------------	--

Definition at line 864 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

SLEEP_BUSY	
SLEEP_NON_BUSY	

Definition at line 926 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_EVENT_TYPE_UNKNOWN	
WD_EVENT_TYPE_PCI	
WD_EVENT_TYPE_USB	
WD_EVENT_TYPE_IPC	

Definition at line 1468 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_USB_HARD_RESET	
WD_USB_CYCLE_PORT	

Definition at line 1521 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

INTERRUPT_LATCHED	Legacy Latched Interrupts .
INTERRUPT_LEVEL_SENSITIVE	Legacy Level Sensitive Interrupts.
INTERRUPT_CMD_COPY	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.
INTERRUPT_CE_INT_ID	Obsolete.
INTERRUPT_CMD_RETURN_VALUE	Obsolete.
INTERRUPT_MESSAGE	Message-Signaled Interrupts (MSI).
INTERRUPT_MESSAGE_X	Extended Message-Signaled Interrupts (MSI-X).
INTERRUPT_DONT_GET_MSI_MESSAGE	Linux Only. Do not read the MSI/MSI-X message from the card.

Definition at line 583 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_BUS_USB	USB.
WD_BUS_UNKNOWN	Unknown bus type.
WD_BUS_ISA	ISA.
WD_BUS_EISA	EISA, including ISA PnP.
WD_BUS_PCI	PCI.

Definition at line 653 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_CARD_ITEMS	
---------------	--

Definition at line 743 of file [windrvr.h](#).

anonymous enum

```
anonymous enum
```

Enumerator

WD_IPC_MAX_PROCS	
------------------	--

Definition at line 781 of file [windrvr.h](#).

anonymous enum

anonymous enum

Enumerator

WD_IPC_UID_UNICAST	<input type="checkbox"/>
WD_IPC_SUBGROUP_MULTICAST	<input type="checkbox"/>
WD_IPC_MULTICAST	<input type="checkbox"/>

Definition at line 792 of file [windrvr.h](#).

anonymous enum

anonymous enum

Enumerator

WD_FORCE_CLEANUP	<input type="checkbox"/>
------------------	--------------------------

Definition at line 820 of file [windrvr.h](#).

anonymous enum

anonymous enum

Enumerator

WD_PCI_CARDS	<input type="checkbox"/>
--------------	--------------------------

Definition at line 822 of file [windrvr.h](#).

anonymous enum

anonymous enum

Enumerator

WD_PCI_MAX_CAPS	<input type="checkbox"/>
-----------------	--------------------------

Definition at line 862 of file [windrvr.h](#).

DEBUG_COMMAND

enum DEBUG_COMMAND

Enumerator

DEBUG_STATUS	<input type="checkbox"/>
DEBUG_SET_FILTER	<input type="checkbox"/>
DEBUG_SET_BUFFER	<input type="checkbox"/>

Enumerator

DEBUG_CLEAR_BUFFER	
DEBUG_DUMP_SEC_ON	
DEBUG_DUMP_SEC_OFF	
KERNEL_DEBUGGER_ON	
KERNEL_DEBUGGER_OFF	
DEBUG_DUMP_CLOCK_ON	
DEBUG_DUMP_CLOCK_OFF	
DEBUG_CLOCK_RESET	

Definition at line 963 of file [windrvr.h](#).

DEBUG_LEVEL

enum DEBUG_LEVEL

Enumerator

D_OFF	
D_ERROR	
D_WARN	
D_INFO	
D_TRACE	

Definition at line 934 of file [windrvr.h](#).

DEBUG_SECTION

enum DEBUG_SECTION

Enumerator

S_ALL	
S_IO	
S_MEM	
S_INT	
S_PCI	
S_DMA	
S_MISC	
S_LICENSE	
S_PNP	
S_CARD_REG	
S_KER_DRV	
S_USB	
S_KER_PLUG	
S_EVENT	
S_IPC	
S_KER_BUF	

Definition at line 943 of file [windrvr.h](#).

ITEM_TYPE

```
enum ITEM_TYPE
```

Enumerator

ITEM_NONE	
ITEM_INTERRUPT	Interrupt.
ITEM_MEMORY	Memory.
ITEM_IO	I/O.
ITEM_BUS	Bus.

Definition at line 671 of file [windrvr.h](#).

PCI_ACCESS_RESULT

```
enum PCI_ACCESS_RESULT
```

Enumerator

PCI_ACCESS_OK	
PCI_ACCESS_ERROR	
PCI_BAD_BUS	
PCI_BAD_SLOT	

Definition at line 905 of file [windrvr.h](#).

WD_DMA_OPTIONS

```
enum WD_DMA_OPTIONS
```

Enumerator

DMA_KERNEL_BUFFER_ALLOC	The system allocates a contiguous buffer. The user does not need to supply linear address.
DMA_KBUF_BELOW_16M	If DMA_KERNEL_BUFFER_ALLOC is used, this will make sure it is under 16M.
DMA_LARGE_BUFFER	If DMA_LARGE_BUFFER is used, the maximum number of pages are dwPages, and not WD_DMA_PAGES. If you lock a user buffer (not a kernel allocated buffer) that is larger than 1MB, then use this option and allocate memory for pages.
DMA_ALLOW_CACHE	Allow caching of contiguous memory.
DMA_KERNEL_ONLY_MAP	Only map to kernel, dont map to user-mode. relevant with DMA_KERNEL_BUFFER_ALLOC flag only
DMA_FROM_DEVICE	memory pages are locked to be written by device
DMA_TO_DEVICE	memory pages are locked to be read by device
DMA_TO_FROM_DEVICE	memory pages are locked for both read and write
DMA_ALLOW_64BIT_ADDRESS	Use this value for devices that support 64-bit DMA addressing.
DMA_ALLOW_NO_HCARD	Allow memory lock without hCard.
DMA_GET_EXISTING_BUF	Get existing buffer by hDma handle.
DMA_RESERVED_MEM	

Enumerator

DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH	When using this flag, the width of the address must be entered in the fourth byte of dwOptions and then the allocated address will be limited to this width. Linux: works with contiguous buffers only.
DMA_GET_PREALLOCATED_BUFFERS_ONLY	Windows: Try to allocate buffers from preallocated buffers pool ONLY (if none of them are available, function will fail).
DMA_TRANSACTION	Use this flag to use the DMA transaction mechanism.
DMA_GPUDIRECT	Linux only.
DMA_DISABLE_MERGE_ADJACENT_PAGES	Disable merge adjacent pages. In case the flag is omitted, the merge will take place automatically. Used for scatter gather mode only.

Definition at line 416 of file [windrvr.h](#).

WD_ERROR_CODES

enum [WD_ERROR_CODES](#)

Enumerator

WD_STATUS_SUCCESS	[0] Operation completed successfully
WD_STATUS_INVALID_WD_HANDLE	[0xffffffff]
WD_WINDRIVER_STATUS_ERROR	[0x20000000]
WD_INVALID_HANDLE	[0x20000001] Invalid WinDriver handle. This usually occurs when <ul style="list-style-type: none"> 1. The WinDriver kernel module is not installed. 2. The user application is trying to access a WinDriver kernel module with the wrong name 3. A wrong handle was provided to an API.
WD_INVALID_PIPE_NUMBER	[0x20000002] USB: Invalid Pipe Number. Occurs when the provided pipe number does not exist or is not available on the USB device.
WD_READ_WRITE_CONFLICT	[0x20000003] Request to read from an OUT (write) pipe or request to write to an IN (read) pipe
WD_ZERO_PACKET_SIZE	[0x20000004] Maximum packet size is zero
WD_INSUFFICIENT_RESOURCES	[0x20000005] Insufficient resources. Occurs when WinDriver is trying to allocate more memory than currently available.
WD_UNKNOWN_PIPE_TYPE	[0x20000006] Obsolete.
WD_SYSTEM_INTERNAL_ERROR	[0x20000007] Internal System Error
WD_DATA_MISMATCH	[0x20000008] Data Mismatch. Occurs when compiling a 32 bit user application on a 64-bit platform without adding the KERNEL_64BIT define.
WD_NO_LICENSE	0x[20000009] No License. Occurs either when no license was entered, license has expired, license was not entered correctly. Contact wd_license@jungo.com for more assistance.

Enumerator

WD_NOT_IMPLEMENTED	[0x2000000a] Function not implemented. Occurs when a certain API is not implemented for the platform that is being used.
WD_KERPLUG_FAILURE	[0x2000000b] Kernel PlugIn failure. Occurs when trying to open a KP which is not currently installed, or when the KP was not implemented correctly.
WD_FAILED_ENABLING_INTERRUPT	[0x2000000c] Failed Enabling Interrupts. Occurs when the OS fails to enable interrupts for the device or when trying to enable MSI interrupts for a device that doesn't support them. Note that an INF file MUST be installed in order to enable interrupts under Windows.
WD_INTERRUPT_NOT_ENABLED	[0x2000000d] Interrupt not enabled. Occurs when calling WD_IntWait() with an invalid interrupt handle, or when trying to disable a shared interrupt that was not enabled.
WD_RESOURCE_OVERLAP	[0x2000000e] Resource Overlap. Happens when trying to open the same device with more than one user application (when that device was not defined to be opened by more than one process).
WD_DEVICE_NOT_FOUND	[0x2000000f] Device not found. Occurs when trying to open a device that was not detected by the OS.
WD_WRONG_UNIQUE_ID	[0x20000010] Wrong unique ID (USB). Occurs when trying to access a USB device with an invalid unique ID.
WD_OPERATION_ALREADY_DONE	[0x20000011] Operation Already Done. Occurs when trying to perform twice an operation that should be done once (i.e. Enabling interrupts)
WD_USB_DESCRIPTOR_ERROR	[0x20000012] USB Descriptor Error. Occurs when WinDriver receives faulty USB descriptor data, might be a result of an invalid parameter to the WinDriver API or a result of a bug in the device's firmware.
WD_SET_CONFIGURATION_FAILED	[0x20000013] Set Configuration Failed (USB). Occurs when WinDriver failed to set a configuration for a USB device.
WD_CANT_OBTAIN_PDO	[0x20000014] Obsolete
WD_TIME_OUT_EXPIRED	[0x20000015] Time Out Expired. Occurs when operations that have a timeout limit run out of time (i.e. DMA or USB transfers)
WD_IRP_CANCELED	[0x20000016] IRP Cancelled. Occurs on Windows when an interrupt was cancelled by the user in the middle of an operation.
WD_FAILED_USER_MAPPING	[0x20000017] Failed to map memory to User Space. Occurs when there is not enough available memory to map a PCI device's BARs to user space memory. Possible ways to deal with this are to decrease the BAR size through the firmware, add more RAM to your computer or free up more RAM.
WD_FAILED_KERNEL_MAPPING	[0x20000018] Failed to map memory to Kernel Space. Occurs when there is not enough available memory to map a PCI device's BARs to kernel space memory. Possible ways to deal with this are to decrease the BAR size through the firmware, add more RAM to your computer or free up more RAM.

Enumerator

WD_NO_RESOURCES_ON_DEVICE	[0x20000019] No Resources On Device. Occurs when a user application expects a resource (i.e. BAR, Interrupt...) to be available on a device but that resource is unavailable. This is usually a result of how the device was designed/programmed.
WD_NO_EVENTS	[0x2000001a] No Events. Occurs when calling WD_EventSend() , WD_EventPull() without registering events using WD_EventRegister() beforehand.
WD_INVALID_PARAMETER	[0x2000001b] Invalid Parameter. Occurs when providing an invalid parameter to an API. Try fixing the parameters provided to the function.
WD_INCORRECT_VERSION	[0x2000001c] Incorrect version. Occurs when <ol style="list-style-type: none"> 1. Mixing together components from different WinDriver versions. 2. Failure to obtain the version of a Kernel Plugin
WD_TRY AGAIN	[0x2000001d] Try Again. Occurs when an operation has failed due to a device being busy. When trying again later the operation may succeed.
WD_WINDRIVER_NOT_FOUND	[0x2000001e] Obsolete
WD_INVALID_IOCTL	[0x2000001f] Invalid IOCTL. Occurs when providing an invalid API to the WinDriver kernel module. Use only APIs that are defined in this user manual.
WD_OPERATION_FAILED	[0x20000020] Operation Failed. Occurs either because of an invalid input from user, or an internal system error.
WD_INVALID_32BIT_APP	[0x20000021] Invalid 32 Bit User Application. Occurs when compiling a 32 bit user application on a 64-bit platform without adding the KERNEL_64BIT define.
WD_TOO_MANY_HANDLES	[0x20000022] Too Many Handles. Occurs when too many handles to WinDriver or to a certain WinDriver feature have been opened. Try closing some handles to prevent this from happening.
WD_NO_DEVICE_OBJECT	[0x20000023] No Device Object. <ol style="list-style-type: none"> 1. On Windows this usually means that no INF file was installed for the device. Please install an INF using the DriverWizard to resolve this. 2. On Linux this may mean that the device is controlled by a different driver and not by WinDriver. A possible solution would be to blacklist the other driver in order to allow WinDriver to take control of the device.
WD_MORE_PROCESSING_REQUIRED	[0xC0000016] More Processing Required. Occurs when a part of the operation was complete but more calls to an API for continuing to process the data will be required.
WD_USBD_STATUS_SUCCESS	[0x00000000] Success
WD_USBD_STATUS_PENDING	[0x40000000] Operation pending
WD_USBD_STATUS_ERROR	[0x80000000] Error
WD_USBD_STATUS_HALTED	[0xC0000000] Halted

Enumerator

WD_USBD_STATUS_CRC	[0xC0000001] HC status: CRC
WD_USBD_STATUS_BTSTUFF	[0xC0000002] HC status: Bit stuffing
WD_USBD_STATUS_DATA_TOGGLE_MISMATCH	[0xC0000003] HC status: Data toggle mismatch
WD_USBD_STATUS_STALL_PID	[0xC0000004] HC status: PID stall
WD_USBD_STATUS_DEV_NOT RESPONDING	[0xC0000005] HC status: Device not responding
WD_USBD_STATUS_PID_CHECK_FAILURE	[0xC0000006] HC status: PID check failed
WD_USBD_STATUS_UNEXPECTED_PID	[0xC0000007] HC status: Unexpected PID
WD_USBD_STATUS_DATA_OVERRUN	[0xC0000008] HC status: Data overrun
WD_USBD_STATUS_DATA_UNDERRUN	[0xC0000009] HC status: Data underrun
WD_USBD_STATUS_RESERVED1	[0xC000000A] HC status: Reserved1
WD_USBD_STATUS_RESERVED2	[0xC000000B] HC status: Reserved1
WD_USBD_STATUS_BUFFER_OVERRUN	[0xC000000C] HC status: Buffer overrun
WD_USBD_STATUS_BUFFER_UNDERUN	[0xC000000D] HC status: Buffer underrun
WD_USBD_STATUS_NOT_ACCESED	[0xC000000F] HC status: Not accessed
WD_USBD_STATUS_FIFO	[0xC0000010] HC status: FIFO
WD_USBD_STATUS_XACT_ERROR	[0xC0000011] HC status: The host controller has set the Transaction Error (XactErr) bit in the transfer descriptor's status field
WD_USBD_STATUS_BABBLE_DETECTED	[0xC0000012] HC status: Babble detected
WD_USBD_STATUS_DATA_BUFFER_ERROR	[0xC0000013] HC status: Data buffer error
WD_USBD_STATUS_CANCELED	[0xC0010000] USBD: Transfer canceled
WD_USBD_STATUS_ENDPOINT_HALTED	Returned by HCD (Host Controller Driver) if a transfer is submitted to an endpoint that is stalled:
WD_USBD_STATUS_NO_MEMORY	[0x80000100] USBD: Out of memory
WD_USBD_STATUS_INVALID_URB_FUNCTION	[0x80000200] USBD: Invalid URB function
WD_USBD_STATUS_INVALID_PARAMETER	[0x80000300] USBD: Invalid parameter
WD_USBD_STATUS_ERROR_BUSY	[0x80000400] Returned if client driver attempts to close an endpoint/interface or configuration with outstanding transfers:
WD_USBD_STATUS_REQUEST_FAILED	[0x80000500] Returned by USBD if it cannot complete a URB request. Typically this will be returned in the URB status field when the Irp is completed with a more specific error code. [The Irp status codes are indicated in WinDriver's Debug Monitor tool (wddebug/wddebug_gui):]
WD_USBD_STATUS_INVALID_PIPE_HANDLE	[80000600] USBD: Invalid pipe handle
WD_USBD_STATUS_NO_BANDWIDTH	[0x80000700] Returned when there is not enough bandwidth available to open a requested endpoint:
WD_USBD_STATUS_INTERNAL_HC_ERROR	[0x80000800] Generic HC (Host Controller) error:
WD_USBD_STATUS_ERROR_SHORT_TRANSFER	[0x80000900] Returned when a short packet terminates the transfer i.e. USBD_SHORT_TRANSFER_OK bit not set:
WD_USBD_STATUS_BAD_START_FRAME	[0x80000A00] Returned if the requested start frame is not within USBD_ISO_START_FRAME_RANGE of the current USB frame, Note The stall bit is set:

Enumerator

WD_USBD_STATUS_ISOCH_REQUEST_FAILED	[0xC0000B00] Returned by HCD (Host Controller Driver) if all packets in an isochronous transfer complete with an error:
WD_USBD_STATUS_FRAME_CONTROL_OWNED	[0xC0000C00] Returned by USBD if the frame length control for a given HC (Host Controller) is already taken by another driver:
WD_USBD_STATUS_FRAME_CONTROL_NOT_↔ OWNED	[0xC0000D00] Returned by USBD if the caller does not own frame length control and attempts to release or modify the HC frame length:
WD_USBD_STATUS_NOT_SUPPORTED	[0xC0000E00] Returned for APIs not supported/implemented
WD_USBD_STATUS_INAVLID_CONFIGURATION↔ _DESCRIPTOR	[0xC000F00] USBD: Invalid configuration descriptor
WD_USBD_STATUS_INSUFFICIENT_RESOURCES	[0xC0001000] USBD: Insufficient resources
WD_USBD_STATUS_SET_CONFIG_FAILED	[0xC0002000] USBD: Set configuration failed
WD_USBD_STATUS_BUFFER_TOO_SMALL	[0xC0003000] USBD: Buffer too small
WD_USBD_STATUS_INTERFACE_NOT_FOUND	[0xC0004000] USBD: Interface not found
WD_USBD_STATUS_INAVLID_PIPE_FLAGS	[0xC0005000] USBD: Invalid pipe flags
WD_USBD_STATUS_TIMEOUT	[0xC0006000] USBD: Timeout
WD_USBD_STATUS_DEVICE_GONE	[0xC0007000] USBD: Device Gone
WD_USBD_STATUS_STATUS_NOT_MAPPED	[0xC0008000] USBD: Status not mapped
WD_USBD_STATUS_ISO_NOT_ACCESSED_BY↔ _HW	Extended isochronous error codes returned by USBD. These errors appear in the packet status field of an isochronous transfer. [0xC0020000] For some reason the controller did not access the TD associated with this packet:
WD_USBD_STATUS_ISO_TD_ERROR	[0xC0030000] Controller reported an error in the TD. Since TD errors are controller specific they are reported generically with this error code:
WD_USBD_STATUS_ISO_NA_LATE_USBPORT	[0xC0040000] The packet was submitted in time by the client but failed to reach the miniport in time:
WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE	[0xC0050000] The packet was not sent because the client submitted it too late to transmit:

Definition at line 1040 of file [windrvr.h](#).

WD_EVENT_ACTION

enum [WD_EVENT_ACTION](#)

Enumerator

WD_INSERT	
WD_REMOVE	
WD_OBSOLETE	Obsolete.
WD_POWER_CHANGED_D0	Power states for the power management.
WD_POWER_CHANGED_D1	
WD_POWER_CHANGED_D2	
WD_POWER_CHANGED_D3	
WD_POWER_SYSTEM_WORKING	
WD_POWER_SYSTEM_SLEEPING1	

Enumerator

WD_POWER_SYSTEM_SLEEPING2	
WD_POWER_SYSTEM_SLEEPING3	
WD_POWER_SYSTEM_HIBERNATE	
WD_POWER_SYSTEM_SHUTDOWN	
WD_IPC_UNICAST_MSG	
WD_IPC_MULTICAST_MSG	

Definition at line 1434 of file [windrvr.h](#).

WD_EVENT_OPTION

enum [WD_EVENT_OPTION](#)

Enumerator

WD_ACKNOWLEDGE	
WD_ACCEPT_CONTROL	used in WD_EVENT_SEND (acknowledge)

Definition at line 1456 of file [windrvr.h](#).

WD_GET_DEVICE_PROPERTY_OPTION

enum [WD_GET_DEVICE_PROPERTY_OPTION](#)

IOCTL Structures.

Enumerator

WD_DEVICE_PCI	
WD_DEVICE_USB	

Definition at line 1019 of file [windrvr.h](#).

WD_INTERRUPT_WAIT_RESULT

enum [WD_INTERRUPT_WAIT_RESULT](#)

Enumerator

INTERRUPT_RECEIVED	Interrupt was received.
INTERRUPT_STOPPED	Interrupt was disabled during wait.
INTERRUPT_INTERRUPTED	Wait was interrupted before an actual hardware interrupt was received.

Definition at line 611 of file [windrvr.h](#).

WD_ITEM_MEM_OPTIONS

enum [WD_ITEM_MEM_OPTIONS](#)

Enumerator

WD_ITEM_MEM_DO_NOT_MAP_KERNEL	Skip the mapping of physical memory to the kernel address space.
WD_ITEM_MEM_ALLOW_CACHE	Map physical memory as cached; applicable only to host RAM, not to local memory on the card.
WD_ITEM_MEM_USER_MAP	Map physical memory from user mode, Linux only.

Definition at line 680 of file [windrvr.h](#).

WD_KER_BUF_OPTION

enum [WD_KER_BUF_OPTION](#)

Enumerator

KER_BUF_ALLOC_NON_CONTIG	
KER_BUF_ALLOC_CONTIG	
KER_BUF_ALLOC_CACHED	
KER_BUF_GET_EXISTING_BUF	

Definition at line 541 of file [windrvr.h](#).

WD_PCI_SCAN_CAPS_OPTIONS

enum [WD_PCI_SCAN_CAPS_OPTIONS](#)

Enumerator

WD_PCI_SCAN_CAPS_BASIC	Scan basic PCI capabilities.
WD_PCI_SCAN_CAPS_EXTENDED	Scan extended (PCIe) PCI capabilities.

Definition at line 872 of file [windrvr.h](#).

WD_PCI_SCAN_OPTIONS

enum [WD_PCI_SCAN_OPTIONS](#)

Enumerator

WD_PCI_SCAN_DEFAULT	
WD_PCI_SCAN_BY_TOPOLOGY	
WD_PCI_SCAN_REGISTERED	
WD_PCI_SCAN_INCLUDE_DOMAINS	

Definition at line 855 of file [windrvr.h](#).

WD_TRANSFER_CMD

enum [WD_TRANSFER_CMD](#)

IN [WD_TRANSFER_CMD](#) and [WD_Transfer\(\)](#) DWORD stands for 32 bits and QWORD is 64 bit.

Enumerator

CMD_NONE	No command.
CMD_END	End command.
CMD_MASK	Interrupt Mask.
RP_BYTE	Read port byte.
RP_WORD	Read port word.
RP_DWORD	Read port dword.
WP_BYTE	Write port byte.
WP_WORD	Write port word.
WP_DWORD	Write port dword.
RP_QWORD	Read port qword.
WP_QWORD	Write port qword.
RP_SBYTE	Read port string byte.
RP_SWORD	Read port string word.
RP_SDWORD	Read port string dword.
WP_SBYTE	Write port string byte.
WP_SWORD	Write port string word.
WP_SDWORD	Write port string dword.
RP_SQWORD	Read port string qword.
WP_SQWORD	Write port string qword.
RM_BYTE	Read memory byte.
RM_WORD	Read memory word.
RM_DWORD	Read memory dword.
WM_BYTE	Write memory byte.
WM_WORD	Write memory word.
WM_DWORD	Write memory dword.
RM_QWORD	Read memory qword.
WM_QWORD	Write memory qword.
RM_SBYTE	Read memory string byte.
RM_SWORD	Read memory string word.
RM_SDWORD	Read memory string dword.
WM_SBYTE	Write memory string byte.
WM_SWORD	Write memory string word.
WM_SDWORD	Write memory string dword.
RM_SQWORD	Read memory string quad word.
WM_SQWORD	Write memory string quad word.

Definition at line 367 of file [windrvr.h](#).

Function Documentation

check_secureBoot_enabled()

```
DWORD DLLCALLCONV check_secureBoot_enabled (
    void )
```

Checks whether the Secure Boot feature is enabled on the system.

Developing drivers while Secure Boot is enabled can result in driver installation problems.

Returns

WD_STATUS_SUCCESS (0) when Secure Boot is enabled. WD_WINDRIVER_STATUS_ERROR when Secure Boot is disabled. Any other status when Secure Boot is not supported.

get_os_type()

```
WD_OS_INFO DLLCALLCONV get_os_type (
    void )
```

Retrieves the type of the operating system in section.

Returns

Returns the type of the running operating system. If the operating system type is not detected, cProdName field will get a OS_CAN_NOT_DETECT_TEXT value.

WD_DriverName()

```
const char *DLLCALLCONV WD_DriverName (
    const char * sName )
```

Sets the name of the WinDriver kernel module, which will be used by the calling application.

The default driver name, which is used if the function is not called, is windrvr@WD_VERSION_NUMBER@.

This function must be called once, and only once, from the beginning of your application, before calling any other WinDriver function (including [WD_Open\(\)](#) / [WDC_DriverOpen\(\)](#) / [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#)), as demonstrated in the sample and generated DriverWizard WinDriver applications, which include a call to this function with the default driver name - windrvr@WD_VERSION_NUMBER@

On Windows and Linux, if you select to modify the name of the WinDriver kernel module (windrvr@WD_VERSION_NUMBER@.sys/.dll/.o/.ko), as explained in [17.2. Renaming the WinDriver Kernel Driver](#), you must ensure that your application calls [WD_DriverName\(\)](#) with your new driver name.

In order to use the [WD_DriverName\(\)](#) function, your user-mode driver project must be built with WD_DRIVER_NAME_CHANGE preprocessor flag (e.g.: -DWD_DRIVER_NAME_CHANGE — for MS Visual Studio, Windows GCC, and GCC). The sample and generated DriverWizard Windows and Linux WinDriver projects/makefiles already set this preprocessor flag.

Parameters

in	sName	The name of the WinDriver kernel module to be used by the application. NOTE: The driver name should be indicated without the driver file's extension. For example, use windrvr@WD_VERSION_NUMBER@, not windrvr@WD_VERSION_NUMBER@.sys or windrvr@WD_VERSION_NUMBER@.o.
----	-------	--

Returns

Returns the selected driver name on success; returns NULL on failure (e.g., if the function is called twice from the same application).

```
int main(void)
{
    DWORD dwStatus;
    WDC_DEVICE_HANDLE hDev;
    WD_PCI_SLOT slot = {0, 0, 0, 0}; /* Replace domain/bus/slot/function with your own */
    WD_PCI_CARD_INFO deviceInfo;
    BYTE bData = 0x1;
    WORD wData = 0x2;
    UINT32 u32Data = 0x3;
    UINT64 u64Data = 0x4;
    /* Make sure to fully initialize WinDriver! */
    if (!WD_DriverName(DEFAULT_DRIVER_NAME))
    {
        printf("Failed to set the driver name for WDC library.\n");
        return WD_INTERNAL_ERROR;
    }
```

```
dwStatus = WDC_SetDebugOptions(WDC_DBG_DEFAULT, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize debug options for WDC library.\n"
        "Error 0x%lx - %s\n", dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
/* Open a handle to the driver and initialize the WDC library */
dwStatus = WDC_DriverOpen(WDC_DRV_OPEN_DEFAULT, DEFAULT_LICENSE_STRING);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed to initialize the WDC library. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    return dwStatus;
}
BZERO(deviceInfo);
deviceInfo.pciSlot = slot;
dwStatus = WDC_PciGetDeviceInfo(&deviceInfo);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed retrieving the device's resources "
        "information. Error [0x%lx - %s]\n", dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Open a device handle */
dwStatus = WDC_PciDeviceOpen(&hDev, &deviceInfo, NULL);
if (WD_STATUS_SUCCESS != dwStatus)
{
    printf("Failed opening a WDC device handle. Error 0x%lx - %s\n",
        dwStatus, Stat2Str(dwStatus));
    goto Exit;
}
/* Read/Write memory examples: */
/* Check BYTE */
dwResult = WDC_WriteAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
dwResult = WDC_ReadAddrBlock8(hDev, MEM_SPACE, REG, 1,
    &bData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
/* Check WORD */
dwResult = WDC_WriteAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
dwResult = WDC_ReadAddrBlock16(hDev, MEM_SPACE, REG, 2,
    &wData, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
/* Check UINT32 */
dwResult = WDC_WriteAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
dwResult = WDC_ReadAddrBlock32(hDev, MEM_SPACE, REG, 4,
    &u32Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
/* Check UINT64 */
dwResult = WDC_WriteAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
dwResult = WDC_ReadAddrBlock64(hDev, MEM_SPACE, REG, 8,
    &u64Data, WDC_ADDR_RW_DEFAULT);
if (dwResult)
{
    goto Exit;
}
Exit:
if (hDev)
{
    dwStatus = WDC_PciDeviceClose(hDev);
    if (WD_STATUS_SUCCESS != dwStatus)
    {
        printf("Failed closing a WDC device handle (0x%p). Error 0x%lx "
            "- %s\n", hDev, dwStatus, Stat2Str(dwStatus));
    }
}
WDC_DriverClose();
}
```

windrvr.h

Go to the documentation of this file.

```
00001 /* @JUNGO_COPYRIGHT@ */
00002 /*
00003 */
00004 * W i n D r i v e r
00005 * =====
00006 *
00007 * FOR DETAILS ON THE WinDriver FUNCTIONS, PLEASE SEE THE WinDriver MANUAL
00008 * OR INCLUDED HELP FILES.
00009 *
00010 * This file may not be distributed, it may only be used for development
00011 * or evaluation purposes. The only exception is distribution to Linux.
00012 * For details refer to \WinDriver\docs\license.txt.
00013 *
00014 * Web site: https://www.jungo.com
00015 * Email: support@jungo.com
00016 */
00017 #ifndef _WINDRVR_H_
00018 #define _WINDRVR_H_
00019
00020 #if defined(__cplusplus)
00021     extern "C" {
00022 #endif
00023
00024 #include "wd_ver.h"
00025
00026 #if defined(WIN32) && !defined(__MINGW32__)
00027     #define DLLCALLCONV __stdcall
00028     #if !defined(_SAL_VERSION)
00029         #include <sal.h>
00030     #endif
00031 #else
00032     #define DLLCALLCONV
00033
00034 /* the following macros are part of SAL annotations macros in Windows and
00035 are used in prototype of API functions, therefore, must be defined in
00036 Unix as well */
00037 #define _In_
00038 #define _Inout_
00039 #define _Out_
00040 #define _Outptr_
00041 #endif
00042
00043 #if defined(WIN32)
00044     #define WD_DRIVER_NAME_PREFIX "\\\\".\\\""
00045 #elif defined(LINUX)
00046     #define WD_DRIVER_NAME_PREFIX "/dev/"
00047 #else
00048     #define WD_DRIVER_NAME_PREFIX ""
00049 #endif
00050
00051 #if !defined(WIN32)
00052     #define __FUNCTION__ __func__
00053 #endif
00054
00055 #define WD_DEFAULT_DRIVER_NAME_BASE "windrvr" WD_VER_ITOA
00056 #define WD_DEFAULT_DRIVER_NAME \
00057     WD_DRIVER_NAME_PREFIX WD_DEFAULT_DRIVER_NAME_BASE
00058
00059 #define WD_MAX_DRIVER_NAME_LENGTH 128
00060 #define WD_MAX_KP_NAME_LENGTH 128
00061 #define WD_VERSION_STR_LENGTH 128
00062
00063 #if defined(WD_DRIVER_NAME_CHANGE)
00103     const char* DLLCALLCONV WD_DriverName(const char *sName);
00105     #define WD_DRIVER_NAME WD_DriverName(NULL)
00106 #else
00107     #define WD_DRIVER_NAME WD_DEFAULT_DRIVER_NAME
00108 #endif
00109
00110 #define WD_PROD_NAME "WinDriver"
00111
00112 #if !defined(ARM) && \
00113     !defined(ARM64) && \
00114     !defined(x86) && \
00115     (defined(LINUX) || (defined(WIN32)))
00116     #define x86
00117 #endif
00118
00119 #if !defined(x86_64) && \
00120     (defined(x86) && (defined(KERNEL_64BIT) || defined(__x86_64__)))
00121     #define x86_64
00122 #endif
00123
```

```
00124 #if defined(x86_64)
00125     #define WD_CPU_SPEC " x86_64"
00126 #elif defined(ARM)
00127     #define WD_CPU_SPEC " ARM"
00128 #elif defined(ARM64)
00129     #define WD_CPU_SPEC " ARM64"
00130 #else
00131     #define WD_CPU_SPEC " X86"
00132 #endif
00133
00134 #if defined(WINNT)
00135     #define WD_FILE_FORMAT " sys"
00136 #elif defined(APPLE)
00137     #define WD_FILE_FORMAT " kext"
00138 #elif defined(LINUX)
00139     #define WD_FILE_FORMAT " ko"
00140 #endif
00141
00142 #if defined(KERNEL_64BIT)
00143     #define WD_DATA_MODEL " 64bit"
00144 #else
00145     #define WD_DATA_MODEL " 32bit"
00146 #endif
00147
00148 #define WD_VER_STR WD_PROD_NAME " v" WD_VERSION_STR \
00149     " Jungs Connectivity (c) 1997 - " COPYRIGHTS_YEAR_STR \
00150     " Build Date: " __DATE__ \
00151 WD_CPU_SPEC WD_DATA_MODEL WD_FILE_FORMAT
00152
00153 #if !defined(POSIX) && defined(LINUX)
00154     #define POSIX
00155 #endif
00156
00157 #if !defined(UNIX) && defined(POSIX)
00158     #define UNIX
00159 #endif
00160
00161 #if !defined(WIN32) && defined(WINNT)
00162     #define WIN32
00163 #endif
00164
00165 #if !defined(WIN32) && !defined(UNIX) && !defined(APPLE)
00166     #define WIN32
00167 #endif
00168
00169 #if defined(_KERNEL_MODE) && !defined(KERNEL)
00170     #define KERNEL
00171 #endif
00172
00173 #if defined(KERNEL) && !defined(__KERNEL__)
00174     #define __KERNEL__
00175 #endif
00176
00177 #if defined(_KERNEL) && !defined(__KERNEL__)
00178     #define __KERNEL__
00179 #endif
00180
00181 #if defined(__KERNEL__) && !defined(_KERNEL)
00182     #define _KERNEL
00183 #endif
00184
00185 #if defined(LINUX) && defined(__x86_64__) && !defined(__KERNEL__)
00186     /* This fixes binary compatibility with older version of GLIBC
00187      * (64bit only) */
00188     __asm__(".symver memcpy,memcpy@GLIBC_2.2.5");
00189 #endif
00190
00191 #if defined(UNIX)
00192
00193     #if !defined(__P_TYPES__)
00194         #define __P_TYPES__
00195         #include <wd_types.h>
00196
00197         typedef void VOID;
00198         typedef unsigned char UCHAR;
00199         typedef unsigned short USHORT;
00200         typedef unsigned int UINT;
00201         #if !defined(APPLE_USB)
00202             typedef unsigned long ULONG;
00203         #endif
00204         typedef u32 BOOL;
00205         typedef void *PVOID;
00206         typedef unsigned char *PBYTE;
00207         typedef char CHAR;
00208         typedef char *PCHAR;
00209         typedef unsigned short *PWORD;
00210     
```

```
00211     typedef u32 DWORD, *PDWORD;
00212     typedef PVOID HANDLE;
00213 #endif
00214 #if !defined(__KERNEL__)
00215     #include <string.h>
00216     #include <ctype.h>
00217     #include <stdlib.h>
00218 #endif
00219 #ifndef TRUE
00220     #define TRUE 1
00221 #endif
00222 #ifndef FALSE
00223     #define FALSE 0
00224 #endif
00225 #define __cdecl
00226 #define WINAPI
00227
00228 #if defined(__KERNEL__)
00229     #if defined(LINUX)
00230         /* For _IO macros and for mapping Linux status codes
00231          * to WD status codes */
00232         #include <asm-generic/ioctl.h>
00233         #include <asm-generic/errno.h>
00234     #endif
00235 #else
00236     #include <unistd.h>
00237     #if defined(LINUX)
00238         #include <sys/ioctl.h> /* for BSD ioctl() */
00239         #include <sys/mman.h>
00240     #endif
00241     #include <sys/types.h>
00242     #include <sys/stat.h>
00243     #include <fcntl.h>
00244 #endif
00245     typedef unsigned long long UINT64;
00246 #if defined(APPLE)
00247     #include <libkern/OSTypes.h>
00248     #include <libkern/OSAtomic.h>
00249     typedef UInt16 UInt16;
00250     typedef UInt8 UInt8;
00251     typedef UInt8 u8;
00252     typedef UInt16 u16;
00253     typedef UInt32 u32;
00254     #include <IOKit/IOTypes.h>
00255 #if defined(__KERNEL__)
00256     typedef struct OSObject* FILEHANDLE;
00257     typedef UInt32 PRHANDLE;
00258 #else
00259     #include <IOKit/IOKitLib.h>
00260 #endif
00261 enum {
00262     kWinDriverMethodSyncIoctl = 0,
00263     kWinDriverNumOfMethods,
00264 };
00265 #if !defined(__KERNEL__)
00266     #include <string.h>
00267     #include <ctype.h>
00268 #endif
00269 #ifndef TRUE
00270     #define TRUE true
00271 #endif
00272 #ifndef FALSE
00273     #define FALSE false
00274 #endif
00275 #define __cdecl
00276 #define WINAPI
00277 #ifdef __LP64__
00278     #define PTR2INT(value) ((UInt64)(value))
00279 #else
00280     #define PTR2INT(value) ((UInt32)(value))
00281 #endif
00282 #define MAC_UC_MAGIC_MASK 0xFF0000FF
00283 #define MAC_UC_MAGIC_NUM 0x7E000041
00284 #define MAC_UC_64BIT_APP 0x100
00285 #endif
00286
00287 #elif defined(WIN32)
00288     #if defined(__KERNEL__)
00289         #if !defined(CMAKE_WD_BUILD)
00290             int sprintf(char *buffer, const char *format, ...);
00291         #endif
00292     #else
00293         #include <windows.h>
00294         #include <winiocrtl.h>
00295     #endif
00296     #define strcmp _strcmp
00297     typedef unsigned __int64 UInt64;
```

```
00298 #endif
00299
00300 #if !defined(__KERNEL__)
00301     #include <stdarg.h>
00302     #if !defined(va_copy) && !defined(__va_copy)
00303         #define va_copy(ap2,ap1) (ap2)=(ap1)
00304     #endif
00305     #if !defined(va_copy) && defined(__va_copy)
00306         #define va_copy __va_copy
00307     #endif
00308 #endif
00309
00310 #ifndef WINAPI
00311     #define WINAPI
00312 #endif
00313
00314 #if !defined(_WINDEF_)
00315     typedef unsigned char BYTE;
00316     typedef unsigned short int WORD;
00317 #endif
00318
00319 #if !defined(_BASSETSD_H_)
00320     typedef unsigned int UINT32;
00321 #endif
00322
00323
00324 #if defined(UNIX)
00325     #define PRI64      "ll"
00326 #elif defined(WIN32)
00327     #define PRI64      "I64"
00328 #endif
00329
00330
00331 #if defined(KERNEL_64BIT)
00332     #define KPRI PRI64
00333     #if defined(WIN32)
00334         #define UPRI KPRI
00335     #else
00336         #define UPRI "l"
00337     #endif
00338 #else
00339     #define KPRI ""
00340     #define UPRI "l"
00341 #endif
00342
00343 /*
00344 * The KPTR is guaranteed to be the same size as a kernel-mode pointer
00345 * The UPTR is guaranteed to be the same size as a user-mode pointer
00346 */
00347
00348 #if defined(KERNEL_64BIT)
00349     typedef UINT64 KPTR;
00350 #else
00351     typedef UINT32 KPTR;
00352 #endif
00353
00354 #if defined(UNIX)
00355     typedef unsigned long UPTR;
00356 #else
00357     typedef size_t UPTR;
00358 #endif
00359
00360 typedef UINT64 DMA_ADDR;
00361 typedef UINT64 PHYS_ADDR;
00362
00363 #include "windrvr_usb.h"
00364
00365 typedef enum
00366 {
00367     CMD_NONE = 0,
00368     CMD_END = 1,
00369     CMD_MASK = 2,
00370     RP_BYTE = 10,
00371     RP_WORD = 11,
00372     RP_DWORD = 12,
00373     WP_BYTE = 13,
00374     WP_WORD = 14,
00375     WP_DWORD = 15,
00376     RP_QWORD = 16,
00377     WP_QWORD = 17,
00378     RP_SBYTE = 20,
00379     RP_SWORD = 21,
00380     RP_SDWORD = 22,
00381     WP_SBYTE = 23,
00382     WP_SWORD = 24,
00383     WP_SDWORD = 25,
00384     RP_SQWORD = 26,
00385     WP_SQWORD = 27,
00386     RM_BYTE = 30,
```

```
00392     RM_WORD = 31,
00393     RM_DWORD = 32,
00394     WM_BYTE = 33,
00395     WM_WORD = 34,
00396     WM_DWORD = 35,
00397     RM_QWORD = 36,
00398     WM_QWORD = 37,
00399     RM_SBYTE = 40,
00400     RM_SWORD = 41,
00401     RM_SDWORD = 42,
00402     WM_SBYTE = 43,
00403     WM_SWORD = 44,
00404     WM_SDWORD = 45,
00405     RM_SQWORD = 46,
00406     WM_SQWORD = 47
00407 } WD_TRANSFER_CMD;
00408
00409 enum { WD_DMA_PAGES = 256 };
00410
00411 #ifndef DMA_BIT_MASK
00412     #define DMA_BIT_MASK(n) (((n) == 64) ? ~0ULL : ((1ULL<<(n))-1))
00413
00414 #endif
00415
00416 typedef enum {
00417     DMA_KERNEL_BUFFER_ALLOC = 0x1,
00420     DMA_KBUF_BELOW_16M = 0x2,
00423     DMA_LARGE_BUFFER = 0x4,
00429     DMA_ALLOW_CACHE = 0x8,
00431     DMA_KERNEL_ONLY_MAP = 0x10,
00434     DMA_FROM_DEVICE = 0x20,
00437     DMA_TO_DEVICE = 0x40,
00439     DMA_TO_FROM_DEVICE = (DMA_FROM_DEVICE | DMA_TO_DEVICE),
00442     DMA_ALLOW_64BIT_ADDRESS = 0x80,
00445     DMA_ALLOW_NO_HCARD = 0x100,
00447     DMA_GET_EXISTING_BUF = 0x200,
00449     DMA_RESERVED_MEM = 0x400,
00450
00451     DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH = 0x800,
00456     DMA_GET_PREALLOCATED_BUFFERS_ONLY = 0x1000,
00460     DMA_TRANSACTION = 0x2000,
00463     DMA_GPUREDIRECT = 0x4000,
00465     DMA_DISABLE_MERGE_ADJACENT_PAGES = 0x8000,
00469 } WD_DMA_OPTIONS;
00470
00471 #define DMA_ADDRESS_WIDTH_MASK 0x7f000000
00472
00473 #define DMA_OPTIONS_ALL \
00474     (DMA_KERNEL_BUFFER_ALLOC | DMA_KBUF_BELOW_16M | DMA_LARGE_BUFFER \
00475     | DMA_ALLOW_CACHE | DMA_KERNEL_ONLY_MAP | DMA_FROM_DEVICE | DMA_TO_DEVICE \
00476     | DMA_ALLOW_64BIT_ADDRESS | DMA_ALLOW_NO_HCARD | DMA_GET_EXISTING_BUF \
00477     | DMA_RESERVED_MEM | DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH \
00478     | DMA_ADDRESS_WIDTH_MASK)
00479
00480 #define DMA_DIRECTION_MASK DMA_TO_FROM_DEVICE
00481
00483 #define DMA_READ_FROM_DEVICE DMA_FROM_DEVICE
00484 #define DMA_WRITE_TO_DEVICE DMA_TO_DEVICE
00485
00486 #define DMA_OPTIONS_ADDRESS_WIDTH_SHIFT 24
00491 enum {
00492     WD_MATCH_EXCLUDE = 0x1
00493 };
00494
00495 /* Use it to pad struct size to 64 bit, when using 32 on 64 bit applications */
00496 #if defined(i386) && defined(KERNEL_64BIT)
00497     #define PAD_TO_64(pName) DWORD dwPad_##pName;
00498 #else
00499     #define PAD_TO_64(pName)
00500 #endif
00501
00502 typedef struct
00503 {
00504     DMA_ADDR pPhysicalAddr;
00505     DWORD dwBytes;
00506     PAD_TO_64(dwBytes)
00507 } WD_DMA_PAGE, WD_DMA_PAGE_V80;
00508
00509 typedef void (DLLCALLCONV * DMA_TRANSACTION_CALLBACK)(PVOID pData);
00510
00511 typedef struct
00512 {
00513     DWORD hDma;
00514     PAD_TO_64(hDma)
00515
00516     PVOID pUserAddr;
00517     PAD_TO_64(pUserAddr)
00518 }
```

```
00519     KPTR pKernelAddr;
00520     DWORD dwBytes;
00521     DWORD dwOptions;
00524     DWORD dwPages;
00525     DWORD hCard;
00528     /* Windows: The following 6 parameters are used for DMA transaction only */
00529
00530     DMA_TRANSACTION_CALLBACK DMATransactionCallback;
00531     PAD_TO_64(DMATransactionCallback)
00532
00533     PVOID DMATransactionCallbackCtx;
00534     PAD_TO_64(DMATransactionCallbackCtx)
00535
00536     DWORD dwAlignment;
00537     DWORD dwMaxTransferSize;
00538     DWORD dwTransferElementsSize;
00539     DWORD dwBytesTransferred;
00541     WD_DMA_PAGE Page[WD_DMA_PAGES];
00542 } WD_DMA, WD_DMA_V80;
00543
00544 typedef enum {
00545     /* KER_BUF_ALLOC_NON_CONTIG and KER_BUF_GET_EXISTING_BUF options are valid
00546     * only as part of "WinDriver for Server" API and require
00547     * "WinDriver for Server" license.
00548     * @note "WinDriver for Server" APIs are included in WinDriver evaluation
00549     * version. */
00550     KER_BUF_ALLOC_NON_CONTIG = 0x0001,
00551     KER_BUF_ALLOC_CONTIG      = 0x0002,
00552     KER_BUF_ALLOC_CACHED      = 0x0004,
00553     KER_BUF_GET_EXISTING_BUF = 0x0008,
00554 } WD_KER_BUF_OPTION;
00555
00556 typedef struct
00557 {
00558     DWORD hKerBuf;
00559     DWORD dwOptions;
00560     UINT64 qwBytes;
00561     KPTR pKernelAddr;
00562     UPTR pUserAddr;
00563     PAD_TO_64(pUserAddr)
00564 } WD_KERNEL_BUFFER, WD_KERNEL_BUFFER_V121;
00565
00566 typedef struct
00567 {
00568     KPTR pPort;
00569     DWORD cmdTrans;
00571     /* Parameters used for string transfers: */
00572     DWORD dwBytes;
00573     DWORD fAutoinc;
00575     DWORD dwOptions;
00576     union
00577     {
00578         BYTE Byte;
00579         WORD Word;
00580         UINT32 Dword;
00581         UINT64 Qword;
00582         PVOID pBuffer;
00583     } Data;
00584 } WD_TRANSFER, WD_TRANSFER_V61;
00585
00586 enum {
00587     INTERRUPT_LATCHED = 0x00,
00588     INTERRUPT_LEVEL_SENSITIVE = 0x01,
00590     INTERRUPT_CMD_COPY = 0x02,
00595     INTERRUPT_CE_INT_ID = 0x04,
00596     INTERRUPT_CMD_RETURN_VALUE = 0x08,
00597     INTERRUPT_MESSAGE = 0x10,
00598     INTERRUPT_MESSAGE_X = 0x20,
00600     INTERRUPT_DONT_GET_MSI_MESSAGE = 0x40
00602 };
00603
00604 typedef struct
00605 {
00606     DWORD hKernelPlugIn;
00607     DWORD dwMessage;
00608     PVOID pData;
00609     PAD_TO_64(pData)
00610     DWORD dwResult;
00611     PAD_TO_64(dwResult)
00612 } WD_KERNEL_PLUGIN_CALL, WD_KERNEL_PLUGIN_CALL_V40;
00613
00614 typedef enum {
00615     INTERRUPT_RECEIVED = 0,
00616     INTERRUPT_STOPPED,
00617     INTERRUPT_INTERRUPTED
00619 } WD_INTERRUPT_WAIT_RESULT;
00620
```

```
00621 typedef struct
00622 {
00623     DWORD hInterrupt;
00624     DWORD dwOptions;
00625     WD_TRANSFER *Cmd;
00626     PAD_TO_64(Cmd)
00627     DWORD dwCmds;
00628     PAD_TO_64(dwCmds)
00629
00630     /* For WD_IntEnable(): */
00631     WD_KERNEL_PLUGIN_CALL kpCall;
00632     DWORD fEnableOk;
00633
00634     /* For WD_IntWait() and WD_IntCount(): */
00635     DWORD dwCounter;
00636     DWORD dwLost;
00637     DWORD fStopped;
00638     DWORD dwLastMessage;
00639     DWORD dwEnabledIntType;
00640 } WD_INTERRUPT, WD_INTERRUPT_V91;
00641
00642 typedef struct
00643 {
00644     DWORD dwVer;
00645     CHAR cVer[WD_VERSION_STR_LENGTH];
00646 } WD_VERSION, WD_VERSION_V30;
00647
00648 #define WD_LICENSE_LENGTH 3072
00649 typedef struct
00650 {
00651     CHAR cLicense[WD_LICENSE_LENGTH];
00652 } WD_LICENSE, WD_LICENSE_V122;
00653
00654 enum
00655 {
00656     WD_BUS_USB = (int)0xffffffff,
00657     WD_BUS_UNKNOWN = 0,
00658     WD_BUS_ISA = 1,
00659     WD_BUS_EISA = 2,
00660     WD_BUS_PCI = 5,
00661 };
00662 typedef DWORD WD_BUS_TYPE;
00663
00664 typedef struct
00665 {
00666     WD_BUS_TYPE dwBusType;
00667     DWORD dwDomainNum;
00668     DWORD dwBusNum;
00669     DWORD dwSlotFunc;
00670 } WD_BUS, WD_BUS_V30;
00671
00672 typedef enum
00673 {
00674     ITEM_NONE      = 0,
00675     ITEM_INTERRUPT = 1,
00676     ITEM_MEMORY    = 2,
00677     ITEM_IO         = 3,
00678     ITEM_BUS        = 5
00679 } ITEM_TYPE;
00680
00681 typedef enum
00682 {
00683     WD_ITEM_MEM_DO_NOT_MAP_KERNEL = 0x1,
00684     WD_ITEM_MEM_ALLOW_CACHE = 0x2,
00685     WD_ITEM_MEM_USER_MAP = 0x4,
00686 } WD_ITEM_MEM_OPTIONS;
00687
00688 typedef struct
00689 {
00690     DWORD item;
00691     DWORD fNotSharable;
00692     union
00693     {
00694         struct
00695         {
00696             PHYS_ADDR pPhysicalAddr;
00697             UINT64 qwBytes;
00698             KPTR pTransAddr;
00699             UPTR pUserDirectAddr;
00700             PAD_TO_64(pUserDirectAddr)
00701             DWORD dwBar;
00702             DWORD dwOptions;
00703             KPTR pReserved;
00704         } Mem;
00705
00706         struct
00707         {
00708             KPTR pAddr;
00709         } Page;
00710     } u;
00711 }
```

```
00724             DWORD dwBytes;
00725             DWORD dwBar;
00726         } IO;
00727
00729     struct
00730     {
00731         DWORD dwInterrupt;
00732         DWORD dwOptions;
00733         DWORD hInterrupt;
00734         DWORD dwReserved1;
00735         KPTR pReserved2;
00736     } Int;
00737     WD_BUS Bus;
00738 } I;
00739 } WD_ITEMS, WD_ITEMS_V118;
00740
00741 enum { WD_CARD_ITEMS = 128 };
00742
00743 typedef struct
00744 {
00745     DWORD dwItems;
00746     PAD_TO_64(dwItems)
00747     WD_ITEMS Item[WD_CARD_ITEMS];
00748 } WD_CARD, WD_CARD_V118;
00749
00750 typedef struct
00751 {
00752     WD_CARD Card;
00753     DWORD fCheckLockOnly;
00754     DWORD hCard;
00755     DWORD dwOptions;
00756     CHAR cName[32];
00757     CHAR cDescription[100];
00758 } WD_CARD_REGISTER, WD_CARD_REGISTER_V118;
00759
00760 #define WD_PROCESS_NAME_LENGTH 128
00761 typedef struct
00762 {
00763     CHAR cProcessName[WD_PROCESS_NAME_LENGTH];
00764     DWORD dwSubGroupID;
00765     DWORD dwGroupID;
00766     DWORD hIpc;
00767 } WD_IPC_PROCESS, WD_IPC_PROCESS_V121;
00768
00769 typedef struct
00770 {
00771     WD_IPC_PROCESS procInfo;
00772     DWORD dwOptions;
00773 } WD_IPC_REGISTER, WD_IPC_REGISTER_V121;
00774
00775 enum { WD_IPC_MAX_PROCS = 0x40 };
00776
00777 typedef struct
00778 {
00779     WD_IPC_PROCESS procInfo;
00780     DWORD dwOptions;
00781 } WD_IPC_REGISTER, WD_IPC_REGISTER_V121;
00782
00783 enum { WD_IPC_MAX_PROCS = 0x40 };
00784
00785 typedef struct
00786 {
00787     DWORD hIpc;
00788     DWORD dwNumProcs;
00789     WD_IPC_PROCESS procInfo[WD_IPC_MAX_PROCS];
00790 } WD_IPC_SCAN_PROCS, WD_IPC_SCAN_PROCS_V121;
00791
00792 enum
00793 {
00794     WD_IPC_UID_UNICAST = 0x1,
00795     WD_IPC_SUBGROUP_MULTICAST = 0x2,
00796     WD_IPC_MULTICAST = 0x4,
00797 };
00798
00799
00800 };
00801
00802 typedef struct
00803 {
00804     DWORD hIpc;
00805     DWORD dwOptions;
00806     DWORD dwRecipientID;
00807     DWORD dwMsgID;
00808     UINT64 qwMsgData;
00809 } WD_IPC_SEND, WD_IPC_SEND_V121;
00810
00811
00812
00813 typedef struct
00814 {
00815     DWORD hCard;
00816     PAD_TO_64(hCard)
00817     WD_TRANSFER *Cmd;
00818     PAD_TO_64(Cmd)
00819     DWORD dwCmds;
00820     DWORD dwOptions;
00821 } WD_CARD_CLEANUP;
00822
00823 enum { WD_FORCE_CLEANUP = 0x1 };
00824
```

```
00825 enum { WD_PCI_CARDS = 256 };
00827 typedef struct
00828 {
00829     DWORD dwDomain;
00831     DWORD dwBus;
00832     DWORD dwSlot;
00833     DWORD dwFunction;
00834 } WD_PCI_SLOT;
00835
00836 typedef struct
00837 {
00838     DWORD dwVendorId;
00839     DWORD dwDeviceId;
00840 } WD_PCI_ID;
00841
00842 typedef struct
00843 {
00845     WD_PCI_ID searchId;
00848     DWORD dwCards;
00851     WD_PCI_ID cardId[WD_PCI_CARDS];
00852     WD_PCI_SLOT cardSlot[WD_PCI_CARDS];
00855     DWORD dwOptions;
00856 } WD_PCI_SCAN_CARDS, WD_PCI_SCAN_CARDS_V124;
00857
00858 typedef enum {
00859     WD_PCI_SCAN_DEFAULT = 0x0,
00860     WD_PCI_SCAN_BY_TOPOLOGY = 0x1,
00861     WD_PCI_SCAN_REGISTERED = 0x2,
00862     WD_PCI_SCAN_INCLUDE_DOMAINS = 0x4,
00863 } WD_PCI_SCAN_OPTIONS;
00864
00865 enum { WD_PCI_MAX_CAPS = 50 };
00866
00867 enum { WD_PCI_CAP_ID_ALL = 0x0 };
00868
00869 typedef struct
00870 {
00871     DWORD dwCapId;
00872     DWORD dwCapOffset;
00873 } WD_PCI_CAP;
00874
00875 typedef enum {
00876     WD_PCI_SCAN_CAPS_BASIC = 0x1,
00877     WD_PCI_SCAN_CAPS_EXTENDED = 0x2
00879 } WD_PCI_SCAN_CAPS_OPTIONS;
00880
00881 typedef struct
00882 {
00884     WD_PCI_SLOT pciSlot;
00885     DWORD dwCapId;
00887     DWORD dwOptions;
00891     DWORD dwNumCaps;
00892     WD_PCI_CAP pciCaps[WD_PCI_MAX_CAPS];
00894 } WD_PCI_SCAN_CAPS, WD_PCI_SCAN_CAPS_V118;
00895
00896 typedef struct
00897 {
00898     WD_PCI_SLOT pciSlot;
00899     DWORD dwNumVFs;
00900 } WD_PCI_SRIOV, WD_PCI_SRIOV_V122;
00901
00902 typedef struct
00903 {
00904     WD_PCI_SLOT pciSlot;
00905     WD_CARD Card;
00906 } WD_PCI_CARD_INFO, WD_PCI_CARD_INFO_V118;
00907
00908 typedef enum
00909 {
00910     PCI_ACCESS_OK = 0,
00911     PCI_ACCESS_ERROR = 1,
00912     PCI_BAD_BUS = 2,
00913     PCI_BAD_SLOT = 3
00914 } PCI_ACCESS_RESULT;
00915
00916 typedef struct
00917 {
00918     WD_PCI_SLOT pciSlot;
00919     PVOID pBuffer;
00920     PAD_TO_64(pBuffer)
00921     DWORD dwOffset;
00923     DWORD dwBytes;
00925     DWORD fIsRead;
00926     DWORD dwResult;
00927 } WD_PCI_CONFIG_DUMP, WD_PCI_CONFIG_DUMP_V30;
00928
```

```
00929 enum { SLEEP_BUSY = 0, SLEEP_NON_BUSY = 1 };
00930 typedef struct
00931 {
00932     DWORD dwMicroSeconds;
00934     DWORD dwOptions;
00935 } WD_SLEEP, WD_SLEEP_V40;
00936
00937 typedef enum
00938 {
00939     D_OFF      = 0,
00940     D_ERROR    = 1,
00941     D_WARN     = 2,
00942     D_INFO     = 3,
00943     D_TRACE    = 4
00944 } DEBUG_LEVEL;
00945
00946 typedef enum
00947 {
00948     S_ALL      = (int)0xffffffff,
00949     S_IO       = 0x00000008,
00950     S_MEM      = 0x00000010,
00951     S_INT      = 0x00000020,
00952     S_PCI      = 0x00000040,
00953     S_DMA      = 0x00000080,
00954     S_MISC     = 0x00000100,
00955     S_LICENSE  = 0x00000200,
00956     S_PNP      = 0x00001000,
00957     S_CARD_REG = 0x00002000,
00958     S_KER_DRV  = 0x00004000,
00959     S_USB      = 0x00008000,
00960     S_KER_PLUG = 0x00010000,
00961     S_EVENT    = 0x00020000,
00962     S_IPC      = 0x00040000,
00963     S_KER_BUF  = 0x00080000,
00964 } DEBUG_SECTION;
00965
00966 typedef enum
00967 {
00968     DEBUG_STATUS = 1,
00969     DEBUG_SET_FILTER = 2,
00970     DEBUG_SET_BUFFER = 3,
00971     DEBUG_CLEAR_BUFFER = 4,
00972     DEBUG_DUMP_SEC_ON = 5,
00973     DEBUG_DUMP_SEC_OFF = 6,
00974     KERNEL_DEBUGGER_ON = 7,
00975     KERNEL_DEBUGGER_OFF = 8,
00976     DEBUG_DUMP_CLOCK_ON = 9,
00977     DEBUG_DUMP_CLOCK_OFF = 10,
00978     DEBUG_CLOCK_RESET = 11
00979 } DEBUG_COMMAND;
00980
00981 typedef struct
00982 {
00983     DWORD dwCmd;
00985 /* used for DEBUG_SET_FILTER */
00986     DWORD dwLevel;
00988     DWORD dwSection;
00990     DWORD dwLevelMessageBox;
00991 /* used for DEBUG_SET_BUFFER */
00992     DWORD dwBufferSize;
00993 } WD_DEBUG, WD_DEBUG_V40;
00994
00995 #define DEBUG_USER_BUF_LEN 2048
00996 typedef struct
00997 {
00998     CHAR cBuffer[DEBUG_USER_BUF_LEN];
00999 } WD_DEBUG_DUMP, WD_DEBUG_DUMP_V40;
01000
01001 typedef struct
01002 {
01003     CHAR pcBuffer[256];
01004     DWORD dwLevel;
01005     DWORD dwSection;
01006 } WD_DEBUG_ADD, WD_DEBUG_ADD_V503;
01007
01008 typedef struct
01009 {
01010     DWORD hKernelPlugIn;
01011     CHAR cDriverName[WD_MAX_KP_NAME_LENGTH];
01012     CHAR cDriverPath[WD_MAX_KP_NAME_LENGTH];
01016     PAD_TO_64(hKernelPlugIn) /* 64 bit app as a 4 byte hole here */
01017     PVOID pOpenData;
01018     PAD_TO_64(pOpenData)
01019 } WD_KERNEL_PLUGIN, WD_KERNEL_PLUGIN_V40;
01020
01022 typedef enum
01023 {
```

```
01024     WD_DEVICE_PCI = 0x1,
01025     WD_DEVICE_USB = 0x2
01026 } WD_GET_DEVICE_PROPERTY_OPTION;
01027
01028 typedef struct
01029 {
01030     union
01031     {
01032         HANDLE hDevice;
01033         PAD_TO_64(hDevice)
01034         DWORD dwUniqueID;
01035     } h;
01036     PVOID pBuf;
01037     PAD_TO_64(pBuf)
01038     DWORD dwBytes;
01039     DWORD dwProperty;
01040     DWORD dwOptions;
01041 } WD_GET_DEVICE_PROPERTY;
01042
01043 typedef enum
01044 {
01045     WD_STATUS_SUCCESS = 0,
01046     WD_STATUS_INVALID_WD_HANDLE = (int)0xffffffff,
01047     WD_WINDRIVER_STATUS_ERROR = 0x20000000L,
01048     WD_INVALID_HANDLE = 0x20000001L,
01049     WD_INVALID_PIPE_NUMBER = 0x20000002L,
01050     WD_READ_WRITE_CONFLICT = 0x20000003L,
01051     WD_ZERO_PACKET_SIZE = 0x20000004L,
01052     WD_INSUFFICIENT_RESOURCES = 0x20000005L,
01053     WD_UNKNOWN_PIPE_TYPE = 0x20000006L,
01054     WD_SYSTEM_INTERNAL_ERROR = 0x20000007L,
01055     WD_DATA_MISMATCH = 0x20000008L,
01056     WD_NO_LICENSE = 0x20000009L,
01057     WD_NOT_IMPLEMENTED = 0x2000000AL,
01058     WD_KERPLUG_FAILURE = 0x2000000BL,
01059     WD_FAILED_ENABLING_INTERRUPT = 0x2000000cL,
01060     WD_INTERRUPT_NOT_ENABLED = 0x2000000dL,
01061     WD_RESOURCE_OVERLAP = 0x2000000eL,
01062     WD_DEVICE_NOT_FOUND = 0x2000000fL,
01063     WD_WRONG_UNIQUE_ID = 0x20000010L,
01064     WD_OPERATION_ALREADY_DONE = 0x20000011L,
01065     WD_USB_DESCRIPTOR_ERROR = 0x20000012L,
01066     WD_SET_CONFIGURATION_FAILED = 0x20000013L,
01067     WD_CANT_OBTAIN_PDO = 0x20000014L,
01068     WD_TIME_OUT_EXPIRED = 0x20000015L,
01069     WD_IRP_CANCELED = 0x20000016L,
01070     WD_FAILED_USER_MAPPING = 0x20000017L,
01071     WD_FAILED_KERNEL_MAPPING = 0x20000018L,
01072     WD_NO_RESOURCES_ON_DEVICE = 0x20000019L,
01073     WD_NO_EVENTS = 0x2000001aL,
01074     WD_INVALID_PARAMETER = 0x2000001bL,
01075     WD_INcorrect_VERSION = 0x2000001cL,
01076     WD_TRY AGAIN = 0x2000001dL,
01077     WD_WINDRIVER_NOT_FOUND = 0x2000001eL,
01078     WD_INVALID_IOCTL = 0x2000001fL,
01079     WD_OPERATION_FAILED = 0x20000020L,
01080     WD_INVALID_32BIT_APP = 0x20000021L,
01081     WD_TOO_MANY_HANDLES = 0x20000022L,
01082     WD_NO_DEVICE_OBJECT = 0x20000023L,
01083     WD_MORE_PROCESSING_REQUIRED = (int)0xC0000016L,
01084     /* The following status codes are returned by USBD:
01085     * USBD status types: */
01086     WD_USBD_STATUS_SUCCESS = 0x00000000L,
01087     WD_USBD_STATUS_PENDING = 0x40000000L,
01088     WD_USBD_STATUS_ERROR = (int)0x80000000L,
01089     WD_USBD_STATUS_HALTED = (int)0xC0000000L,
01090     /* USBD status codes: */
01091     /* @note The following status codes are comprised of one of the status
01092     * types above and an error code [i.e. 0xXYYYYYYY - where: X = status type;
01093     * YYYYYYY = error code].
01094     * The same error codes may also appear with one of the other status types
01095     * as well. */
01096
01097     /* HC (Host Controller) status codes.
01098     @note These status codes use the WD_USBD_STATUS_HALTED status type]: */
01099     WD_USBD_STATUS_CRC = (int)0xC0000001L,
01100     WD_USBD_STATUS_BTSTUFF = (int)0xC0000002L,
01101     WD_USBD_STATUS_DATA_TOGGLE_MISMATCH = (int)0xC0000003L,
01102     WD_USBD_STATUS_STALL_PID = (int)0xC0000004L,
01103     WD_USBD_STATUS_DEV_NOT_RESPONDING = (int)0xC0000005L,
01104     WD_USBD_STATUS_PID_CHECK_FAILURE = (int)0xC0000006L,
01105     WD_USBD_STATUS_UNEXPECTED_PID = (int)0xC0000007L,
01106     WD_USBD_STATUS_DATA_OVERRUN = (int)0xC0000008L,
01107     WD_USBD_STATUS_DATA_UNDERRUN = (int)0xC0000009L,
01108     WD_USBD_STATUS_RESERVED1 = (int)0xC000000AL,
01109     WD_USBD_STATUS_RESERVED2 = (int)0xC000000BL,
01110     WD_USBD_STATUS_BUFFER_OVERRUN = (int)0xC000000CL,
01111     WD_USBD_STATUS_BUFFER_UNDERRUN = (int)0xC000000DL,
```

```
01293     WD_USBD_STATUS_NOT_ACSESSED = (int)0xC000000FL,
01295     WD_USBD_STATUS_FIFO = (int)0xC0000010L,
01298 #if defined(WIN32)
01299     WD_USBD_STATUS_XACT_ERROR = (int)0xC0000011L,
01305     WD_USBD_STATUS_BABBLE_DETECTED = (int)0xC0000012L,
01308     WD_USBD_STATUS_DATA_BUFFER_ERROR = (int)0xC0000013L,
01311 #endif
01312
01313     WD_USBD_STATUS_CANCELED = (int)0xC0010000L,
01319     WD_USBD_STATUS_ENDPOINT_HALTED = (int)0xC0000030L,
01320
01321 /* Software status codes
01322 @note The following status codes have only the error bit set: */
01323     WD_USBD_STATUS_NO_MEMORY = (int)0x80000100L,
01326     WD_USBD_STATUS_INVALID_URB_FUNCTION = (int)0x80000200L,
01329     WD_USBD_STATUS_INVALID_PARAMETER = (int)0x80000300L,
01335     WD_USBD_STATUS_ERROR_BUSY = (int)0x80000400L,
01336
01341     WD_USBD_STATUS_REQUEST_FAILED = (int)0x80000500L,
01343     WD_USBD_STATUS_INVALID_PIPE_HANDLE = (int)0x80000600L,
01344
01347     WD_USBD_STATUS_NO_BANDWIDTH = (int)0x80000700L,
01348
01350     WD_USBD_STATUS_INTERNAL_HC_ERROR = (int)0x80000800L,
01351
01354     WD_USBD_STATUS_ERROR_SHORT_TRANSFER = (int)0x80000900L,
01355
01359     WD_USBD_STATUS_BAD_START_FRAME = (int)0xC0000A00L,
01360
01363     WD_USBD_STATUS_ISOCH_REQUEST_FAILED = (int)0xC0000B00L,
01364
01367     WD_USBD_STATUS_FRAME_CONTROL_OWNED = (int)0xC0000C00L,
01368
01371     WD_USBD_STATUS_FRAME_CONTROL_NOT_OWNED = (int)0xC0000D00L
01372
01373 #if defined(WIN32)
01374
01375 /* Additional USB 2.0 software error codes added for USB 2.0: */
01376     WD_USBD_STATUS_NOT_SUPPORTED = (int)0xC0000E00L,
01379     WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR = (int)0xC0000F00L,
01385     WD_USBD_STATUS_INSUFFICIENT_RESOURCES = (int)0xC0001000L,
01389     WD_USBD_STATUS_SET_CONFIG_FAILED = (int)0xC0002000L,
01394     WD_USBD_STATUS_BUFFER_TOO_SMALL = (int)0xC0003000L,
01398     WD_USBD_STATUS_INTERFACE_NOT_FOUND = (int)0xC0004000L,
01402     WD_USBD_STATUS_INAVLID_PIPE_FLAGS = (int)0xC0005000L,
01406     WD_USBD_STATUS_TIMEOUT = (int)0xC0006000L,
01409     WD_USBD_STATUS_DEVICE_GONE = (int)0xC0007000L,
01412     WD_USBD_STATUS_STATUS_NOT_MAPPED = (int)0xC0008000L,
01423     WD_USBD_STATUS_ISO_NOT_ACSESSED_BY_HW = (int)0xC0020000L,
01427     WD_USBD_STATUS_ISO_TD_ERROR = (int)0xC0030000L,
01430     WD_USBD_STATUS_ISO_NA_LATE_USBPORT = (int)0xC0040000L,
01433     WD_USBD_STATUS_ISO_NOT_ACSESSED_LATE = (int)0xC0050000L
01434 #endif
01435 } WD_ERROR_CODES;
01436
01437 typedef enum
01438 {
01439     WD_INSERT = 0x1,
01440     WD_REMOVE = 0x2,
01441     WD_OBSOLETE = 0x8,
01442     WD_POWER_CHANGED_D0 = 0x10,
01444     WD_POWER_CHANGED_D1 = 0x20,
01445     WD_POWER_CHANGED_D2 = 0x40,
01446     WD_POWER_CHANGED_D3 = 0x80,
01447     WD_POWER_SYSTEM_WORKING = 0x100,
01448     WD_POWER_SYSTEM_SLEEPING1 = 0x200,
01449     WD_POWER_SYSTEM_SLEEPING2 = 0x400,
01450     WD_POWER_SYSTEM_SLEEPING3 = 0x800,
01451     WD_POWER_SYSTEM_HIBERNATE = 0x1000,
01452     WD_POWER_SYSTEM_SHUTDOWN = 0x2000,
01453     WD_IPC_UNICAST_MSG = 0x4000,
01454     WD_IPC_MULTICAST_MSG = 0x8000,
01455 } WD_EVENT_ACTION;
01456
01457 #define WD_IPC_ALL_MSG (WD_IPC_UNICAST_MSG | WD_IPC_MULTICAST_MSG)
01458
01459 typedef enum
01460 {
01461     WD_ACKNOWLEDGE = 0x1,
01462     WD_ACCEPT_CONTROL = 0x2
01463 } WD_EVENT_OPTION;
01464
01465 #define WD_ACTIONS_POWER (WD_POWER_CHANGED_D0 | WD_POWER_CHANGED_D1 | \
01466     WD_POWER_CHANGED_D2 | WD_POWER_CHANGED_D3 | WD_POWER_SYSTEM_WORKING | \
01467     WD_POWER_SYSTEM_SLEEPING1 | WD_POWER_SYSTEM_SLEEPING3 | \
01468     WD_POWER_SYSTEM_HIBERNATE | WD_POWER_SYSTEM_SHUTDOWN)
01469 #define WD_ACTIONS_ALL (WD_ACTIONS_POWER | WD_INSERT | WD_REMOVE)
```

```
01470
01471 enum
01472 {
01473     WD_EVENT_TYPE_UNKNOWN = 0,
01474     WD_EVENT_TYPE_PCI      = 1,
01475     WD_EVENT_TYPE_USB      = 3,
01476     WD_EVENT_TYPE_IPC      = 4,
01477 };
01478 typedef DWORD WD_EVENT_TYPE;
01479
01480 typedef struct
01481 {
01482     DWORD hEvent;
01483     DWORD dwEventType;
01484     DWORD dwAction;
01485     DWORD dwEventId;
01486     DWORD hKernelPlugIn;
01487     DWORD dwOptions;
01488     union
01489     {
01490         struct
01491         {
01492             {
01493                 WD_PCI_ID cardId;
01494                 WD_PCI_SLOT pciSlot;
01495             } Pci;
01496             struct
01497             {
01498                 DWORD dwUniqueId;
01499             } Usb;
01500             struct
01501             {
01502                 DWORD hIpc;
01503                 DWORD dwSubGroupID;
01504                 DWORD dwGroupID;
01505
01506                 DWORD dwSenderUID;
01507                 DWORD dwMsgID;
01508                 PAD_TO_64(dwMsgID)
01509                 UINT64 qwMsgData;
01510             } Ipc;
01511         } u;
01512     };
01513     DWORD dwNumMatchTables;
01514     WDU_MATCH_TABLE matchTables[1];
01515 } WD_EVENT, WD_EVENT_V121;
01516
01517
01518 typedef struct
01519 {
01520     DWORD applications_num;
01521     DWORD devices_num;
01522 } WD_USAGE;
01523
01524 enum
01525 {
01526     WD_USB_HARD_RESET = 1,
01527     WD_USB_CYCLE_PORT = 2
01528 };
01529
01530 #ifndef BZERO
01531     #define BZERO(buf) memset(&(buf), 0, sizeof(buf))
01532 #endif
01533
01534 #ifndef INVALID_HANDLE_VALUE
01535     #define INVALID_HANDLE_VALUE ((HANDLE)(-1))
01536 #endif
01537
01538 #ifndef CTL_CODE
01539     #define CTL_CODE(DeviceType, Function, Method, Access) ( \
01540         ((DeviceType)<<16) | ((Access)<<14) | ((Function)<<2) | (Method) \
01541     )
01542
01543     #define METHOD_BUFFERED    0
01544     #define METHOD_IN_DIRECT   1
01545     #define METHOD_OUT_DIRECT  2
01546     #define METHOD_NEITHER     3
01547     #define FILE_ANY_ACCESS    0
01548     #define FILE_READ_ACCESS   1
01549     #define FILE_WRITE_ACCESS  2
01550 #endif
01551
01552 #if defined(LINUX) && defined(KERNEL_64BIT)
01553     #define WD_TYPE 0
01554     #define WD_CTL_CODE(wFuncNum) \
01555         _IOC(_IOC_READ|_IOC_WRITE, WD_TYPE, wFuncNum, 0)
01556     #define WD_CTL_DECODE_FUNC(IoControlCode) _IOC_NR(IoControlCode)
01557     #define WD_CTL_DECODE_TYPE(IoControlCode) _IOC_TYPE(IoControlCode)
01558 #elif defined(UNIX)
```

```
01559     #define WD_TYPE 0
01560     #define WD_CTL_CODE(wFuncNum)  (wFuncNum)
01561     #define WD_CTL_DECODE_FUNC(IoControlCode)  (IoControlCode)
01562     #define WD_CTL_DECODE_TYPE(IoControlCode)  (WD_TYPE)
01563 #else
01564     #define WD_TYPE 38200
01565     #if defined(KERNEL_64BIT)
01566         #define FUNC_MASK 0x400
01567     #else
01568         #define FUNC_MASK 0x0
01569     #endif
01570     #define WD_CTL_CODE(wFuncNum) CTL_CODE(WD_TYPE, (wFuncNum | FUNC_MASK), \
01571             METHOD_NEITHER, FILE_ANY_ACCESS)
01572     #define WD_CTL_DECODE_FUNC(IoControlCode) ((IoControlCode >> 2) & 0xffff)
01573     #define WD_CTL_DECODE_TYPE(IoControlCode) \
01574             DEVICE_TYPE_FROM_CTL_CODE(IoControlCode)
01575
01576 #endif
01577
01578 #if defined(LINUX)
01579     #define WD_CTL_IS_64BIT_AWARE(IoControlCode) \
01580             (_IOC_DIR(IoControlCode) & (_IOC_READ|_IOC_WRITE))
01581 #elif defined(UNIX)
01582     #define WD_CTL_IS_64BIT_AWARE(IoControlCode) TRUE
01583 #else
01584     #define WD_CTL_IS_64BIT_AWARE(IoControlCode) \
01585             (WD_CTL_DECODE_FUNC(IoControlCode) & FUNC_MASK)
01586 #endif
01587
01588 /* WinDriver function IOCTL calls. For details on the WinDriver functions, */
01589 /* see the WinDriver manual or included help files. */
01590
01591 #define IOCTL_WD_KERNEL_BUF_LOCK          WD_CTL_CODE(0x9f3)
01592 #define IOCTL_WD_KERNEL_BUF_UNLOCK        WD_CTL_CODE(0x9f4)
01593 #define IOCTL_WD_DMA_LOCK                WD_CTL_CODE(0x9be)
01594 #define IOCTL_WD_DMA_UNLOCK              WD_CTL_CODE(0x902)
01595 #define IOCTL_WD_TRANSFER               WD_CTL_CODE(0x98c)
01596 #define IOCTL_WD_MULTI_TRANSFER         WD_CTL_CODE(0x98d)
01597 #define IOCTL_WD_PCI_SCAN_CARDS         WD_CTL_CODE(0x9fa)
01598 #define IOCTL_WD_PCI_GET_CARD_INFO      WD_CTL_CODE(0x9e8)
01599 #define IOCTL_WD_VERSION                WD_CTL_CODE(0x910)
01600 #define IOCTL_WD_PCI_CONFIG_DUMP        WD_CTL_CODE(0x91a)
01601 #define IOCTL_WD_KERNEL_PLUGIN_OPEN     WD_CTL_CODE(0x91b)
01602 #define IOCTL_WD_KERNEL_PLUGIN_CLOSE    WD_CTL_CODE(0x91c)
01603 #define IOCTL_WD_KERNEL_PLUGIN_CALL     WD_CTL_CODE(0x91d)
01604 #define IOCTL_WD_INT_ENABLE             WD_CTL_CODE(0x9b6)
01605 #define IOCTL_WD_INT_DISABLE            WD_CTL_CODE(0x9bb)
01606 #define IOCTL_WD_INT_COUNT              WD_CTL_CODE(0x9ba)
01607 #define IOCTL_WD_SLEEP                 WD_CTL_CODE(0x927)
01608 #define IOCTL_WD_DEBUG                 WD_CTL_CODE(0x928)
01609 #define IOCTL_WD_DEBUG_DUMP             WD_CTL_CODE(0x929)
01610 #define IOCTL_WD_CARD_UNREGISTER       WD_CTL_CODE(0x9e7)
01611 #define IOCTL_WD_CARD_REGISTER          WD_CTL_CODE(0x9e6)
01612 #define IOCTL_WD_INT_WAIT              WD_CTL_CODE(0x9b9)
01613 #define IOCTL_WD_LICENSE               WD_CTL_CODE(0x9f9)
01614 #define IOCTL_WD_EVENT_REGISTER        WD_CTL_CODE(0x9ef)
01615 #define IOCTL_WD_EVENT_UNREGISTER      WD_CTL_CODE(0x9f0)
01616 #define IOCTL_WD_EVENT_PULL            WD_CTL_CODE(0x9f1)
01617 #define IOCTL_WD_EVENT_SEND            WD_CTL_CODE(0x9f2)
01618 #define IOCTL_WD_DEBUG_ADD             WD_CTL_CODE(0x964)
01619 #define IOCTL_WD_USAGE                WD_CTL_CODE(0x976)
01620 #define IOCTL_WDU_GET_DEVICE_DATA      WD_CTL_CODE(0x9a7)
01621 #define IOCTL_WDU_SET_INTERFACE        WD_CTL_CODE(0x981)
01622 #define IOCTL_WDU_RESET_PIPE           WD_CTL_CODE(0x982)
01623 #define IOCTL_WDU_TRANSFER             WD_CTL_CODE(0x983)
01624 #define IOCTL_WDU_HALT_TRANSFER        WD_CTL_CODE(0x985)
01625 #define IOCTL_WDU_WAKEUP               WD_CTL_CODE(0x98a)
01626 #define IOCTL_WDU_RESET_DEVICE         WD_CTL_CODE(0x98b)
01627 #define IOCTL_WD_GET_DEVICE_PROPERTY   WD_CTL_CODE(0x990)
01628 #define IOCTL_WD_CARD_CLEANUP_SETUP    WD_CTL_CODE(0x995)
01629 #define IOCTL_WD_DMA_SYNC_CPU          WD_CTL_CODE(0x99f)
01630 #define IOCTL_WD_DMA_SYNC_IO           WD_CTL_CODE(0x9a0)
01631 #define IOCTL_WDU_STREAM_OPEN          WD_CTL_CODE(0x9a8)
01632 #define IOCTL_WDU_STREAM_CLOSE         WD_CTL_CODE(0x9a9)
01633 #define IOCTL_WDU_STREAM_START         WD_CTL_CODE(0x9af)
01634 #define IOCTL_WDU_STREAM_STOP          WD_CTL_CODE(0x9b0)
01635 #define IOCTL_WDU_STREAM_FLUSH         WD_CTL_CODE(0x9aa)
01636 #define IOCTL_WDU_STREAM_GET_STATUS    WD_CTL_CODE(0x9b5)
01637 #define IOCTL_WDU_SELECTIVE_SUSPEND    WD_CTL_CODE(0x9ae)
01638 #define IOCTL_WD_PCI_SCAN_CAPS        WD_CTL_CODE(0x9e5)
01639 #define IOCTL_WD_IPC_REGISTER          WD_CTL_CODE(0x9eb)
01640 #define IOCTL_WD_IPC_UNREGISTER        WD_CTL_CODE(0x9ec)
01641 #define IOCTL_WD_IPC_SCAN_PROCS        WD_CTL_CODE(0x9ed)
01642 #define IOCTL_WD_IPC_SEND              WD_CTL_CODE(0x9ee)
01643 #define IOCTL_WD_PCI_SRIOV_ENABLE      WD_CTL_CODE(0x9f5)
01644 #define IOCTL_WD_PCI_SRIOV_DISABLE     WD_CTL_CODE(0x9f6)
01645 #define IOCTL_WD_PCI_SRIOV_GET_NUMVFS  WD_CTL_CODE(0x9f7)
01646 #define IOCTL_WD_IPC_SHARED_INT_ENABLE WD_CTL_CODE(0x9fc)
```

```
01647 #define IOCTL_WD_IPC_SHARED_INT_DISABLE WD_CTL_CODE(0x9fd)
01648 #define IOCTL_WD_DMA_TRANSACTION_INIT WD_CTL_CODE(0x9fe)
01649 #define IOCTL_WD_DMA_TRANSACTION_EXECUTE WD_CTL_CODE(0x9ff)
01650 #define IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK WD_CTL_CODE(0xa00)
01651 #define IOCTL_WD_DMA_TRANSACTION_RELEASE WD_CTL_CODE(0xa01)
01652
01653 #if defined(UNIX)
01654     typedef struct
01655     {
01656         DWORD dwHeader;
01657         DWORD dwSize;
01658         PVOID pData;
01659         PAD_TO_64(pData)
01660     } WD_IOCTL_HEADER;
01661
01662     #define WD_IOCTL_HEADER_CODE 0xa410b413UL
01663 #endif
01664
01665 #if defined(__KERNEL__)
01666     HANDLE __cdecl WD_Open(void);
01667     void __cdecl WD_Close(HANDLE hWD);
01668     DWORD __cdecl KP_DeviceIoControl(DWORD dwFuncNum, HANDLE h, PVOID pParam,
01669         DWORD dwSize);
01670     #define WD_FUNCTION(wFuncNum, h, pParam, dwSize, fWait) \
01671         KP_DeviceIoControl((DWORD)wFuncNum, h, (PVOID)pParam, (DWORD)dwSize)
01672 #else
01673     #define REGKEY_BUFSIZE 256
01674     #define OS_CAN_NOT_DETECT_TEXT "OS CAN NOT DETECT"
01675     #define INSTALLATION_TYPE_NOT_DETECT_TEXT "unknown"
01676     typedef struct
01677     {
01678         CHAR cProdName[REGKEY_BUFSIZE];
01679         CHAR cInstallationType[REGKEY_BUFSIZE];
01680         #ifdef WIN32
01681             CHAR cCurrentVersion[REGKEY_BUFSIZE];
01682             CHAR cBuild[REGKEY_BUFSIZE];
01683             CHAR cCsDVersion[REGKEY_BUFSIZE];
01684             DWORD dwMajorVersion;
01685             DWORD dwMinorVersion;
01686         #else
01687             CHAR cRelease[REGKEY_BUFSIZE];
01688             CHAR cReleaseVersion[REGKEY_BUFSIZE];
01689         #endif
01690     } WD_OS_INFO;
01691
01700     WD_OS_INFO DLLCALLCONV get_os_type(void);
01701
01712     DWORD DLLCALLCONV check_secureBoot_enabled(void);
01713     #if defined(APPLE)
01714         DWORD WD_FUNCTION_LOCAL(int wFuncNum, HANDLE h,
01715             PVOID pParam, DWORD dwSize, BOOL fWait);
01716
01717         HANDLE WD_OpenLocal(void);
01718
01719         void WD_CloseLocal(HANDLE h);
01720
01721         #define WD_OpenStreamLocal(read, sync) INVALID_HANDLE_VALUE
01722
01723         #define WD_UStreamRead(hFile, pBuffer, dwNumberOfBytesToRead, \
01724             dwNumberOfBytesRead) \
01725             WD_NOT_IMPLEMENTED
01726
01727         #define WD_UStreamWrite(hFile, pBuffer, dwNumberOfBytesToWrite, \
01728             dwNumberOfBytesWritten) \
01729             WD_NOT_IMPLEMENTED
01730
01731     #elif defined(UNIX)
01732         static inline ULONG WD_FUNCTION_LOCAL(int wFuncNum, HANDLE h,
01733             PVOID pParam, DWORD dwSize, BOOL fWait)
01734         {
01735             WD_IOCTL_HEADER ioctl_hdr;
01736
01737             BZERO(ioctl_hdr);
01738             ioctl_hdr.dwHeader = WD_IOCTL_HEADER_CODE;
01739             ioctl_hdr.dwSize = dwSize;
01740             ioctl_hdr.pData = pParam;
01741             (void)fWait;
01742             #if defined(LINUX)
01743                 return (ULONG)ioctl((int)(long)h, wFuncNum, &ioctl_hdr);
01744             #endif
01745         }
01746
01747         #define WD_OpenLocal() \
01748             ((HANDLE)(long)open(WD_DRIVER_NAME, O_RDWR | O_SYNC))
01749         #define WD_OpenStreamLocal(read, sync) \
01750             ((HANDLE)(long)open(WD_DRIVER_NAME, \
01751                 (read) ? O_RDONLY : O_WRONLY) | \
```

```
01752         ((sync) ? O_SYNC : O_NONBLOCK))  
01753  
01754     #define WD_CloseLocal(h) close((int)(long)(h))  
01755  
01756     #define WD_UStreamRead(hFile, pBuffer, dwNumberOfBytesToRead, \  
01757             dwNumberOfBytesRead)\ \  
01758             WD_NOT_IMPLEMENTED  
01759  
01760     #define WD_UStreamWrite(hFile, pBuffer, dwNumberOfBytesToWrite, \  
01761             dwNumberOfBytesWritten)\ \  
01762             WD_NOT_IMPLEMENTED  
01763  
01764 #elif defined(WIN32)  
01765     #define WD_CloseLocal(h) CloseHandle(h)  
01766  
01767     #define WD_UStreamRead(hFile, pBuffer, dwNumberOfBytesToRead, \  
01768             dwNumberOfBytesRead)\ \  
01769             ReadFile(hFile, pBuffer, dwNumberOfBytesToRead, \  
01770                 dwNumberOfBytesRead, NULL) ? WD_STATUS_SUCCESS : \  
01771                 WD_OPERATION_FAILED  
01772  
01773     #define WD_UStreamWrite(hFile, pBuffer, dwNumberOfBytesToWrite, \  
01774             dwNumberOfBytesWritten)\ \  
01775             WriteFile(hFile, pBuffer, dwNumberOfBytesToWrite, \  
01776                 dwNumberOfBytesWritten, NULL) ? WD_STATUS_SUCCESS : \  
01777                 WD_OPERATION_FAILED  
01778  
01779 #if defined(WIN32)  
01780     #define WD_OpenLocal() \  
01781         CreateFileA(\ \  
01782             WD_DRIVER_NAME, \  
01783             GENERIC_READ, \  
01784             FILE_SHARE_READ | FILE_SHARE_WRITE, \  
01785             NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL)  
01786  
01787     #define WD_OpenStreamLocal(read, sync) \  
01788         CreateFileA(\ \  
01789             WD_DRIVER_NAME, \  
01790             (read) ? GENERIC_READ : GENERIC_WRITE, \  
01791             FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, \  
01792             (sync) ? 0 : FILE_FLAG_OVERLAPPED, NULL)  
01793  
01794     static DWORD WD_FUNCTION_LOCAL(int wFuncNum, HANDLE h, PVOID pParam,  
01795             DWORD dwSize, BOOL fWait)  
01796     {  
01797         DWORD dwTmp;  
01798         HANDLE hWD = fWait ? WD_OpenLocal() : h;  
01799         DWORD rc = WD_WINDRIVER_STATUS_ERROR;  
01800  
01801         if (hWD == INVALID_HANDLE_VALUE)  
01802             return (DWORD)WD_STATUS_INVALID_WD_HANDLE;  
01803  
01804         DeviceIoControl(hWD, (DWORD)wFuncNum, pParam, dwSize, &rc,  
01805             sizeof(DWORD), &dwTmp, NULL);  
01806  
01807         if (fWait)  
01808             WD_CloseLocal(hWD);  
01809  
01810         return rc;  
01811     }  
01812     #endif  
01813 #endif  
01814  
01815     #define WD_FUNCTION WD_FUNCTION_LOCAL  
01816     #define WD_Close WD_CloseLocal  
01817     #define WD_Open WD_OpenLocal  
01818     #define WD_StreamOpen WD_OpenStreamLocal  
01819     #define WD_StreamClose WD_CloseLocal  
01820 #endif  
01821  
01822 #define SIZE_OF_WD_DMA(pDma) \  
01823     ((DWORD)(sizeof(WD_DMA) + ((pDma)->dwPages <= WD_DMA_PAGES ? \  
01824         0 : ((pDma)->dwPages - WD_DMA_PAGES) * sizeof(WD_DMA_PAGE))))  
01825 #define SIZE_OF_WD_EVENT(pEvent) \  
01826     ((DWORD)(sizeof(WD_EVENT) + ((pEvent)->dwNumMatchTables > 0 ? \  
01827         sizeof(WDU_MATCH_TABLE) * ((pEvent)->dwNumMatchTables - 1) : 0)))  
01828  
01829 #define WD_Debug(h, pDebug) \  
01830     WD_FUNCTION(IOCTL_WD_DEBUG, h, pDebug, sizeof(WD_DEBUG), FALSE)  
01831  
01832 #define WD_DebugDump(h, pDebugDump) \  
01833     WD_FUNCTION(IOCTL_WD_DEBUG_DUMP, h, pDebugDump, sizeof(WD_DEBUG_DUMP), \  
01834             FALSE)  
01835  
01836 #define WD_DebugAdd(h, pDebugAdd) \  
01837     WD_FUNCTION(IOCTL_WD_DEBUG_ADD, h, pDebugAdd, sizeof(WD_DEBUG_ADD), FALSE)  
01838  
01839
```

```
01884 #define WD_Transfer(h,pTransfer) \
01885     WD_FUNCTION(IOCTL_WD_TRANSFER, h, pTransfer, sizeof(WD_TRANSFER), FALSE)
01886
01903 #define WD_MultiTransfer(h, pTransferArray, dwNumTransfers) \
01904     WD_FUNCTION(IOCTL_WD_MULTI_TRANSFER, h, pTransferArray, \
01905         sizeof(WD_TRANSFER) * (dwNumTransfers), FALSE)
01906
01920 #define WD_KernelBufLock(h, pKerBuf) \
01921     WD_FUNCTION(IOCTL_WD_KERNEL_BUF_LOCK, h, pKerBuf, \
01922         sizeof(WD_KERNEL_BUFFER), FALSE)
01923
01935 #define WD_KernelBufUnlock(h, pKerBuf) \
01936     WD_FUNCTION(IOCTL_WD_KERNEL_BUF_UNLOCK, h, pKerBuf, \
01937         sizeof(WD_KERNEL_BUFFER), FALSE)
01938
02001 #define WD_DMALock(h,pDma) \
02002     WD_FUNCTION(IOCTL_WD_DMA_LOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
02003
02016 #define WD_DMAUnlock(h,pDma) \
02017     WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
02018
02032 #define WD_DMATransactionInit(h,pDma) \
02033     WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_INIT, h, pDma, SIZE_OF_WD_DMA(pDma), \
02034         FALSE)
02035
02046 #define WD_DMATransactionExecute(h,pDma) \
02047     WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_EXECUTE, h, pDma, \
02048         SIZE_OF_WD_DMA(pDma), FALSE)
02049
02060 #define WD_DMATransferCompletedAndCheck(h,pDma) \
02061     WD_FUNCTION(IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK, h, pDma, \
02062         SIZE_OF_WD_DMA(pDma), FALSE)
02063
02075 #define WD_DMATransactionRelease(h,pDma) \
02076     WD_FUNCTION(IOCTL_WD_DMA_TRANSACTION_RELEASE, h, pDma, \
02077         SIZE_OF_WD_DMA(pDma), FALSE)
02078
02090 #define WD_DMATransactionUninit(h,pDma) \
02091     WD_FUNCTION(IOCTL_WD_DMA_UNLOCK, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
02092
02116 #define WD_DMASyncCpu(h,pDma) \
02117     WD_FUNCTION(IOCTL_WD_DMA_SYNC_CPU, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
02118
02143 #define WD_DMASyncIo(h,pDma) \
02144     WD_FUNCTION(IOCTL_WD_DMA_SYNC_IO, h, pDma, SIZE_OF_WD_DMA(pDma), FALSE)
02145
02194 #define WD_CardRegister(h,pCard) \
02195     WD_FUNCTION(IOCTL_WD_CARD_REGISTER, h, pCard, sizeof(WD_CARD_REGISTER), \
02196         FALSE)
02197
02210 #define WD_CardUnregister(h,pCard) \
02211     WD_FUNCTION(IOCTL_WD_CARD_UNREGISTER, h, pCard, sizeof(WD_CARD_REGISTER), \
02212         FALSE)
02213
02235 #define WD_IpcRegister(h, pIpcRegister) \
02236     WD_FUNCTION(IOCTL_WD_IPC_REGISTER, h, pIpcRegister, \
02237         sizeof(WD_IPC_REGISTER), FALSE)
02238
02251 #define WD_IpcUnRegister(h, pProcInfo) \
02252     WD_FUNCTION(IOCTL_WD_IPC_UNREGISTER, h, pProcInfo, sizeof(WD_IPC_PROCESS), \
02253         FALSE)
02254
02268 #define WD_IpcScanProcs(h, pIpcScanProcs) \
02269     WD_FUNCTION(IOCTL_WD_IPC_SCAN_PROCS, h, pIpcScanProcs, \
02270         sizeof(WD_IPC_SCAN_PROCS), FALSE)
02271
02285 #define WD_IpcSend(h, pIpcSend) \
02286     WD_FUNCTION(IOCTL_WD_IPC_SEND, h, pIpcSend, sizeof(WD_IPC_SEND), FALSE)
02287
02305 #define WD_SharedIntEnable(h, pIpcRegister) \
02306     WD_FUNCTION(IOCTL_WD_IPC_SHARED_INT_ENABLE, h, pIpcRegister, \
02307         sizeof(WD_IPC_REGISTER), FALSE)
02308
02320 #define WD_SharedIntDisable(h) \
02321     WD_FUNCTION(IOCTL_WD_IPC_SHARED_INT_DISABLE, h, 0, 0, FALSE)
02322
02336 #define WD_PciSriovEnable(h,pPciSRIOV) \
02337     WD_FUNCTION(IOCTL_WD_PCI_SRIOV_ENABLE, h, pPciSRIOV, \
02338         sizeof(WD_PCI_SRIOV), FALSE)
02339
02353 #define WD_PciSriovDisable(h,pPciSRIOV) \
02354     WD_FUNCTION(IOCTL_WD_PCI_SRIOV_DISABLE, h, pPciSRIOV, \
02355         sizeof(WD_PCI_SRIOV), FALSE)
02356
02372 #define WD_PciSriovGetNumVFs(h,pPciSRIOV) \
02373     WD_FUNCTION(IOCTL_WD_PCI_SRIOV_GET_NUMVFS, h, pPciSRIOV, \
02374         sizeof(WD_PCI_SRIOV), FALSE)
```

```
02375
02398 #define WD_CardCleanupSetup(h,pCardCleanup) \
02399     WD_FUNCTION(IOCTL_WD_CARD_CLEANUP_SETUP, h, pCardCleanup, \
02400         sizeof(WD_CARD_CLEANUP), FALSE)
02401
02416 #define WD_PciScanCards(h,pPciScan) \
02417     WD_FUNCTION(IOCTL_WD_PCI_SCAN_CARDS, h, pPciScan, \
02418         sizeof(WD_PCI_SCAN_CARDS), FALSE)
02419
02433 #define WD_PciScanCaps(h,pPciScanCaps) \
02434     WD_FUNCTION(IOCTL_WD_PCI_SCAN_CAPS, h, pPciScanCaps, \
02435         sizeof(WD_PCI_SCAN_CAPS), FALSE)
02436
02450 #define WD_PciGetCardInfo(h,pPciCard) \
02451     WD_FUNCTION(IOCTL_WD_PCI_GET_CARD_INFO, h, pPciCard, \
02452         sizeof(WD_PCI_CARD_INFO), FALSE)
02453
02468 #define WD_PciConfigDump(h,pPciConfigDump) \
02469     WD_FUNCTION(IOCTL_WD_PCI_CONFIG_DUMP, h, pPciConfigDump, \
02470         sizeof(WD_PCI_CONFIG_DUMP), FALSE)
02471
02484 #define WD_Version(h,pVerInfo) \
02485     WD_FUNCTION(IOCTL_WD_VERSION, h, pVerInfo, sizeof(WD_VERSION), FALSE)
02486
02507 #define WD_License(h,pLicense) \
02508     WD_FUNCTION(IOCTL_WD_LICENSE, h, pLicense, sizeof(WD_LICENSE), FALSE)
02509
02523 #define WD_KernelPlugInOpen(h,pKernelPlugIn) \
02524     WD_FUNCTION(IOCTL_WD_KERNEL_PLUGIN_OPEN, h, pKernelPlugIn, \
02525         sizeof(WD_KERNEL_PLUGIN), FALSE)
02526
02539 #define WD_KernelPlugInClose(h,pKernelPlugIn) \
02540     WD_FUNCTION(IOCTL_WD_KERNEL_PLUGIN_CLOSE, h, pKernelPlugIn, \
02541         sizeof(WD_KERNEL_PLUGIN), FALSE)
02542
02562 #define WD_KernelPlugInCall(h,pKernelPlugInCall) \
02563     WD_FUNCTION(IOCTL_WD_KERNEL_PLUGIN_CALL, h, pKernelPlugInCall, \
02564         sizeof(WD_KERNEL_PLUGIN_CALL), FALSE)
02565
02589 #define WD_IntEnable(h,pInterrupt) \
02590     WD_FUNCTION(IOCTL_WD_INT_ENABLE, h, pInterrupt, sizeof(WD_INTERRUPT), FALSE)
02591
02603 #define WD_IntDisable(h,pInterrupt) \
02604     WD_FUNCTION(IOCTL_WD_INT_DISABLE, h, pInterrupt, sizeof(WD_INTERRUPT), \
02605         FALSE)
02606
02618 #define WD_IntCount(h,pInterrupt) \
02619     WD_FUNCTION(IOCTL_WD_INT_COUNT, h, pInterrupt, sizeof(WD_INTERRUPT), FALSE)
02620
02637 #define WD_IntWait(h,pInterrupt) \
02638     WD_FUNCTION(IOCTL_WD_INT_WAIT, h, pInterrupt, sizeof(WD_INTERRUPT), TRUE)
02639
02654 #define WD_Sleep(h,pSleep) \
02655     WD_FUNCTION(IOCTL_WD_SLEEP, h, pSleep, sizeof(WD_SLEEP), FALSE)
02656
02657
02658 #define WD_EventRegister(h, pEvent) \
02659     WD_FUNCTION(IOCTL_WD_EVENT_REGISTER, h, pEvent, SIZE_OF_WD_EVENT(pEvent), \
02660         FALSE)
02661 #define WD_EventUnregister(h, pEvent) \
02662     WD_FUNCTION(IOCTL_WD_EVENT_UNREGISTER, h, pEvent, \
02663         SIZE_OF_WD_EVENT(pEvent), FALSE)
02664 #define WD_EventPull(h,pEvent) \
02665     WD_FUNCTION(IOCTL_WD_EVENT_PULL, h, pEvent, SIZE_OF_WD_EVENT(pEvent), FALSE)
02666 #define WD_EventSend(h,pEvent) \
02667     WD_FUNCTION(IOCTL_WD_EVENT_SEND, h, pEvent, SIZE_OF_WD_EVENT(pEvent), FALSE)
02668 #define WD_Usage(h, pStop) \
02669     WD_FUNCTION(IOCTL_WD_USAGE, h, pStop, sizeof(WD_USAGE), FALSE)
02670
02671 #define WD_UGetDeviceData(h, pGetDevData) \
02672     WD_FUNCTION(IOCTL_WDU_GET_DEVICE_DATA, h, pGetDevData, \
02673         sizeof(WDU_GET_DEVICE_DATA), FALSE);
02674 #define WD_GetDeviceProperty(h, pGetDevProperty) \
02675     WD_FUNCTION(IOCTL_WD_GET_DEVICE_PROPERTY, h, pGetDevProperty, \
02676         sizeof(WD_GET_DEVICE_PROPERTY), FALSE);
02677 #define WD_USetInterface(h, pSetIfc) \
02678     WD_FUNCTION(IOCTL_WDU_SET_INTERFACE, h, pSetIfc, \
02679         sizeof(WDU_SET_INTERFACE), FALSE);
02680 #define WD_UResetPipe(h, pResetPipe) \
02681     WD_FUNCTION(IOCTL_WDU_RESET_PIPE, h, pResetPipe, sizeof(WDU_RESET_PIPE), \
02682         FALSE);
02683 #define WD_UTransfer(h, pTrans) \
02684     WD_FUNCTION(IOCTL_WDU_TRANSFER, h, pTrans, sizeof(WDU_TRANSFER), TRUE);
02685 #define WD_UHaltTransfer(h, pHaltTrans) \
02686     WD_FUNCTION(IOCTL_WDU_HALT_TRANSFER, h, pHaltTrans, \
02687         sizeof(WDU_HALT_TRANSFER), FALSE);
02688 #define WD_UWakeUp(h, pWakeup) \
```

```

02689     WD_FUNCTION(IOCTL_WDU_WAKEUP, h, pWakeUp, sizeof(WDU_WAKEUP), FALSE);
02690 #define WD_USelectiveSuspend(h, pSelectiveSuspend) \
02691     WD_FUNCTION(IOCTL_WDU_SELECTIVE_SUSPEND, h, pSelectiveSuspend, \
02692         sizeof(WDU_SELECTIVE_SUSPEND), FALSE);
02693 #define WD_UResetDevice(h, pResetDevice) \
02694     WD_FUNCTION(IOCTL_WDU_RESET_DEVICE, h, pResetDevice, \
02695         sizeof(WDU_RESET_DEVICE), FALSE);
02696 #define WD_UStreamOpen(h, pStream) \
02697     WD_FUNCTION(IOCTL_WDU_STREAM_OPEN, h, pStream, sizeof(WDU_STREAM), FALSE);
02698 #define WD_UStreamClose(h, pStream) \
02699     WD_FUNCTION(IOCTL_WDU_STREAM_CLOSE, h, pStream, sizeof(WDU_STREAM), FALSE);
02700 #define WD_UStreamStart(h, pStream) \
02701     WD_FUNCTION(IOCTL_WDU_STREAM_START, h, pStream, sizeof(WDU_STREAM), FALSE);
02702 #define WD_UStreamStop(h, pStream) \
02703     WD_FUNCTION(IOCTL_WDU_STREAM_STOP, h, pStream, sizeof(WDU_STREAM), FALSE);
02704 #define WD_UStreamFlush(h, pStream) \
02705     WD_FUNCTION(IOCTL_WDU_STREAM_FLUSH, h, pStream, sizeof(WDU_STREAM), FALSE);
02706 #define WD_UStreamGetStatus(h, pStreamStatus) \
02707     WD_FUNCTION(IOCTL_WDU_STREAM_GET_STATUS, h, pStreamStatus, \
02708         sizeof(WDU_STREAM_STATUS), FALSE);
02709
02710 #define __ALIGN_DOWN(val,alignment) ( (val) & ~((alignment) - 1) )
02711 #define __ALIGN_UP(val,alignment) \
02712     ( ((val) + (alignment) - 1) & ~((alignment) - 1) )
02713
02714 #ifdef WDLOG
02715     #include "wd_log.h"
02716 #endif
02717
02718 #ifndef MIN
02719     #define MIN(a,b) ((a) > (b) ? (b) : (a))
02720 #endif
02721 #ifndef MAX
02722     #define MAX(a,b) ((a) > (b) ? (a) : (b))
02723 #endif
02724 #define SAFE_STRING(s) ((s) ? (s) : "")
02725
02726 #define UNUSED_VAR(x) (void)x
02727
02728 #ifdef __cplusplus
02729 }
02730 #endif
02732 #endif /* _WINDRVR_H_ */
02733

```

windrivr_32bit.h File Reference

```
#include "wd_types.h"
```

Typedefs

- `typedef u32 ptr32`

Typedef Documentation

`ptr32`

```
typedef u32 ptr32
```

Definition at line 17 of file [windrivr_32bit.h](#).

windrivr_32bit.h

[Go to the documentation of this file.](#)

```

00001 #ifndef _WINDRVR_32BIT_H_
00002 #define _WINDRVR_32BIT_H_
00003
00004 /* This header contains 32bit representations of structures which
00005 * are used internally and externally */
00006
00007 #include "wd_types.h"

```

```
00008
00009 #if defined(__cplusplus)
00010 extern "C" {
00011 #endif
00012
00013 #if defined(LINUX)
00014     #pragma pack(push,4)
00015 #endif
00016
00017 typedef u32 ptr32;
00018 typedef u32 WD_BUS_TYPE_32B;
00019
00020 typedef struct
00021 {
00022     WD_BUS_TYPE_32B dwBusType;
00023     u32 dwDomainNum;
00024     u32 dwBusNum;
00025     u32 dwSlotFunc;
00026 } WD_BUS_V30_32B;
00027
00028 typedef struct
00029 {
00030     u32 item;
00031     u32 fNotSharable;
00032     union
00033     {
00034         struct
00035         {
00036             u64 pPhysicalAddr;
00037             u64 qwBytes;
00038             u64 pTransAddr;
00039             u32 pUserDirectAddr;
00040             u32 dwBar;
00041             u32 dwOptions;
00042             u64 pReserved;
00043         } Mem;
00044         struct
00045         {
00046             u64 pAddr;
00047             u32 dwBytes;
00048             u32 dwBar;
00049         } IO;
00050         struct
00051         {
00052             u32 dwInterrupt;
00053             u32 dwOptions;
00054             u32 hInterrupt;
00055             u32 dwReserved1;
00056             u64 pReserved2;
00057         } Int;
00058     } I;
00059     WD_BUS_V30_32B Bus;
00060 } I;
00061 } WD_ITEMS_V118_32B;
00062
00063 typedef struct
00064 {
00065     u32 dwItems;
00066     WD_ITEMS_V118_32B Item[WD_CARD_ITEMS];
00067 } WD_CARD_V118_32B;
00068
00069 typedef struct
00070 {
00071     u32 dwVendorId;
00072     u32 dwDeviceId;
00073 } WD_PCI_ID_32B;
00074
00075 typedef struct
00076 {
00077     u32 hKernelPlugIn;
00078     u32 dwMessage;
00079     ptr32 pData;
00080     u32 dwResult;
00081 } WD_KERNEL_PLUGIN_CALL_V40_32B;
00082
00083 typedef struct
00084 {
00085     u32 hInterrupt;
00086     u32 dwOptions;
00087     u32 Cmd;
00088     u32 dwCmds;
00089     // For WD_IntEnable():
00090     WD_KERNEL_PLUGIN_CALL_V40_32B kpCall;
00091     u32 fEnableOk;
00092     // For WD_IntWait() and WD_IntCount()
00093     u32 dwCounter;
00094     u32 dwLost;
00095     u32 fStopped;
```

```
00105     u32 dwLastMessage;
00108     u32 dwEnabledIntType;
00110 } WD_INTERRUPT_V91_32B;
00111
00112 typedef struct
00113 {
00114     WD_CARD_V118_32B Card;
00115     u32 fCheckLockOnly;
00116     u32 hCard;
00117     u32 dwOptions;
00118     CHAR cName[32];
00119     CHAR cDescription[100];
00120 } WD_CARD_REGISTER_V118_32B;
00121
00122
00123 typedef struct
00124 {
00125     u32 hKernelPlugIn;
00126     CHAR cDriverName[WD_MAX_KP_NAME_LENGTH];
00127     CHAR cDriverPath[WD_MAX_KP_NAME_LENGTH];
00128     ptr32 pOpenData;
00129 } WD_KERNEL_PLUGIN_V40_32B;
00130
00131 typedef struct
00132 {
00133     u32 dwDomain;
00135     u32 dwBus;
00136     u32 dwSlot;
00137     u32 dwFunction;
00138 } WD_PCI_SLOT_32B;
00139
00140 typedef struct
00141 {
00142     u32 hEvent;
00143     u32 dwEventType;
00145     u32 dwAction;
00146     u32 dwEventId;
00147     u32 hKernelPlugIn;
00148     u32 dwOptions;
00149     union
00150     {
00151         struct
00152         {
00153             WD_PCI_ID_32B cardId;
00154             WD_PCI_SLOT_32B pciSlot;
00155         } Pci;
00156         struct
00157         {
00158             u32 dwUniqueID;
00159         } Usb;
00160         struct
00161         {
00162             u32 hIpc;
00163             u32 dwSubGroupID;
00164             u32 dwGroupID;
00165
00166             u32 dwSenderUID;
00167             u32 dwMsgID;
00168             u64 qwMsgData;
00169         } Ipc;
00170     } u;
00171     u32 dwNumMatchTables;
00172     WDU_MATCH_TABLE matchTables[1];
00173 } WD_EVENT_V121_32B;
00174
00175 /* Only wdc_defs.h structures should be packed */
00176 #if defined(WINNT)
00177     #include <pshpack1.h>
00178 #endif
00179
00180 typedef struct {
00181     u32 dwAddrSpace;
00183     u32 fisMemory;
00184     u32 dwItemIndex;
00186     u32 reserved;
00187     u64 qwBytes;
00188     u64 pAddr;
00191     u32 pUserDirectMemAddr;
00192 } WDC_ADDR_DESC_32B;
00193
00195 typedef struct WDC_DEVICE_32B{
00196     WD_PCI_ID_32B           id;
00197     WD_PCI_SLOT_32B          slot;
00199     u32                      dwNumAddrSpaces;
00201     ptr32                    pAddrDesc;
00203     WD_CARD_REGISTER_V118_32B cardReg;
00205     WD_KERNEL_PLUGIN_V40_32B kerPlug;
00207     WD_INTERRUPT_V91_32B      Int;
```

```
00208     u32                      hIntThread;
00209
00210     WD_EVENT_V121_32B          Event;
00211     u32                      hEvent;
00212
00213     u32                      pCtx;
00214 } WDC_DEVICE_32B, *PWDC_DEVICE_32B;
00215
00216 #if defined(WINNT)
00217     #include <poppack.h>
00218 #endif
00219
00220 #if defined(LINUX)
00221     #pragma pack(pop)
00222 #endif
00223
00224 #if defined(__cplusplus)
00225 }
00226 #endif
00227
00228 #endif /* _WINDRVVR_32BIT_H_ */
00229
```

windrivr_events.h File Reference

```
#include "windrvr.h"
```

Typedefs

- `typedef void(* EVENT_HANDLER) (WD_EVENT *pEvent, void *pData)`
- `typedef void event_handle_t`

Functions

- `DWORD DLLCALLCONV EventRegister (HANDLE *phEvent, HANDLE hWD, WD_EVENT *pEvent, EVENT_HANDLER pFunc, void *pData)`
- `DWORD DLLCALLCONV EventUnregister (HANDLE hEvent)`
- `WD_EVENT *DLLCALLCONV EventAlloc (DWORD dwNumMatchTables)`
- `void DLLCALLCONV EventFree (WD_EVENT *pe)`
- `WD_EVENT *DLLCALLCONV EventDup (WD_EVENT *peSrc)`
- `WD_EVENT *DLLCALLCONV UsbEventCreate (WDU_MATCH_TABLE *pMatchTables, DWORD dwNumMatchTables, DWORD dwOptions, DWORD dwAction)`
- `WD_EVENT *DLLCALLCONV PciEventCreate (WD_PCI_ID cardId, WD_PCI_SLOT pciSlot, DWORD dwOptions, DWORD dwAction)`

Typedef Documentation

`event_handle_t`

```
typedef void event_handle_t
```

Definition at line 13 of file [windrvr_events.h](#).

`EVENT_HANDLER`

```
typedef void(* EVENT_HANDLER) (WD_EVENT *pEvent, void *pData)
```

Definition at line 12 of file [windrvr_events.h](#).

Function Documentation

EventAlloc()

```
WD_EVENT *DCALLCONV EventAlloc (
    DWORD dwNumMatchTables )
```

EventDup()

```
WD_EVENT *DCALLCONV EventDup (
    WD_EVENT * peSrc )
```

EventFree()

```
void DLLCALLCONV EventFree (
    WD_EVENT * pe )
```

EventRegister()

```
DWORD DLLCALLCONV EventRegister (
    HANDLE * phEvent,
    HANDLE hWD,
    WD_EVENT * pEvent,
    EVENT_HANDLER pFunc,
    void * pData )
```

EventUnregister()

```
DWORD DLLCALLCONV EventUnregister (
    HANDLE hEvent )
```

PciEventCreate()

```
WD_EVENT *DCALLCONV PciEventCreate (
    WD_PCI_ID cardId,
    WD_PCI_SLOT pcislot,
    DWORD dwOptions,
    DWORD dwAction )
```

UsbEventCreate()

```
WD_EVENT *DCALLCONV UsbEventCreate (
    WDU_MATCH_TABLE * pMatchTables,
    DWORD dwNumMatchTables,
    DWORD dwOptions,
    DWORD dwAction )
```

windrvr_events.h

Go to the documentation of this file.

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WINDRVR_EVENTS_H_
00004 #define _WINDRVR_EVENTS_H_
00005
00006 #include "windrvr.h"
00007
```

```
00008 #ifdef __cplusplus
00009 extern "C" {
00010 #endif
00011
00012 typedef void (*EVENT_HANDLER) (WD_EVENT *pEvent, void *pData);
00013 typedef void event_handle_t;
00014
00015 DWORD DLLCALLCONV EventRegister(HANDLE *phEvent, HANDLE hWD, WD_EVENT *pEvent,
00016     EVENT_HANDLER pFunc, void *pData);
00017 DWORD DLLCALLCONV EventUnregister(HANDLE hEvent);
00018
00019 WD_EVENT * DLLCALLCONV EventAlloc(DWORD dwNumMatchTables);
00020 void DLLCALLCONV EventFree(WD_EVENT *pe);
00021 WD_EVENT * DLLCALLCONV EventDup(WD_EVENT *peSrc);
00022 WD_EVENT * DLLCALLCONV UsbEventCreate(WDU_MATCH_TABLE *pMatchTables,
00023     DWORD dwNumMatchTables, DWORD dwOptions, DWORD dwAction);
00024 WD_EVENT * DLLCALLCONV PciEventCreate(WD_PCI_ID cardId, WD_PCI_SLOT pciSlot,
00025     DWORD dwOptions, DWORD dwAction);
00026
00027 #ifdef __cplusplus
00028 }
00029 #endif
00030
00031 #endif /* _WINDRVR_EVENTS_H_ */
00032
```

windrvr_int_thread.h File Reference

```
#include "windrvr.h"
#include "windrvr_events.h"
#include "status_strings.h"
```

Typedefs

- `typedef void(DLLCALLCONV * INT_HANDLER) (PVOID pData)`
- `typedef INT_HANDLER INT_HANDLER_FUNC`

Functions

- `DWORD DLLCALLCONV InterruptEnable (HANDLE *phThread, HANDLE hWD, WD_INTERRUPT *pInt, INT_HANDLER func, PVOID pData)`
- `DWORD DLLCALLCONV InterruptDisable (HANDLE hThread)`

Typedef Documentation

INT_HANDLER

`typedef void(DLLCALLCONV * INT_HANDLER) (PVOID pData)`
Definition at line 14 of file [windrvr_int_thread.h](#).

INT_HANDLER_FUNC

`typedef INT_HANDLER INT_HANDLER_FUNC`
Definition at line 15 of file [windrvr_int_thread.h](#).

Function Documentation

InterruptDisable()

```
DWORD DLLCALLCONV InterruptDisable (
    HANDLE hThread )
```

InterruptEnable()

```
DWORD DLLCALLCONV InterruptEnable (
    HANDLE * phThread,
    HANDLE hWD,
    WD_INTERRUPT * pInt,
    INT_HANDLER func,
    PVOID pData )
```

windrvr_int_thread.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 #ifndef _WINDRVR_INT_THREAD_H_
00004 #define _WINDRVR_INT_THREAD_H_
00005
00006 #include "windrvr.h"
00007 #include "windrvr_events.h"
00008 #include "status_strings.h"
00009
00010 #ifdef __cplusplus
00011 extern "C" {
00012 #endif
00013
00014 typedef void (DLLCALLCONV * INT_HANDLER)(PVOID pData);
00015 typedef INT_HANDLER INT_HANDLER_FUNC;
00016
00017 DWORD DLLCALLCONV InterruptEnable(HANDLE *phThread, HANDLE hWD,
00018     WD_INTERRUPT *pInt, INT_HANDLER func, PVOID pData);
00019
00020 DWORD DLLCALLCONV InterruptDisable(HANDLE hThread);
00021
00022 #ifdef __cplusplus
00023 }
00024 #endif
00025
00026 #endif /* _WINDRVR_INT_THREAD_H_ */
00027
```

windrvr_usb.h File Reference

Data Structures

- struct **WDU_PIPE_INFO**
- struct **WDU_INTERFACE_DESCRIPTOR**
- struct **WDU_ENDPOINT_DESCRIPTOR**
- struct **WDU_CONFIGURATION_DESCRIPTOR**
- struct **WDU_DEVICE_DESCRIPTOR**
- struct **WDU_ALTERNATE_SETTING**
- struct **WDU_INTERFACE**
- struct **WDU_CONFIGURATION**
- struct **WDU_DEVICE**
- struct **WDU_MATCH_TABLE**
- struct **WDU_GET_DEVICE_DATA**
- struct **WDU_SET_INTERFACE**
- struct **WDU_RESET_PIPE**
- struct **WDU_HALT_TRANSFER**
- struct **WDU_WAKEUP**
- struct **WDU_SELECTIVE_SUSPEND**

- struct [WDU_RESET_DEVICE](#)
- struct [WDU_TRANSFER](#)
- struct [WDU_GET_DESCRIPTOR](#)
- struct [WDU_STREAM](#)
- struct [WDU_STREAM_STATUS](#)

Macros

- #define PAD_TO_64(pName)
- #define PAD_TO_64_PTR_ARR(pName, size)
- #define WD_USB_MAX_PIPE_NUMBER 32
- #define WD_USB_MAX_ENDPOINTS WD_USB_MAX_PIPE_NUMBER
- #define WD_USB_MAX_INTERFACES 30
- #define WD_USB_MAX_ALT_SETTINGS 255
- #define WDU_DEVICE_DESC_TYPE 0x01
- #define WDU_CONFIG_DESC_TYPE 0x02
- #define WDU_STRING_DESC_STRING 0x03
- #define WDU_INTERFACE_DESC_TYPE 0x04
- #define WDU_ENDPOINT_DESC_TYPE 0x05
- #define WDU_ENDPOINT_TYPE_MASK 0x03
- #define WDU_ENDPOINT_DIRECTION_MASK 0x80
- #define WDU_ENDPOINT_ADDRESS_MASK 0x0f
- #define WDU_ENDPOINT_DIRECTION_OUT(addr) (!((addr) & WDU_ENDPOINT_DIRECTION_MASK))
- #define WDU_ENDPOINT_DIRECTION_IN(addr) ((addr) & WDU_ENDPOINT_DIRECTION_MASK)
- #define WDU_GET_MAX_PACKET_SIZE(x) ((USHORT) (((x) & 0x7ff) * (1 + (((x) & 0x1800) >> 11))))

Typedefs

- typedef PVOID [WDU_REGISTER_DEVICES_HANDLE](#)

Enumerations

- enum [USB_PIPE_TYPE](#) { PIPE_TYPE_CONTROL = 0 , PIPE_TYPE_ISOCRONOUS = 1 , PIPE_TYPE_BULK = 2 , PIPE_TYPE_INTERRUPT = 3 }
- enum [WDU_DIR](#) { WDU_DIR_IN = 1 , WDU_DIR_OUT = 2 , WDU_DIR_IN_OUT = 3 }
- enum {
 USB_ISOCH_RESET = 0x10 , USB_ISOCH_FULL_PACKETS_ONLY = 0x20 , USB_ABORT_PIPE = 0x40 ,
 USB_ISOCH_NOASAP = 0x80 ,
 USB_BULK_INT_URB_SIZE_OVERRIDE_128K = 0x100 , USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL = 0x200 ,
 USB_TRANSFER_HALT = 0x1 , USB_SHORT_TRANSFER = 0x2 ,
 USB_FULL_TRANSFER = 0x4 , USB_ISOCH_ASAP = 0x8 }
- enum [USB_DIR](#) { USB_DIR_IN = 1 , USB_DIR_OUT = 2 , USB_DIR_IN_OUT = 3 }
- enum [WD_DEVICE_REGISTRY_PROPERTY](#) {
 WdDevicePropertyDeviceDescription , WdDevicePropertyHardwareID , WdDevicePropertyCompatibleIDs ,
 WdDevicePropertyBootConfiguration ,
 WdDevicePropertyBootConfigurationTranslated , WdDevicePropertyClassName , WdDevicePropertyClassGuid ,
 WdDevicePropertyDriverKeyName ,
 WdDevicePropertyManufacturer , WdDevicePropertyFriendlyName , WdDevicePropertyLocationInformation ,
 WdDevicePropertyPhysicalDeviceObjectName ,
 WdDevicePropertyBusTypeGuid , WdDevicePropertyLegacyBusType , WdDevicePropertyBusNumber ,
 WdDevicePropertyEnumeratorName ,
 WdDevicePropertyAddress , WdDevicePropertyUINumber , WdDevicePropertyInstallState , WdDevicePropertyRemovalPolicy }
- enum [WDU_WAKEUP_OPTIONS](#) { WDU_WAKEUP_ENABLE = 0x1 , WDU_WAKEUP_DISABLE = 0x2 }
- enum [WDU_SELECTIVE_SUSPEND_OPTIONS](#) { WDU_SELECTIVE_SUSPEND_SUBMIT = 0x1 ,
 WDU_SELECTIVE_SUSPEND_CANCEL = 0x2 }

Macro Definition Documentation

PAD_TO_64

```
#define PAD_TO_64(
```

```
    pName )
```

Definition at line [15](#) of file [windrvr_usb.h](#).

PAD_TO_64_PTR_ARR

```
#define PAD_TO_64_PTR_ARR(
```

```
    pName,
```

```
    size )
```

Definition at line [22](#) of file [windrvr_usb.h](#).

WD_USB_MAX_ALT_SETTINGS

```
#define WD_USB_MAX_ALT_SETTINGS 255
```

Definition at line [63](#) of file [windrvr_usb.h](#).

WD_USB_MAX_ENDPOINTS

```
#define WD_USB_MAX_ENDPOINTS WD_USB_MAX_PIPE_NUMBER
```

Definition at line [61](#) of file [windrvr_usb.h](#).

WD_USB_MAX_INTERFACES

```
#define WD_USB_MAX_INTERFACES 30
```

Definition at line [62](#) of file [windrvr_usb.h](#).

WD_USB_MAX_PIPE_NUMBER

```
#define WD_USB_MAX_PIPE_NUMBER 32
```

Definition at line [60](#) of file [windrvr_usb.h](#).

WDU_CONFIG_DESC_TYPE

```
#define WDU_CONFIG_DESC_TYPE 0x02
```

Definition at line [94](#) of file [windrvr_usb.h](#).

WDU_DEVICE_DESC_TYPE

```
#define WDU_DEVICE_DESC_TYPE 0x01
```

Definition at line [93](#) of file [windrvr_usb.h](#).

WDU_ENDPOINT_ADDRESS_MASK

```
#define WDU_ENDPOINT_ADDRESS_MASK 0x0f
```

Definition at line [102](#) of file [windrvr_usb.h](#).

WDU_ENDPOINT_DESC_TYPE

```
#define WDU_ENDPOINT_DESC_TYPE 0x05
```

Definition at line 97 of file [windrvr_usb.h](#).

WDU_ENDPOINT_DIRECTION_IN

```
#define WDU_ENDPOINT_DIRECTION_IN(
```

```
    addr ) ((addr) & WDU_ENDPOINT_DIRECTION_MASK)
```

Definition at line 107 of file [windrvr_usb.h](#).

WDU_ENDPOINT_DIRECTION_MASK

```
#define WDU_ENDPOINT_DIRECTION_MASK 0x80
```

Definition at line 101 of file [windrvr_usb.h](#).

WDU_ENDPOINT_DIRECTION_OUT

```
#define WDU_ENDPOINT_DIRECTION_OUT(
```

```
    addr ) (!((addr) & WDU_ENDPOINT_DIRECTION_MASK))
```

Definition at line 105 of file [windrvr_usb.h](#).

WDU_ENDPOINT_TYPE_MASK

```
#define WDU_ENDPOINT_TYPE_MASK 0x03
```

Definition at line 100 of file [windrvr_usb.h](#).

WDU_GET_MAX_PACKET_SIZE

```
#define WDU_GET_MAX_PACKET_SIZE(
```

```
    x ) ((USHORT) (((x) & 0x7ff) * (1 + (((x) & 0x1800) >> 11)))
```

Definition at line 109 of file [windrvr_usb.h](#).

WDU_INTERFACE_DESC_TYPE

```
#define WDU_INTERFACE_DESC_TYPE 0x04
```

Definition at line 96 of file [windrvr_usb.h](#).

WDU_STRING_DESC_STRING

```
#define WDU_STRING_DESC_STRING 0x03
```

Definition at line 95 of file [windrvr_usb.h](#).

Typedef Documentation

WDU_REGISTER_DEVICES_HANDLE

```
typedef PVOID WDU_REGISTER_DEVICES_HANDLE
```

Definition at line 90 of file [windrvr_usb.h](#).

Enumeration Type Documentation

anonymous enum

```
anonymous enum
```

Enumerator

USB_ISOCH_RESET	
USB_ISOCH_FULL_PACKETS_ONLY	
USB_ABORT_PIPE	
USB_ISOCH_NOASAP	
USB_BULK_INT_URB_SIZE_OVERRIDE_128K	
USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL	
USB_TRANSFER_HALT	
USB_SHORT_TRANSFER	
USB_FULL_TRANSFER	
USB_ISOCH_ASAP	

Definition at line 72 of file [windrvr_usb.h](#).

USB_DIR

```
enum USB_DIR
```

Enumerator

USB_DIR_IN	
USB_DIR_OUT	
USB_DIR_IN_OUT	

Definition at line 113 of file [windrvr_usb.h](#).

USB_PIPE_TYPE

```
enum USB_PIPE_TYPE
```

Enumerator

PIPE_TYPE_CONTROL	
PIPE_TYPE_ISOCRONOUS	
PIPE_TYPE_BULK	
PIPE_TYPE_INTERRUPT	

Definition at line 53 of file [windrvr_usb.h](#).

WD_DEVICE_REGISTRY_PROPERTY

```
enum WD_DEVICE_REGISTRY_PROPERTY
```

Enumerator

<code>WdDevicePropertyDeviceDescription</code>	Device description.
<code>WdDevicePropertyHardwareID</code>	The device's hardware IDs.
<code>WdDevicePropertyCompatibleIDs</code>	The device's compatible IDs.
<code>WdDevicePropertyBootConfiguration</code>	The hardware resources assigned to the device by the firmware, in raw data form.
<code>WdDevicePropertyBootConfigurationTranslated</code>	The hardware resources assigned to the device by the firmware, in translated form.
<code>WdDevicePropertyClassName</code>	The name of the device's setup class, in text format.
<code>WdDevicePropertyClassGuid</code>	The GUID for the device's setup class (string format)
<code>WdDevicePropertyDriverKeyName</code>	The name of the driver-specific registry key.
<code>WdDevicePropertyManufacturer</code>	Device manufacturer string.
<code>WdDevicePropertyFriendlyName</code>	Friendly device name (typically defined by the class installer), which can be used to distinguish between two similar devices.
<code>WdDevicePropertyLocationInformation</code>	Information about the device's Location on the bus (string format). The interpretation of this information is bus-specific.
<code>WdDevicePropertyPhysicalDeviceObjectName</code>	The name of the Physical Device Object (PDO) for the device.
<code>WdDevicePropertyBusTypeGuid</code>	The GUID for the bus to which the device is connected.
<code>WdDevicePropertyLegacyBusType</code>	The bus type (e.g., PCI Bus)
<code>WdDevicePropertyBusNumber</code>	The legacy bus number of the bus to which the device is connected.
<code>WdDevicePropertyEnumeratorName</code>	The name of the device's enumerator (e.g., "PCI" or "root")
<code>WdDevicePropertyAddress</code>	The device's bus address. The interpretation of this address is bus-specific.
<code>WdDevicePropertyUINumber</code>	A number associated with the device that can be displayed in the user interface.
<code>WdDevicePropertyInstallState</code>	The device's installation state.
<code>WdDevicePropertyRemovalPolicy</code>	The device's current removal policy (Windows)

Definition at line 320 of file [windrvr_usb.h](#).

WDU_DIR

```
enum WDU_DIR
```

Enumerator

<code>WDU_DIR_IN</code>	
<code>WDU_DIR_OUT</code>	
<code>WDU_DIR_IN_OUT</code>	

Definition at line 65 of file [windrvr_usb.h](#).

WDU_SELECTIVE_SUSPEND_OPTIONS

```
enum WDU_SELECTIVE_SUSPEND_OPTIONS
```

Enumerator

<code>WDU_SELECTIVE_SUSPEND_SUBMIT</code>	
---	--

Enumerator

WDU_SELECTIVE_SUSPEND_CANCEL	<input type="checkbox"/>
------------------------------	--------------------------

Definition at line 386 of file [windrvr_usb.h](#).

WDU_WAKEUP_OPTIONS

enum [WDU_WAKEUP_OPTIONS](#)

Enumerator

WDU_WAKEUP_ENABLE	<input type="checkbox"/>
WDU_WAKEUP_DISABLE	<input type="checkbox"/>

Definition at line 381 of file [windrvr_usb.h](#).

windrvr_usb.h

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 /* @JUNGO_COPYRIGHT_GPL@ */
00004
00005 /* @JUNGO_COPYRIGHT_GPL_OTHER_OS@ */
00006
00007 #if !defined(_WINDRVR_USB_H_)
00008 #define _WINDRVR_USB_H_
00009
00010 /* Use it to pad struct size to 64 bit, when using 32 on 64 bit application */
00011 #ifndef PAD_TO_64
00012 #if defined(i386) && defined(KERNEL_64BIT)
00013 #define PAD_TO_64(pName) DWORD dwPad_##pName;
00014 #else
00015 #define PAD_TO_64(pName)
00016 #endif
00017 #endif
00018
00019 #if defined(i386) && defined(KERNEL_64BIT)
00020 #define PAD_TO_64_PTR_ARR(pName, size) PVOID ptPad_##pName[size];
00021 #else
00022 #define PAD_TO_64_PTR_ARR(pName, size)
00023 #endif
00024
00025 #if defined(LINUX)
00026     #if !defined(__P_TYPES__)
00027         #define __P_TYPES__
00028         #include <wd_types.h>
00029         typedef void VOID;
00030         typedef unsigned char UCHAR;
00031         typedef unsigned short USHORT;
00032         typedef unsigned int UINT;
00033         typedef unsigned long ULONG;
00034         typedef u32 BOOL;
00035         typedef void *PVOID;
00036         typedef unsigned char *PBYTE;
00037         typedef char CHAR;
00038         typedef char *PCHAR;
00039         typedef unsigned short *PWORD;
00040         typedef u32 DWORD, *PDWORD;
00041         typedef int PRHANDLE;
00042         typedef PVOID HANDLE;
00043         typedef long LONG;
00044     #endif
00045     #if !defined(TRUE)
00046         #define TRUE 1
00047     #endif
00048     #if !defined(FALSE)
00049         #define FALSE 0
00050     #endif
00051 #endif
00052
00053 typedef enum {
00054     PIPE_TYPE_CONTROL      = 0,
```

```
00055     PIPE_TYPE_ISOCRONOUS = 1,
00056     PIPE_TYPE_BULK        = 2,
00057     PIPE_TYPE_INTERRUPT   = 3
00058 } USB_PIPE_TYPE;
00059
00060 #define WD_USB_MAX_PIPE_NUMBER 32
00061 #define WD_USB_MAX_ENDPOINTS WD_USB_MAX_PIPE_NUMBER
00062 #define WD_USB_MAX_INTERFACES 30
00063 #define WD_USB_MAX_ALT_SETTINGS 255
00064
00065 typedef enum {
00066     WDU_DIR_IN      = 1,
00067     WDU_DIR_OUT     = 2,
00068     WDU_DIR_IN_OUT  = 3
00069 } WDU_DIR;
00070
00071 /* USB TRANSFER options */
00072 enum {
00073     USB_ISOCH_RESET = 0x10,
00074     USB_ISOCH_FULL_PACKETS_ONLY = 0x20,
00075     /* Windows only, ignored on other OS: */
00076     USB_ABORT_PIPE = 0x40,
00077     USB_ISOCH_NOASAP = 0x80,
00078     USB_BULK_INT_URB_SIZE_OVERRIDE_128K = 0x100, /* Force a 128KB maximum
00079                                         URB size */
00080
00081     /* All OS */
00082     USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL = 0x200,
00083
00084     /* The following flags are no longer used beginning with v6.0: */
00085     USB_TRANSFER_HALT = 0x1,
00086     USB_SHORT_TRANSFER = 0x2,
00087     USB_FULL_TRANSFER = 0x4,
00088     USB_ISOCH_ASAP = 0x8
00089 };
00090
00091 typedef PVOID WDU_REGISTER_DEVICES_HANDLE;
00092
00093 /* Descriptor types */
00094 #define WDU_DEVICE_DESC_TYPE          0x01
00095 #define WDU_CONFIG_DESC_TYPE         0x02
00096 #define WDU_STRING_DESC_STRING      0x03
00097 #define WDU_INTERFACE_DESC_TYPE      0x04
00098 #define WDU_ENDPOINT_DESC_TYPE       0x05
00099
00100 /* Endpoint descriptor fields */
00101 #define WDU_ENDPOINT_TYPE_MASK 0x03
00102 #define WDU_ENDPOINT_DIRECTION_MASK 0x80
00103 /* test direction bit in the bEndpointAddress field of an endpoint
00104 * descriptor. */
00105 #define WDU_ENDPOINT_DIRECTION_OUT(addr) \
00106     (!((addr) & WDU_ENDPOINT_DIRECTION_MASK))
00107 #define WDU_ENDPOINT_DIRECTION_IN(addr) \
00108     ((addr) & WDU_ENDPOINT_DIRECTION_MASK)
00109 #define WDU_GET_MAX_PACKET_SIZE(x) \
00110     ((USHORT) (((x) & 0x7ff) * (1 + (((x) & 0x1800) >> 11))))
00111
00112 #ifndef LINUX
00113 typedef enum {
00114     USB_DIR_IN      = 1,
00115     USB_DIR_OUT     = 2,
00116     USB_DIR_IN_OUT  = 3
00117 } USB_DIR;
00118 #endif
00119
00120 typedef struct
00121 {
00122     DWORD dwNumber;
00123     DWORD dwMaximumPacketSize;
00124     DWORD type;
00125     DWORD direction;
00126     DWORD dwInterval;
00127 } WDU_PIPE_INFO;
00128
00129 typedef struct
00130 {
00131     UCHAR bLength;
00132     UCHAR bDescriptorType;
00133     UCHAR bInterfaceNumber;
00134     UCHAR bAlternateSetting;
00135     UCHAR bNumEndpoints;
00136     UCHAR bInterfaceClass;
00137     UCHAR bInterfaceSubClass;
00138     UCHAR bInterfaceProtocol;
00139     UCHAR iInterface;
00140 } WDU_INTERFACE_DESCRIPTOR;
00141
```

```
00151 typedef struct
00152 {
00153     UCHAR bLength;
00154     UCHAR bDescriptorType;
00155     UCHAR bEndpointAddress;
00160     UCHAR bmAttributes;
00163     USHORT wMaxPacketSize;
00165     UCHAR bInterval;
00169 } WDU_ENDPOINT_DESCRIPTOR;
00170
00171 typedef struct
00172 {
00173     UCHAR bLength;
00174     UCHAR bDescriptorType;
00175     USHORT wTotalLength;
00176     UCHAR bNumInterfaces;
00177     UCHAR bConfigurationValue;
00178     UCHAR iConfiguration;
00180     UCHAR bmAttributes;
00184     UCHAR MaxPower;
00186 } WDU_CONFIGURATION_DESCRIPTOR;
00187
00188 typedef struct
00189 {
00190     UCHAR bLength;
00191     UCHAR bDescriptorType;
00192     USHORT bcdUSB;
00194     UCHAR bDeviceClass;
00195     UCHAR bDeviceSubClass;
00196     UCHAR bDeviceProtocol;
00197     UCHAR bMaxPacketSize0;
00199     USHORT idVendor;
00200     USHORT idProduct;
00202     USHORT bcdDevice;
00203     UCHAR iManufacturer;
00204     UCHAR iProduct;
00205     UCHAR iSerialNumber;
00206     UCHAR bNumConfigurations;
00207 } WDU_DEVICE_DESCRIPTOR;
00208
00209 typedef struct
00210 {
00211     WDU_INTERFACE_DESCRIPTOR Descriptor;
00213     PAD_TO_64(Descriptor)
00214     WDU_ENDPOINT_DESCRIPTOR *pEndpointDescriptors;
00220     PAD_TO_64(pEndpointDescriptors)
00221     WDU_PIPE_INFO *pPipes;
00224     PAD_TO_64(pPipes)
00225 } WDU_ALTERNATE_SETTING;
00226
00227 typedef struct
00228 {
00229     WDU_ALTERNATE_SETTING *pAlternateSettings;
00234     PAD_TO_64(pAlternateSettings)
00235     DWORD dwNumAltSettings;
00238     PAD_TO_64(dwNumAltSettings)
00239     WDU_ALTERNATE_SETTING *pActiveAltSetting;
00243     PAD_TO_64(pActiveAltSetting)
00244 } WDU_INTERFACE;
00245
00246 typedef struct
00247 {
00248     WDU_CONFIGURATION_DESCRIPTOR Descriptor;
00250     DWORD dwNumInterfaces;
00252     WDU_INTERFACE *pInterfaces;
00255     PAD_TO_64(pInterfaces)
00256 } WDU_CONFIGURATION;
00257
00258 typedef struct {
00259     WDU_DEVICE_DESCRIPTOR Descriptor;
00261     WDU_PIPE_INFO Pipe0;
00263     WDU_CONFIGURATION *pConfigs;
00266     PAD_TO_64(pConfigs)
00267     WDU_CONFIGURATION *pActiveConfig;
00271     PAD_TO_64(pActiveConfig)
00272     WDU_INTERFACE *pActiveInterface[WD_USB_MAX_INTERFACES];
00286     PAD_TO_64_PTR_ARR(pActiveInterface, WD_USB_MAX_INTERFACES)
00287 } WDU_DEVICE;
00288
00289 /* Note: Any devices found matching this table will be controlled */
00290 typedef struct
00291 {
00292     USHORT wVendorId;
00294     USHORT wProductId;
00296     UCHAR bDeviceClass;
00298     UCHAR bDeviceSubClass;
00300     UCHAR bInterfaceClass;
```

```
00302     UCHAR bInterfaceSubClass;
00304     UCHAR bInterfaceProtocol;
00306 } WDU_MATCH_TABLE;
00307
00308 typedef struct
00309 {
00310     DWORD dwUniqueID;
00311     PAD_TO_64(dwUniqueID)
00312     PVOID pBuf;
00313     PAD_TO_64(pBuf)
00314     DWORD dwBytes;
00315     DWORD dwOptions;
00316 } WDU_GET_DEVICE_DATA;
00317
00318 /* these enum values can be used as dwProperty values, see structure
00319 * WD_GET_DEVICE_PROPERTY below. */
00320 typedef enum
00321 {
00322     WdDevicePropertyDeviceDescription,
00323     WdDevicePropertyHardwareID,
00324     WdDevicePropertyCompatibleIDs,
00325     WdDevicePropertyBootConfiguration,
00326     WdDevicePropertyBootConfigurationTranslated,
00327     WdDevicePropertyClassName,
00328     WdDevicePropertyClassGuid,
00329     WdDevicePropertyDriverKeyName,
00330     WdDevicePropertyManufacturer,
00331     WdDevicePropertyFriendlyName,
00332     WdDevicePropertyLocationInformation,
00333     WdDevicePropertyPhysicalDeviceObjectName,
00334     WdDevicePropertyBusTypeGuid,
00335     WdDevicePropertyLegacyBusType,
00336     WdDevicePropertyBusNumber,
00337     WdDevicePropertyEnumeratorName,
00338     WdDevicePropertyAddress,
00339     WdDevicePropertyUINumber,
00340     WdDevicePropertyInstallState,
00341     WdDevicePropertyRemovalPolicy
00342 } WD_DEVICE_REGISTRY_PROPERTY;
00343
00344 typedef struct
00345 {
00346     DWORD dwUniqueID;
00347     DWORD dwInterfaceNum;
00348     DWORD dwAlternateSetting;
00349     DWORD dwOptions;
00350 } WDU_SET_INTERFACE;
00351
00352 typedef struct
00353 {
00354     DWORD dwUniqueID;
00355     DWORD dwPipeNum;
00356     DWORD dwOptions;
00357 } WDU_RESET_PIPE;
00358
00359 typedef enum {
00360     WDU_WAKEUP_ENABLE = 0x1,
00361     WDU_WAKEUP_DISABLE = 0x2
00362 } WDU_WAKEUP_OPTIONS;
00363
00364 typedef enum {
00365     WDU_SELECTIVE_SUSPEND_SUBMIT = 0x1,
00366     WDU_SELECTIVE_SUSPEND_CANCEL = 0x2,
00367 } WDU_SELECTIVE_SUSPEND_OPTIONS;
00368
00369 typedef struct
00370 {
00371     DWORD dwUniqueID;
00372     DWORD dwPipeNum;
00373     DWORD dwOptions;
00374 } WDU_HALT_TRANSFER;
00375
00376 typedef struct
00377 {
00378     DWORD dwUniqueID;
00379     DWORD dwOptions;
00380 } WDU_WAKEUP;
00381
00382 typedef struct
00383 {
00384     DWORD dwUniqueID;
00385     DWORD dwOptions;
00386 } WDU_SELECTIVE_SUSPEND;
```

```

00413     DWORD dwOptions;
00414 } WDU_RESET_DEVICE;
00415
00416 typedef struct
00417 {
00418     DWORD dwUniqueID;
00419     DWORD dwPipeNum;
00420     DWORD fRead;
00422     DWORD dwOptions;
00430     PVOID pBuffer;
00431     PAD_TO_64(pBuffer)
00432     DWORD dwBufferSize;
00433     DWORD dwBytesTransferred;
00435     UCHAR SetupPacket[8];
00436     DWORD dwTimeout;
00438     PAD_TO_64(dwTimeout)
00439 } WDU_TRANSFER;
00440
00441 typedef struct
00442 {
00443     DWORD dwUniqueID;
00444     UCHAR bType;
00445     UCHAR bIndex;
00446     USHORT wLength;
00447     PVOID pBuffer;
00448     USHORT wLanguage;
00449 } WDU_GET_DESCRIPTOR;
00450
00451 typedef struct
00452 {
00453     DWORD dwUniqueID;
00454     DWORD dwOptions;
00455     DWORD dwPipeNum;
00456     DWORD dwBufferSize;
00457     DWORD dwRxSize;
00458     BOOL fBlocking;
00459     DWORD dwRxtxTimeout;
00460     DWORD dwReserved;
00461 } WDU_STREAM;
00462
00463 typedef struct
00464 {
00465     DWORD dwUniqueID;
00466     DWORD dwOptions;
00467     BOOL fIsRunning;
00468     DWORD dwLastError;
00469     DWORD dwBytesInBuffer;
00470     DWORD dwReserved;
00471 } WDU_STREAM_STATUS;
00472
00473 #endif /* _WINDRVR_USB_H_ */
00474

```

kp_pci.c File Reference

```

#include "kpstdlib.h"
#include "wd_kp.h"
#include "pci_regs.h"
#include "../pci_lib.h"

```

Macros

- #define PTR32 UINT32
- #define DRIVER_VER_STR "My Driver V1.00"
- #define USE_MULTI_TRANSFER

Functions

- BOOL __cdecl KP_PCI_Open (KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData, PVOID *ppDrvContext)

Kernel PlugIn open function.
- BOOL __cdecl KP_PCI_Open_32_64 (KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData, PVOID *ppDrvContext)

KP_PCI_Open_32_64 is called when [WD_KernelPlugInOpen\(\)](#) is called from a 32-bit user mode application to open a handle to a 64-bit Kernel PlugIn.

- void __cdecl [KP_PCI_Close](#) (PVOID pDrvContext)

Called when [WD_KernelPlugInClose\(\)](#) (see the WinDriver PCI Low-Level API Reference) is called from user mode.

- void __cdecl [KP_PCI_Call](#) (PVOID pDrvContext, [WD_KERNEL_PLUGIN_CALL](#) *kpCall)

Called when the user-mode application calls [WDC_CallKerPlug\(\)](#) (or the low-level [WD_KernelPlugInCall\(\)](#) function — see the WinDriver PCI Low-Level API Reference).

- BOOL __cdecl [KP_PCI_IntEnable](#) (PVOID pDrvContext, [WD_KERNEL_PLUGIN_CALL](#) *kpCall, PVOID *ppIntContext)

Called when [WDC_IntEnable\(\)](#) / [WD_IntEnable\(\)](#) is called from the user mode with a Kernel PlugIn handle.

- void __cdecl [KP_PCI_IntDisable](#) (PVOID pIntContext)

Called when [WDC_IntDisable\(\)](#) / [WD_IntDisable\(\)](#) is called from the user mode for interrupts that were enabled in the Kernel PlugIn.

- BOOL __cdecl [KP_PCI_IntAtIrql](#) (PVOID pIntContext, BOOL *pfIsMyInterrupt)

High-priority legacy interrupt handler routine, which is run at high interrupt request level.

- DWORD __cdecl [KP_PCI_IntAtDpc](#) (PVOID pIntContext, DWORD dwCount)

Deferred processing legacy interrupt handler routine.

- BOOL __cdecl [KP_PCI_IntAtIrqlMSI](#) (PVOID pIntContext, ULONG dwLastMessage, DWORD dwReserved)

High-priority Message-Signaled Interrupts (MSI) / Extended Message-Signaled Interrupts (MSI-X) handler routine, which is run at high interrupt request level.

- DWORD __cdecl [KP_PCI_IntAtDpcMSI](#) (PVOID pIntContext, DWORD dwCount, ULONG dwLastMessage, DWORD dwReserved)

Deferred processing Message-Signaled Interrupts (MSI) / Extended Message-Signaled Interrupts (MSI-X) handler routine.

- BOOL __cdecl [KP_PCI_Event](#) (PVOID pDrvContext, [WD_EVENT](#) *wd_event)

Called when a Plug-and-Play or power management event for the device is received, provided the user-mode application first called [WDC_EventRegister\(\)](#) with fUseKP = TRUE (or the low-level [EventRegister\(\)](#) function with a Kernel PlugIn handle — see WinDriver PCI Low-Level API Reference)

- BOOL __cdecl [KP_Init](#) ([KP_INIT](#) *kpInit)

KP_Init is called when the Kernel PlugIn driver is loaded.

Macro Definition Documentation

DRIVER_VER_STR

```
#define DRIVER_VER_STR "My Driver V1.00"
```

PTR32

```
#define PTR32 UINT32
```

Definition at line 46 of file [kp_pci.c](#).

USE_MULTI_TRANSFER

```
#define USE_MULTI_TRANSFER
```

Function Documentation

KP_Init()

```
BOOL __cdecl KP_Init (
    KP_INIT * kpInit )
```

KP_Init is called when the Kernel PlugIn driver is loaded.

You must define a [KP_Init\(\)](#) function to link to the device driver.

This function sets the name of the Kernel PlugIn driver and the driver's open callback function(s).

Parameters

out	<i>kpInit</i>	Pointer to a pre-allocated Kernel PlugIn initialization information structure, whose fields should be updated by the function
-----	---------------	---

Returns

TRUE if successful. Otherwise FALSE.

Definition at line 65 of file [kp_pci.c](#).

KP_PCI_Call()

```
void __cdecl KP_PCI_Call (
    PVOID pDrvContext,
    WD_KERNEL_PLUGIN_CALL * kpCall )
```

Called when the user-mode application calls [WDC_CallKerPlug\(\)](#) (or the low-level [WD_KernelPlugInCall\(\)](#) function — see the WinDriver PCI Low-Level API Reference).

This function is a message handler for your utility functions.

Parameters

in,out	<i>pDrvContext</i>	Driver context data that was set by KP_Open() and will also be passed to KP_Close(), KP_IntEnable() and KP_Event()
in,out	<i>kpCall</i>	Structure with user-mode information received from the WDC_CallKerPlug() (or from the low-level WD_KernelPlugInCall() function — see the WinDriver PCI Low-Level API Reference) and/or with information to return back to the user mode.
in	<i>kpCall->dwMessage</i>	Message number for the handler.
in,out	<i>kpCall->pData</i>	Message context.
out	<i>kpCall->dwResult</i>	Message result to send back to the user application.

Remarks

Calling [WDC_CallKerPlug\(\)](#)(or the low-level [WD_KernelPlugInCall\(\)](#) function — see the WinDriver PCI Low-Level API Reference) in the user mode will call your KP_Call() 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. The fIsKernelMode 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.

Definition at line 377 of file [kp_pci.c](#).

KP_PCI_Close()

```
void __cdecl KP_PCI_Close (
    PVOID pDrvContext )
```

Called when [WD_KernelPlugInClose\(\)](#) (see the WinDriver PCI Low-Level API Reference) is called from user mode. The high-level [WDC_PciDeviceClose\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) functions automatically call [WD_KernelPlugInClose\(\)](#) for devices that contain an open Kernel PlugIn handle.

This functions can be used to perform any required clean-up for the Kernel PlugIn (such as freeing memory previously allocated for the driver context, etc.).

Parameters

in	<i>pDrvContext</i>	Driver context data that was set by KP_Open()
----	--------------------	---

Definition at line 326 of file [kp_pci.c](#).

KP_PCI_Event()

```
BOOL __cdecl KP_PCI_Event (
    PVOID pDrvContext,
    WD_EVENT * wd_event )
```

Called when a Plug-and-Play or power management event for the device is received, provided the user-mode application first called [WDC_EventRegister\(\)](#) with fUseKP = TRUE (or the low-level [EventRegister\(\)](#) function with a Kernel PlugIn handle — see WinDriver PCI Low-Level API Reference)

Parameters

in,out	<i>pDrvContext</i>	Driver context data that was set by KP_Open() and will also be passed to KP_Close() , KP_IntEnable() and KP_Event()
in	<i>wd_event</i>	Pointer to the PnP/power management event information received from the user mode

Returns

TRUE in order to notify the user about the event. FALSE otherwise.

Remarks

[KP_Event](#) will be called if the user mode process called [WDC_EventRegister\(\)](#) with fUseKP= TRUE (or of the low-level [EventRegister\(\)](#) function was called with a Kernel PlugIn handle — see the WinDriver PCI Low-Level API Reference).

Definition at line 712 of file [kp_pci.c](#).

KP_PCI_IntAtDpc()

```
DWORD __cdecl KP_PCI_IntAtDpc (
    PVOID pIntContext,
    DWORD dwCount )
```

Deferred processing legacy interrupt handler routine.

This function is called once the high-priority legacy interrupt handling is completed, provided that [KP_IntAtIrql\(\)](#) returned TRUE.

Parameters

in,out	<i>pIntContext</i>	Interrupt context data that was set by KP_IntEnable() , passed to KP_IntAtIrql() , and will be passed to KP_IntDisable() .
in	<i>dwCount</i>	The number of times KP_IntAtIrql() returned TRUE since the last DPC call. If <i>dwCount</i> 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.

Returns

Returns the number of times to notify user mode (i.e., return from [WD_IntWait\(\)](#) — see the WinDriver PCI Low-Level API Reference).

Remarks

Most of the interrupt handling should be implemented within this function, as opposed to the high-priority `KP_IntAtIrql()` interrupt handler. If `KP_IntAtDpc()` returns with a value greater than zero, [WD_IntWait\(\)](#) returns and the user-mode interrupt handler will be called in the amount of times set in the return value of `KP_IntAtDpc()`. If you do not want the user-mode interrupt handler to execute, `KP_IntAtDpc()` should return zero.

Definition at line 602 of file [kp_pci.c](#).

KP_PCI_IntAtDpcMSI()

```
DWORD __cdecl KP_PCI_IntAtDpcMSI (
    PVOID pIntContext,
    DWORD dwCount,
    ULONG dwLastMessage,
    DWORD dwReserved )
```

Deferred processing Message-Signaled Interrupts (MSI) / Extended Message-Signaled Interrupts (MSI-X) handler routine.

This function is called once the high-priority MSI/MSI-X handling is completed, provided that `KP_IntAtIrqlMSI` [B.← 8.10] returned TRUE.

Parameters

<code>in, out</code>	<code>pIntContext</code>	Interrupt context data that was set by <code>KP_IntEnable()</code> , passed to <code>KP_IntAtIrqlMSI()</code> , and will be passed to <code>KP_IntDisable()</code> .
<code>in</code>	<code>dwCount</code>	The number of times <code>KP_IntAtIrqlMSI()</code> returned TRUE since the last DPC call. If <code>dwCount</code> is 1, <code>KP_IntAtIrqlMSI()</code> requested a DPC only once since the last DPC call. If the value is greater than 1, <code>KP_IntAtIrqlMSI()</code> has already requested a DPC a few times, but the interval was too short, therefore <code>KP_IntAtDpcMSI()</code> was not called for each DPC request.
<code>in</code>	<code>dwLastMessage</code>	The message data for the last received interrupt.
<code>in</code>	<code>dwReserved</code>	Reserved for future use. Do not use this parameter.

Returns

Returns the number of times to notify user mode (i.e., return from [WD_IntWait\(\)](#) — see the WinDriver PCI Low-Level API Reference).

Remarks

Most of the MSI/MSI-X handling should be implemented within this function, as opposed to the high-priority `KP_IntAtIrqlMSI()` interrupt handler. If `KP_IntAtDpcMSI` returns with a value greater than zero, [WD_IntWait\(\)](#) returns and the user-mode interrupt handler will be called in the amount of times set in the return value of `KP_IntAtDpcMSI`. If you do not want the user-mode interrupt handler to execute, `KP_IntAtDpcMSI` should return zero.

Definition at line 688 of file [kp_pci.c](#).

KP_PCI_IntAtIrql()

```
BOOL __cdecl KP_PCI_IntAtIrql (
    PVOID pIntContext,
    BOOL * pfIsMyInterrupt )
```

High-priority legacy interrupt handler routine, which is run at high interrupt request level. This function is called upon the arrival of a legacy interrupt that has been enabled using a Kernel PlugIn driver — see the description of [WDC_IntEnable\(\)](#) or the low-level [InterruptEnable\(\)](#) and [WD_IntEnable\(\)](#) functions (see WinDriver PCI Low-Level API Reference).

Parameters

in, out	<i>pIntContext</i>	Pointer to interrupt context data that was set by KP_IntEnable() and will also be passed to KP_IntAtDpc() (if executed) and KP_IntDisable().
out	<i>pfIsMyInterrupt</i>	Set * <i>pfIsMyInterrupt</i> 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.

Returns

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 high IRQL is limited in the following ways: It may only access non-pageable memory. It may only call the following functions (or wrapper functions that call * these functions): WDC_xxx() read/write address or configuration space functions. [WDC_MultiTransfer\(\)](#), or the low-level [WD_Transfer\(\)](#), [WD_MultiTransfer\(\)](#), or [WD_DebugAdd\(\)](#) functions (see the WinDriver PCI Low-Level API Reference). Specific kernel OS functions (such as WDK 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. 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 in KP_IntAtDpc(), which runs at the deferred DISPATCH level and is not subject to the above restrictions.

Definition at line 495 of file [kp_pci.c](#).

KP_PCI_IntAtIrqlMSI()

```
BOOL __cdecl KP_PCI_IntAtIrqlMSI (
    PVOID pIntContext,
    ULONG dwLastMessage,
    DWORD dwReserved )
```

High-priority Message-Signaled Interrupts (MSI) / Extended Message-Signaled Interrupts (MSI-X) handler routine, which is run at high interrupt request level.

This function is called upon the arrival of an MSI/MSI-X that has been enabled using a Kernel PlugIn — see the description of [WDC_IntEnable\(\)](#) or the low-level [InterruptEnable\(\)](#) and [WD_IntEnable\(\)](#) functions (see WinDriver PCI Low-Level API Reference).

Parameters

in, out	<i>pIntContext</i>	Pointer to interrupt context data that was set by KP_IntEnable() and will also be passed to KP_IntAtDpcMSI() (if executed) and KP_IntDisable()
in	<i>dwLastMessage</i>	The message data for the last received interrupt. @param [in] dwReserved: Reserved for future use. Do not use this parameter.

Returns

TRUE if deferred MSI/MSI-X processing (DPC) is required; otherwise FALSE.

Remarks

Code running at IRQL will only be interrupted by higher priority interrupts. Code running at high IRQL is limited in the following ways: It may only access non-pageable memory. It may only call the following functions (or wrapper functions that call these functions): WDC_xxx() read/write address or configuration space functions. **WDC_MultiTransfer()**, or the low-level **WD_Transfer()**, **WD_MultiTransfer()**, or **WD_DebugAdd()** functions (see the WinDriver PCI Low-Level API Reference). Specific kernel OS functions (such as WDK 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. The code performed at high interrupt request level should be minimal, since it is operating at a high priority. The rest of your code should be written in KP_IntAtDpcMSI, which runs at the deferred DISPATCH level and is not subject to the above restrictions.

Definition at line 643 of file [kp_pci.c](#).

KP_PCI_IntDisable()

```
void __cdecl KP_PCI_IntDisable (
    PVOID pIntContext )
```

Called when **WDC_IntDisable()** / **WD_IntDisable()** is called from the user mode for interrupts that were enabled in the Kernel Plugin.

WD_IntDisable() is called automatically from **WDC_IntDisable()** and **InterruptDisable()** (see WinDriver PCI Low-Level API Reference).

This function should free any memory that was allocated in **KP_IntEnable**.

Parameters

in	<i>pIntContext</i>	Interrupt context data that was set by KP_IntEnable()
----	--------------------	--

Definition at line 456 of file [kp_pci.c](#).

KP_PCI_IntEnable()

```
BOOL __cdecl KP_PCI_IntEnable (
    PVOID pDrvContext,
    WD_KERNEL_PLUGIN_CALL * kpCall,
    PVOID * ppIntContext )
```

Called when **WDC_IntEnable()** / **WD_IntEnable()** is called from the user mode with a Kernel Plugin handle.

WD_IntEnable() is called automatically from **WDC_IntEnable()** and **InterruptEnable()** (see WinDriver PCI Low-Level API Reference).

The interrupt context that is set by this function (*ppIntContext) will be passed to the rest of the Kernel Plugin interrupt functions.

Parameters

in, out	<i>pDrvContext</i>	Driver context data that was set by KP_Open() and will also be passed to KP_Close() , KP_IntEnable() and KP_Event()
in, out	<i>kpCall</i>	Structure with information from WD_IntEnable()
in, out	<i>ppIntContext</i>	Pointer to interrupt context data that will be passed to KP_IntDisable() and to the Kernel Plugin interrupt handler functions. Use this context to keep interrupt specific information.

Returns

Returns TRUE if enable is successful; otherwise returns FALSE.

Remarks

This function should contain any initialization needed for your Kernel PlugIn interrupt handling.

Definition at line 428 of file [kp_pci.c](#).

KP_PCI_Open()

```
BOOL __cdecl KP_PCI_Open (
    KP_OPEN_CALL * kpOpenCall,
    HANDLE hWD,
    PVOID pOpenData,
    PVOID * ppDrvContext )
```

Kernel PlugIn open function.

This function 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). The returned driver context (*ppDrvContext) will be passed to rest of the Kernel PlugIn callback functions.

The KP_Open callback is called when the [WD_KernelPlugInOpen\(\)](#) function (see the WinDriver PCI Low-Level API Reference) is called from the user mode — either directly (when using the low-level WinDriver API), or via a call to a high-level WDC function. [WD_KernelPlugInOpen\(\)](#) is called from the [WDC_KernelPlugInOpen\(\)](#), and from the [WDC_PciDeviceOpen\(\)](#) / [WDC_IsaDeviceOpen\(\)](#) functions when they are called with the name of a valid Kernel PlugIn driver (set in the pcKPDriverName parameter).

[Note] The [WDC_xxxDeviceOpen\(\)](#) functions cannot be used to open a handle to a 64-bit Kernel PlugIn function from a 32-bit application. For this purpose, use [WDC_KernelPlugInOpen\(\)](#) (or the low-level [WD_KernelPlugInOpen\(\)](#) function). The Kernel PlugIn driver can implement two types of KP_Open() callback functions — A "standard" Kernel PlugIn open function, which is used whenever a user-mode application opens a handle to a Kernel PlugIn driver, except when a 32-bit applications opens a handle to a 64-bit driver. This callback function is set in the funcOpen field of the [KP_INIT](#) structure that is passed as a parameter to [KP_Init\(\)](#). A function that will be used when a 32-bit user-mode application opens a handle to a 64-bit Kernel PlugIn driver. This callback function is set in the funcOpen_32_64 field of the [KP_INIT](#) structure that is passed as a parameter to KP_Init. A Kernel PlugIn driver can provide either one or both of these KP_Open() callbacks, depending on the target configuration(s).

[Note] The KP_PCI sample (WinDriver/samples/pci_diag/kp_pci/kp_pci.c) implements both types of KP_Open callbacks — [KP_PCI_Open\(\)](#) (standard) and [KP_PCI_Open_32_64\(\)](#) (for opening a handle to a 64-bit Kernel PlugIn from a 32-bit application). The generated DriverWizard Kernel PlugIn code always implements a standard Kernel PlugIn open function — KP_XXX_Open(). When selecting the 32-bit application for a 64-bit Kernel PlugIn DriverWizard code-generation option, the wizard also implements a KP_XXX_Open_32_64() function, for opening a handle to a 64-bit Kernel PlugIn driver from a 32-bit application.

Parameters

in	<i>kpOpenCall</i>	Structure to fill in the addresses of the KP_xxx callback functions
in	<i>hWD</i>	The WinDriver handle that WD_KernelPlugInOpen() was called with
in	<i>pOpenData</i>	Pointer to data passed from user mode
out	<i>ppDrvContext</i>	Pointer to driver context data with which the KP_Close(), KP_Call(), KP_IntEnable() and KP_Event() functions will be called. Use this to keep driver-specific information that will be shared among these callbacks.

Returns

TRUE if successful. If FALSE, the call to [WD_KernelPlugInOpen\(\)](#) from the user mode will fail

Definition at line 154 of file [kp_pci.c](#).

KP_PCI_Open_32_64()

```
BOOL __cdecl KP_PCI_Open_32_64 (
    KP_OPEN_CALL * kpOpenCall,
```

```
    HANDLE hWD,
    PVOID pOpenData,
    PVOID * ppDrvContext )
```

KP_PCI_Open_32_64 is called when [WD_KernelPlugInOpen\(\)](#) is called from a 32-bit user mode application to open a handle to a 64-bit Kernel Plugin.

pDrvContext will be passed to the rest of the Kernel Plugin callback functions. See [KP_PCI_Open\(\)](#) for more info. Definition at line 234 of file [kp_pci.c](#).

kp_pci.c

[Go to the documentation of this file.](#)

```
00001 /* @JUNGO_COPYRIGHT@ */
00002
00003 ****
00004 * File: kp_pci.c
00005 *
00006 * Kernel PlugIn driver for accessing PCI devices.
00007 * The code accesses hardware using WinDriver's WDC library.
00008 @CODE_GEN@
00009 *
00010 * Note: This code sample is provided AS-IS and as a guiding sample only.
00011 ****
00012
00013 #include "kpstdlib.h"
00014 #include "wd_kp.h"
00015 #ifndef ISA
00016 #include "pci_regs.h"
00017 #endif /* ifndef ISA */
00018 #include "../pci_lib.h"
00019
00020 ****
00021 Functions prototypes
00022 ****
00023 BOOL __cdecl KP_PCI_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData,
00024     PVOID *ppDrvContext);
00025 /* @32on64@ */
00026 BOOL __cdecl KP_PCI_Open_32_64(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
00027     PVOID pOpenData, PVOID *ppDrvContext);
00028 /* @32on64@ */
00029 void __cdecl KP_PCI_Close(PVOID pDrvContext);
00030 void __cdecl KP_PCI_Call(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall);
00031 BOOL __cdecl KP_PCI_IntEnable(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall,
00032     PVOID *ppIntContext);
00033 void __cdecl KP_PCI_IntDisable(PVOID pIntContext);
00034 BOOL __cdecl KP_PCI_IntAtIrql(PVOID pIntContext, BOOL *pfIsMyInterrupt);
00035 DWORD __cdecl KP_PCI_IntAtDpc(PVOID pIntContext, DWORD dwCount);
00036 BOOL __cdecl KP_PCI_IntAtIrqlMSI(PVOID pIntContext, ULONG dwLastMessage,
00037     DWORD dwReserved);
00038 DWORD __cdecl KP_PCI_IntAtDpcMSI(PVOID pIntContext, DWORD dwCount,
00039     ULONG dwLastMessage, DWORD dwReserved);
00040 #ifndef ISA
00041 BOOL __cdecl KP_PCI_Event(PVOID pDrvContext, WD_EVENT *wd_event);
00042 #endif /* ifndef ISA */
00043 static void KP_PCI_Err(const CHAR *sFormat, ...);
00044 static void KP_PCI_Trace(const CHAR *sFormat, ...);
00045 /* @32on64@ */
00046 #define PTR32 UINT32
00047
00048 typedef struct {
00049     UINT32 dwNumAddrSpaces; /* Total number of device address spaces */
00050     PTR32 pAddrDesc; /* Array of device address spaces information */
00051 } PCI_DEV_ADDR_DESC_32B;
00052 /* @32on64@ */
00053
00054 ****
00055 Functions implementation
00056 ****
00057
00065 BOOL __cdecl KP_Init(KP_INIT *kpInit)
00066 {
00067     /* Verify that the version of the WinDriver Kernel PlugIn library
00068      is identical to that of the windrvr.h and wd_kp.h files */
00069     if (WD_VER != kpInit->dwVerWD)
00070     {
00071         /* Rebuild your Kernel PlugIn driver project with the compatible
00072          version of the WinDriver Kernel PlugIn library (kp_nt<version>.lib)
00073          and windrvr.h and wd_kp.h files */
00074
00075         return FALSE;
00076     }
00077 }
```

```
00078     kpInit->funcOpen = KP_PCI_Open;
00079 /* @32on64@ */
00080     kpInit->funcOpen_32_64 = KP_PCI_Open_32_64;
00081 /* @32on64@ */
00082 #if defined(WINNT)
00083     strcpy(kpInit->cDriverName, KP_PCI_DRIVER_NAME);
00084 #else
00085     strncpy(kpInit->cDriverName, KP_PCI_DRIVER_NAME,
00086             sizeof(kpInit->cDriverName));
00087 #endif
00088     kpInit->cDriverName[sizeof(kpInit->cDriverName) - 1] = 0;
00089
00090     return TRUE;
00091 }
00092
00154 BOOL __cdecl KP_PCI_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD, PVOID pOpenData,
00155     PVOID *ppDrvContext)
00156 {
00157     PCI_DEV_ADDR_DESC *pDevAddrDesc;
00158     DWORD dwSize;
00159     DWORD dwStatus;
00160
00161     /* Initialize the PCI library */
00162     dwStatus = PCI_LibInit();
00163     if (WD_STATUS_SUCCESS != dwStatus)
00164     {
00165         KP_PCI_Err("KP_PCI_Open: Failed to initialize the PCI library. "
00166                 "Last error [%s]\n", PCI_GetLastErr());
00167         return FALSE;
00168     }
00169
00170     KP_PCI_Trace("KP_PCI_Open: Entered. PCI library initialized\n");
00171
00172     kpOpenCall->funcClose = KP_PCI_Close;
00173     kpOpenCall->funcCall = KP_PCI_Call;
00174     kpOpenCall->funcIntEnable = KP_PCI_IntEnable;
00175     kpOpenCall->funcIntDisable = KP_PCI_IntDisable;
00176     kpOpenCall->funcIntAtIrql = KP_PCI_IntAtIrql;
00177     kpOpenCall->funcIntAtDpc = KP_PCI_IntAtDpc;
00178     kpOpenCall->funcIntAtIrqlMSI = KP_PCI_IntAtIrqlMSI;
00179     kpOpenCall->funcIntAtDpcMSI = KP_PCI_IntAtDpcMSI;
00180 #ifndef ISA
00181     kpOpenCall->funcEvent = KP_PCI_Event;
00182 #endif /* ifndef ISA */
00183
00184     if (ppDrvContext)
00185     {
00186         if (pOpenData)
00187         {
00188             WDC_ADDR_DESC *pAddrDesc;
00189
00190             /* Create a copy of device information in the driver context */
00191             dwSize = sizeof(PCI_DEV_ADDR_DESC);
00192             pDevAddrDesc = malloc(dwSize);
00193             if (!pDevAddrDesc)
00194                 goto malloc_error;
00195
00196             COPY_FROM_USER(pDevAddrDesc, pOpenData, dwSize);
00197
00198             dwSize = sizeof(WDC_ADDR_DESC) * pDevAddrDesc->dwNumAddrSpaces;
00199             pAddrDesc = malloc(dwSize);
00200             if (!pAddrDesc)
00201                 goto malloc_error;
00202
00203             COPY_FROM_USER(pAddrDesc, pDevAddrDesc->pAddrDesc, dwSize);
00204             pDevAddrDesc->pAddrDesc = pAddrDesc;
00205
00206             *ppDrvContext = pDevAddrDesc;
00207         }
00208     else
00209     {
00210         *ppDrvContext = NULL;
00211     }
00212 }
00213
00214 KP_PCI_Trace("KP_PCI_Open: Kernel PlugIn driver opened successfully\n");
00215
00216     return TRUE;
00217
00218 malloc_error:
00219     KP_PCI_Err("KP_PCI_Open: Failed allocating [%ld] bytes\n", dwSize);
00220     if (pDevAddrDesc)
00221         free(pDevAddrDesc);
00222     PCI_LibUninit();
00223     return FALSE;
00224 }
00225
```

```
00226 /* @32on64@ */
00227
00228 BOOL __cdecl KP_PCI_Open_32_64(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
00229     PVOID pOpenData, PVOID *ppDrvContext)
00230 {
00231     PCI_DEV_ADDR_DESC *pDevAddrDesc;
00232     DWORD dwSize;
00233     DWORD dwStatus;
00234
00235     /* Initialize the PCI library */
00236     dwStatus = PCI_LibInit();
00237     if (WD_STATUS_SUCCESS != dwStatus)
00238     {
00239         KP_PCI_Err("KP_PCI_Open_32_64: Failed to initialize the PCI library. "
00240             "Last error [%s]\n", PCI_GetLastErr());
00241         return FALSE;
00242     }
00243
00244     KP_PCI_Trace("KP_PCI_Open_32_64: Entered. PCI library initialized\n");
00245
00246     kpOpenCall->funcClose = KP_PCI_Close;
00247     kpOpenCall->funcCall = KP_PCI_Call;
00248     kpOpenCall->funcIntEnable = KP_PCI_IntEnable;
00249     kpOpenCall->funcIntDisable = KP_PCI_IntDisable;
00250     kpOpenCall->funcIntAtIrql = KP_PCI_IntAtIrql;
00251     kpOpenCall->funcIntAtDpc = KP_PCI_IntAtDpc;
00252     kpOpenCall->funcIntAtIrqlMSI = KP_PCI_IntAtIrqlMSI;
00253     kpOpenCall->funcIntAtDpcMSI = KP_PCI_IntAtDpcMSI;
00254
00255 #ifndef ISA
00256     kpOpenCall->funcEvent = KP_PCI_Event;
00257 #endif /* ifndef ISA */
00258
00259     if (ppDrvContext)
00260     {
00261         if (pOpenData)
00262         {
00263             PCI_DEV_ADDR_DESC_32B devAddrDesc_32;
00264             WDC_ADDR_DESC *pAddrDesc;
00265
00266             /* Create a copy of the device information in the driver context */
00267             dwSize = sizeof(PCI_DEV_ADDR_DESC);
00268             pDevAddrDesc = malloc(dwSize);
00269             if (!pDevAddrDesc)
00270                 goto malloc_error;
00271
00272             /* Copy device information sent from a 32-bit user application */
00273             COPY_FROM_USER(&devAddrDesc_32, pOpenData,
00274                           sizeof(PCI_DEV_ADDR_DESC_32B));
00275
00276             /* Copy the 32-bit data to a 64-bit struct */
00277             pDevAddrDesc->dwNumAddrSpaces = devAddrDesc_32.dwNumAddrSpaces;
00278             dwSize = sizeof(WDC_ADDR_DESC) * pDevAddrDesc->dwNumAddrSpaces;
00279             pAddrDesc = malloc(dwSize);
00280             if (!pAddrDesc)
00281                 goto malloc_error;
00282
00283             COPY_FROM_USER(pAddrDesc, (PVOID)(KPTR)devAddrDesc_32.pAddrDesc,
00284                           dwSize);
00285             pDevAddrDesc->pAddrDesc = pAddrDesc;
00286
00287             *ppDrvContext = pDevAddrDesc;
00288         }
00289     }
00290     else
00291     {
00292         *ppDrvContext = NULL;
00293     }
00294 }
00295
00296 KP_PCI_Trace("KP_PCI_Open_32_64: Kernel PlugIn driver opened "
00297     "successfully\n");
00298
00299 return TRUE;
00300
00301 malloc_error:
00302     KP_PCI_Err("KP_PCI_Open_32_64: Failed allocating [%ld] bytes\n", dwSize);
00303     if (pDevAddrDesc)
00304         free(pDevAddrDesc);
00305     PCI_LibUninit();
00306     return FALSE;
00307 }
00308 /* @32on64@ */
00309
00310 void __cdecl KP_PCI_Close(PVOID pDrvContext)
00311 {
00312     DWORD dwStatus;
00313
00314     KP_PCI_Trace("KP_PCI_Close: Entered\n");
00315 }
```

```
00331     /* Uninit the PCI library */
00332     dwStatus = PCI_LibUninit();
00333     if (WD_STATUS_SUCCESS != dwStatus)
00334     {
00335         KP_PCI_Err("KP_PCI_Close: Failed to uninit the PCI library. "
00336                     "Last error [%s]\n", PCI_GetLastErr());
00337     }
00338
00339
00340     /* Free the memory allocated for the driver context */
00341     if (pDrvContext)
00342     {
00343         if (((PCI_DEV_ADDR_DESC *)pDrvContext)->pAddrDesc)
00344             free(((PCI_DEV_ADDR_DESC *)pDrvContext)->pAddrDesc);
00345         free(pDrvContext);
00346     }
00347 }
00348
00349 void __cdecl KP_PCI_Call(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall)
00350 {
00351     KP_PCI_Trace("KP_PCI_Call: Entered. Message [0x%lx]\n", kpCall->dwMessage);
00352
00353     kpCall->dwResult = KP_PCI_STATUS_OK;
00354
00355     switch (kpCall->dwMessage)
00356     {
00357         case KP_PCI_MSG_VERSION: /* Get the version of the Kernel PlugIn driver */
00358         {
00359             KP_PCI_VERSION *pUserKPVer = (KP_PCI_VERSION *) (kpCall->pData);
00360             KP_PCI_VERSION kernelKPVer;
00361
00362             BZERO(kernelKPVer);
00363             kernelKPVer.dwVer = 100;
00364 #define DRIVER_VER_STR "My Driver V1.00"
00365             memcpy(kernelKPVer.cVer, DRIVER_VER_STR, sizeof(DRIVER_VER_STR));
00366             COPY_TO_USER(pUserKPVer, &kernelKPVer, sizeof(KP_PCI_VERSION));
00367             kpCall->dwResult = KP_PCI_STATUS_OK;
00368         }
00369         break;
00370
00371         default:
00372             kpCall->dwResult = KP_PCI_STATUS_MSG_NO_IMPL;
00373     }
00374
00375     /* NOTE: You can modify the messages above and/or add your own
00376      Kernel PlugIn messages.
00377      When changing/adding messages, be sure to also update the
00378      messages definitions in ../pci_lib.h. */
00379 }
00402
00403     /* You can allocate specific memory for each interrupt in *ppIntContext */
00404
00405     /* In this sample we will set the interrupt context to the driver context,
00406      which has been set in KP_PCI_Open to hold the device information. */
00407     *ppIntContext = pDrvContext;
00408
00409     /* TODO: You can add code here to write to the device in order
00410      to physically enable the hardware interrupts */
00411
00412     return TRUE;
00413 }
00444
00456 void __cdecl KP_PCI_IntDisable(PVOID pIntContext)
00457 {
00458     /* Free any memory allocated in KP_PCI_IntEnable() here */
00459 }
00495 BOOL __cdecl KP_PCI_IntAtIrql(PVOID pIntContext, BOOL *pfIsMyInterrupt)
00496 {
00497     static DWORD dwIntCount = 0; /* Interrupts count */
00498     PCI_DEV_ADDR_DESC *pDevAddrDesc = (PCI_DEV_ADDR_DESC *)pIntContext;
00499 /* @kp_IntAtIrql@ */
00500 /* @sample_only_start@ */
00501 #ifndef ISA
00502     WDC_ADDR_DESC *pAddrDesc = &pDevAddrDesc->pAddrDesc[INTCSR_ADDR_SPACE];
00503
00504 #define USE_MULTI_TRANSFER
00505 #if defined USE_MULTI_TRANSFER
00506     /* Define the number of interrupt transfer commands to use */
00507     WD_TRANSFER trans[2];
00508
00509     /*
00510         This sample demonstrates how to set up two transfer commands, one for
00511         reading the device's INTCSR register (as defined in gPCI_Regs) and one
```

```
00512     for writing to it to acknowledge the interrupt.  
00513  
00514     TODO: PCI interrupts are level sensitive interrupts and must be  
00515         acknowledged in the kernel immediately when they are received.  
00516         Since the information for acknowledging the interrupts is  
00517         hardware-specific, YOU MUST MODIFY THE CODE below and set up  
00518         transfer commands in order to correctly acknowledge the interrupts  
00519         on your device, as dictated by your hardware's specifications.  
00520  
00521     ****  
00522     * NOTE: If you attempt to use this code without first modifying it in    *  
00523     *      order to correctly acknowledge your device's interrupts, as      *  
00524     *      explained above, the OS will HANG when an interrupt occurs!    *  
00525     ****  
00526 */  
00527  
00528     EZERO(trans);  
00529  
00530     /* Prepare the interrupt transfer commands */  
00531  
00532     /* #1: Read status from the INTCSR register */  
00533     trans[0].pPort = pAddrDesc->pAddr + INTCSR;  
00534     /* 32bit read: */  
00535     trans[0].cmdTrans = WDC_ADDR_IS_MEM(pAddrDesc) ? RM_DWORD : RP_DWORD;  
00536  
00537     /* #2: Write ALL_INT_MASK to the INTCSR register to acknowledge the  
00538         interrupt */  
00539     /* In this example both commands access the same address (register): */  
00540     trans[1].pPort = trans[0].pPort;  
00541     /* 32bit write: */  
00542     trans[1].cmdTrans = WDC_ADDR_IS_MEM(pAddrDesc) ? WM_DWORD : WP_DWORD;  
00543     trans[1].Data.Dword = ALL_INT_MASK;  
00544  
00545     /* Execute the transfer commands */  
00546     WDC_MultiTransfer(trans, 2);  
00547 #else  
00548     /* NOTE: For memory registers you can replace the use of WDC_MultiTransfer()  
00549     (or any other WD_xxx/WDC_xxx read/write function call) with direct  
00550     memory access. For example, if INTCSR is a memory register, the code  
00551     above can be replaced with the following: */  
00552  
00553     UINT32 readData;  
00554     PVOID pData = (DWORD *) (pAddrDesc->pAddr + INTCSR);  
00555  
00556     /* Read status from the PCI_INTCSR register */  
00557     readData = WDC_ReadMem32(pData, 0);  
00558  
00559     /* Write to the PCI_INTCSR register to acknowledge the interrupt */  
00560     WDC_WriteMem32(pData, 0, ALL_INT_MASK);  
00561 #endif  
00562 #undef USE_MULTI_TRANSFER  
00563  
00564     /* If the data read from the hardware indicates that the interrupt belongs  
00565         to you, you must set *pfIsMyInterrupt to TRUE.  
00566         Otherwise, set it to FALSE (this will let ISR's of other drivers be  
00567         invoked). */  
00568     *pfIsMyInterrupt = FALSE;  
00569 #endif /* ifndef ISA */  
00570 /* @sample_only_end@ */  
00571     /* This sample schedules a DPC once in every 5 interrupts.  
00572         TODO: You can modify the implementation to schedule the DPC as needed. */  
00573     dwIntCount++;  
00574     if (!(dwIntCount % 5))  
00575         return TRUE;  
00576  
00577     return FALSE;  
00578 }  
00579  
00602 DWORD __cdecl KP_PCI_IntAtDpc(PVOID pIntContext, DWORD dwCount)  
00603 {  
00604     return dwCount;  
00605 }  
00643 BOOL __cdecl KP_PCI_IntAtIrqlMSI(PVOID pIntContext, ULONG dwLastMessage,  
00644     DWORD dwReserved)  
00645 {  
00646     static DWORD dwIntCount = 0; /* Interrupts count */  
00647  
00648     /* There is no need to acknowledge MSI/MSI-X. However, you can implement  
00649         the same functionality here as done in the KP_PCI_IntAtIrql handler  
00650         to read/write data from/to registers at HIGH IRQL. */  
00651  
00652     /* This sample schedules a DPC once in every 5 interrupts.  
00653         TODO: You can modify the implementation to schedule the DPC as needed. */  
00654     dwIntCount++;  
00655     if !(dwIntCount % 5))  
00656         return TRUE;  
00657 }
```

```
00658     return FALSE;
00659 }
00660
00688 DWORD __cdecl KP_PCI_IntAtDpcMSI(PVOID pIntContext, DWORD dwCount,
00689     ULONG dwLastMessage, DWORD dwReserved)
00690 {
00691     return dwCount;
00692 }
00693
00694 #ifndef ISA
00695
00712 BOOL __cdecl KP_PCI_Event(PVOID pDrvContext, WD_EVENT *wd_event)
00713 {
00714     return TRUE; /* Return TRUE to notify the user mode of the event */
00715 }
00716 #endif /* ifndef ISA */
00717
00718 /* -----
00719     Debugging and error handling
00720 ----- */
00721 static void KP_PCI_Err(const CHAR *sFormat, ...)
00722 {
00723 #if defined(DEBUG)
00724     CHAR sMsg[256];
00725     va_list argp;
00726
00727     va_start(argp, sFormat);
00728     vsnprintf(sMsg, sizeof(sMsg) - 1, sFormat, argp);
00729     WDC_Err("%s: %s", KP_PCI_DRIVER_NAME, sMsg);
00730     va_end(argp);
00731 #endif
00732 }
00733
00734 static void KP_PCI_Trace(const CHAR *sFormat, ...)
00735 {
00736 #if defined(DEBUG)
00737     CHAR sMsg[256];
00738     va_list argp;
00739
00740     va_start(argp, sFormat);
00741     vsnprintf(sMsg, sizeof(sMsg) - 1, sFormat, argp);
00742     WDC_Trace("%s: %s", KP_PCI_DRIVER_NAME, sMsg);
00743     va_end(argp);
00744 #endif
00745 }
00746
```


Index

_In
 windrvr.h, 516
_Inout
 windrvr.h, 516
_Out
 windrvr.h, 516
_Outptr
 windrvr.h, 516
__ALIGN_DOWN
 windrvr.h, 516
__ALIGN_UP
 windrvr.h, 516
__FUNCTION
 windrvr.h, 516
__KERNEL
 kpstdlib.h, 297
 wd_kp.h, 386
__KERPLUG
 wd_kp.h, 386
~CCString
 CCString, 204

AD_PCI_BAR
 pci_regs.h, 365
AD_PCI_BAR0
 pci_regs.h, 365
AD_PCI_BAR1
 pci_regs.h, 365
AD_PCI_BAR2
 pci_regs.h, 365
AD_PCI_BAR3
 pci_regs.h, 365
AD_PCI_BAR4
 pci_regs.h, 365
AD_PCI_BAR5
 pci_regs.h, 365
AD_PCI_BARS
 pci_regs.h, 365
applications_num
 WD_USAGE, 255

bAlternateSetting
 WDU_INTERFACE_DESCRIPTOR, 279
bcdDevice
 WDU_DEVICE_DESCRIPTOR, 270
bcdUSB
 WDU_DEVICE_DESCRIPTOR, 270
bConfigurationValue
 WDU_CONFIGURATION_DESCRIPTOR, 267
bDescriptorType
 WDU_CONFIGURATION_DESCRIPTOR, 267
 WDU_DEVICE_DESCRIPTOR, 271
 WDU_ENDPOINT_DESCRIPTOR, 273
 WDU_INTERFACE_DESCRIPTOR, 279
bDeviceClass
 WDU_DEVICE_DESCRIPTOR, 271
 WDU_MATCH_TABLE, 281

bDeviceProtocol
 WDU_DEVICE_DESCRIPTOR, 271
bDeviceSubClass
 WDU_DEVICE_DESCRIPTOR, 271
 WDU_MATCH_TABLE, 281
bEndpointAddress
 WDU_ENDPOINT_DESCRIPTOR, 273
bIndex
 WDU_GET_DESCRIPTOR, 275
bInterfaceClass
 WDU_INTERFACE_DESCRIPTOR, 280
 WDU_MATCH_TABLE, 282
bInterfaceNumber
 WDU_INTERFACE_DESCRIPTOR, 280
bInterfaceProtocol
 WDU_INTERFACE_DESCRIPTOR, 280
 WDU_MATCH_TABLE, 282
bInterfaceSubClass
 WDU_INTERFACE_DESCRIPTOR, 280
 WDU_MATCH_TABLE, 282
bInterval
 WDU_ENDPOINT_DESCRIPTOR, 273
BIT
 bits.h, 293
BIT0
 bits.h, 293
BIT1
 bits.h, 293
BIT10
 bits.h, 293
BIT11
 bits.h, 293
BIT12
 bits.h, 293
BIT13
 bits.h, 293
BIT14
 bits.h, 293
BIT15
 bits.h, 293
BIT16
 bits.h, 293
BIT17
 bits.h, 293
BIT18
 bits.h, 293
BIT19
 bits.h, 293
BIT2
 bits.h, 293
BIT20
 bits.h, 293
BIT21
 bits.h, 293
BIT22
 bits.h, 293

BIT23
 bits.h, 293

BIT24
 bits.h, 294

BIT25
 bits.h, 294

BIT26
 bits.h, 294

BIT27
 bits.h, 294

BIT28
 bits.h, 294

BIT29
 bits.h, 294

BIT3
 bits.h, 293

BIT30
 bits.h, 294

BIT31
 bits.h, 294

BIT4
 bits.h, 293

BIT5
 bits.h, 293

BIT6
 bits.h, 293

BIT7
 bits.h, 293

BIT8
 bits.h, 293

BIT9
 bits.h, 293

bLength
 WDU_CONFIGURATION_DESCRIPTOR, 267

 WDU_DEVICE_DESCRIPTOR, 271

 WDU_ENDPOINT_DESCRIPTOR, 273

 WDU_INTERFACE_DESCRIPTOR, 280

bmAttributes
 WDU_CONFIGURATION_DESCRIPTOR, 267

 WDU_ENDPOINT_DESCRIPTOR, 274

bMaxPacketSize0
 WDU_DEVICE_DESCRIPTOR, 271

bNumConfigurations
 WDU_DEVICE_DESCRIPTOR, 271

bNumEndpoints
 WDU_INTERFACE_DESCRIPTOR, 280

bNumInterfaces
 WDU_CONFIGURATION_DESCRIPTOR, 267

bType
 WDU_GET_DESCRIPTOR, 275

Bus
 WD_ITEMS, 235

BYTE
 windrvr.h, 557

Byte
 WD_TRANSFER, 253

BZERO
 windrvr.h, 517

CAP_REG
 pci_regs.h, 366

Card
 WD_CARD_REGISTER, 216

 WD_PCI_CARD_INFO, 244

cardId
 WD_EVENT, 224

 WD_PCI_SCAN_CARDS, 249

cardReg
 WDC_DEVICE, 258

cardSlot
 WD_PCI_SCAN_CARDS, 249

cat_printf
 CCString, 204

cBuffer
 WD_DEBUG_DUMP, 219

cBuild
 WD_OS_INFO, 242

cCsdVersion
 WD_OS_INFO, 242

CCString, 203
 ~CCString, 204

 cat_printf, 204

 CCString, 204

Compare, 205
CompareNoCase, 205
contains, 205
find_first, 205
find_last, 205
Format, 205
GetBuffer, 205
Init, 205
is_empty, 206
IsAllocOK, 206
Length, 206
m_buf_size, 209
m_str, 209
MakeLower, 206
MakeUpper, 206
Mid, 206
operator char *, 206
operator PCSTR, 206
operator!=, 207
operator+=, 207
operator=, 207
operator==, 207
operator[], 208
sprintf, 208
strcmp, 208
stricmp, 208
StrRemove, 208
StrReplace, 208
substr, 208
tolower, 208
tolower_copy, 208
toupper, 209
toupper_copy, 209
trim, 209
vsprintf, 209
cCurrentVersion
 WD_OS_INFO, 242
cDescription
 WD_CARD_REGISTER, 216
cDriverName
 KP_INIT, 210
 WD_KERNEL_PLUGIN, 240
cDriverPath
 WD_KERNEL_PLUGIN, 240
check_secureBoot_enabled
 windrvr.h, 573
cInstallationType
 WD_OS_INFO, 243
cLicense
 WD_LICENSE, 242
Cmd
 WD_CARD_CLEANUP, 214
 WD_INTERRUPT, 228
CMD_END
 windrvr.h, 573
CMD_MASK
 windrvr.h, 573
CMD_NONE

windrvr.h, 573
cmdTrans
 WD_TRANSFER, 253
cName
 WD_CARD_REGISTER, 216
Compare
 CCString, 205
CompareNoCase
 CCString, 205
contains
 CCString, 205
COPY_FROM_USER_OR_KERNEL
 kpstdlib.h, 297
COPY_TO_USER_OR_KERNEL
 kpstdlib.h, 298
COPYRIGHTS_FULL_STR
 wd_ver.h, 394
COPYRIGHTS_YEAR_STR
 wd_ver.h, 394
cProcessName
 WD_IPC_PROCESS, 231
cProdName
 WD_OS_INFO, 243
cstring.h, 294, 295
 operator+, 295
 PCSTR, 295
 stricmp, 295
CTL_CODE
 windrvr.h, 517
cVer
 WD_VERSION, 255
D_ERROR
 windrvr.h, 564
D_INFO
 windrvr.h, 564
D_OFF
 windrvr.h, 564
D_TRACE
 windrvr.h, 564
D_WARN
 windrvr.h, 564
Data
 WD_TRANSFER, 253
DEBUG_CLEAR_BUFFER
 windrvr.h, 564
DEBUG_CLOCK_RESET
 windrvr.h, 564
DEBUG_COMMAND
 windrvr.h, 563
DEBUG_DUMP_CLOCK_OFF
 windrvr.h, 564
DEBUG_DUMP_CLOCK_ON
 windrvr.h, 564
DEBUG_DUMP_SEC_OFF
 windrvr.h, 564
DEBUG_DUMP_SEC_ON
 windrvr.h, 564
DEBUG_LEVEL

windrvr.h, 564
DEBUG_SECTION
 windrvr.h, 564
DEBUG_SET_BUFFER
 windrvr.h, 563
DEBUG_SET_FILTER
 windrvr.h, 563
DEBUG_STATUS
 windrvr.h, 563
DEBUG_USER_BUF_LEN
 windrvr.h, 517
Descriptor
 WDU_ALTERNATE_SETTING, 265
 WDU_CONFIGURATION, 266
 WDU_DEVICE, 269
DEV_CAPS
 pci_regs.h, 366
DEV_CAPS2
 pci_regs.h, 367
DEV_CTL
 pci_regs.h, 366
DEV_CTL2
 pci_regs.h, 367
DEV_STS
 pci_regs.h, 366
DEV_STS2
 pci_regs.h, 367
deviceId
 WDC_PCI_SCAN_RESULT, 262
devices_num
 WD_USAGE, 255
deviceSlot
 WDC_PCI_SCAN_RESULT, 262
direction
 WDU_PIPE_INFO, 283
DLLCALLCONV
 windrvr.h, 517
DMA_ADDR
 windrvr.h, 557
DMA_ADDRESS_WIDTH_MASK
 windrvr.h, 517
DMA_ALLOW_64BIT_ADDRESS
 windrvr.h, 565
DMA_ALLOW_CACHE
 windrvr.h, 565
DMA_ALLOW_NO_HCARD
 windrvr.h, 565
DMA_BIT_MASK
 windrvr.h, 517
DMA_DIRECTION_MASK
 windrvr.h, 517
DMA_DISABLE_MERGE_ADJACENT_PAGES
 windrvr.h, 566
DMA_FROM_DEVICE
 windrvr.h, 565
DMA_GET_EXISTING_BUF
 windrvr.h, 565
DMA_GET_PREALLOCATED_BUFFERS_ONLY
 windrvr.h, 566
DMA_GPUDIRECT
 windrvr.h, 566
DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH
 windrvr.h, 566
DMA_KBUF_BELOW_16M
 windrvr.h, 565
DMA_KERNEL_BUFFER_ALLOC
 windrvr.h, 565
DMA_KERNEL_ONLY_MAP
 windrvr.h, 565
DMA_LARGE_BUFFER
 windrvr.h, 565
DMA_OPTIONS_ADDRESS_WIDTH_SHIFT
 windrvr.h, 517
DMA_OPTIONS_ALL
 windrvr.h, 518
DMA_READ_FROM_DEVICE
 windrvr.h, 518
DMA_RESERVED_MEM
 windrvr.h, 565
DMA_TO_DEVICE
 windrvr.h, 565
DMA_TO_FROM_DEVICE
 windrvr.h, 565
DMA_TRANSACTION
 windrvr.h, 566
DMA_TRANSACTION_CALLBACK
 windrvr.h, 557
DMA_WRITE_TO_DEVICE
 windrvr.h, 518
DMATransactionCallback
 WD_DMA, 220
DMATransactionCallbackCtx
 WD_DMA, 220
DRIVER_VER_STR
 kp_pci.c, 611
dwAction
 WD_EVENT, 224
dwAddrSpace
 WDC_ADDR_DESC, 256
dwAlignment
 WD_DMA, 220
dwAlternateSetting
 WDU_SET_INTERFACE, 286
dwBar
 WD_ITEMS, 235
dwBufferSize
 WD_DEBUG, 217
 WDU_STREAM, 287
 WDU_TRANSFER, 290
dwBus
 WD_PCI_SLOT, 250
dwBusNum
 WD_BUS, 213
dwBusType
 WD_BUS, 213
dwBytes

WD_DMA, 220
WD_DMA_PAGE, 222
WD_GET_DEVICE_PROPERTY, 227
WD_ITEMS, 235
WD_PCI_CONFIG_DUMP, 245
WD_TRANSFER, 253
WDU_GET_DEVICE_DATA, 276
dwBytesInBuffer
 WDU_STREAM_STATUS, 289
dwBytesTransferred
 WD_DMA, 221
 WDU_TRANSFER, 290
dwCapId
 WD_PCI_CAP, 244
 WD_PCI_SCAN_CAPS, 247
dwCapOffset
 WD_PCI_CAP, 244
dwCards
 WD_PCI_SCAN_CARDS, 249
dwCmd
 WD_DEBUG, 217
dwCmds
 WD_CARD_CLEANUP, 215
 WD_INTERRUPT, 229
dwCounter
 WD_INTERRUPT, 229
dwDeviceId
 WD_PCI_ID, 246
dwDomain
 WD_PCI_SLOT, 250
dwDomainNum
 WD_BUS, 213
dwEnabledIntType
 WD_INTERRUPT, 229
dwEventId
 WD_EVENT, 224
dwEventType
 WD_EVENT, 224
dwFunction
 WD_PCI_SLOT, 250
dwGroupId
 WD_EVENT, 224
 WD_IPC_PROCESS, 231
dwInterfaceNum
 WDU_SET_INTERFACE, 286
dwInterrupt
 WD_ITEMS, 235
dwInterval
 WDU_PIPE_INFO, 283
dwItemIndex
 WDC_ADDR_DESC, 256
dwItems
 WD_CARD, 214
dwLastError
 WDU_STREAM_STATUS, 289
dwLastMessage
 WD_INTERRUPT, 229
dwLevel
 WD_DEBUG, 217
 WD_DEBUG_ADD, 218
dwLevelMessageBox
 WD_DEBUG, 218
dwLost
 WD_INTERRUPT, 229
dwMajorVersion
 WD_OS_INFO, 243
dwMaximumPacketSize
 WDU_PIPE_INFO, 283
dwMaxTransferSize
 WD_DMA, 221
dwMessage
 WD_KERNEL_PLUGIN_CALL, 241
dwMicroSeconds
 WD_SLEEP, 252
dwMinorVersion
 WD_OS_INFO, 243
dwMsgID
 WD_EVENT, 224
 WD_IPC_SEND, 233
 WDS_IPC_MSG_RX, 263
dwNumAddrSpaces
 WDC_DEVICE, 258
dwNumAltSettings
 WDU_INTERFACE, 278
dwNumber
 WDU_PIPE_INFO, 283
dwNumCaps
 WD_PCI_SCAN_CAPS, 247
 WDC_PCI_SCAN_CAPS_RESULT, 261
dwNumCmds
 WDC_INTERRUPT_PARAMS, 260
dwNumDevices
 WDC_PCI_SCAN_RESULT, 262
dwNumInterfaces
 WDU_CONFIGURATION, 266
dwNumMatchTables
 WD_EVENT, 224
dwNumProcs
 WD_IPC_SCAN_PROCS, 232
 WDS_IPC_SCAN_RESULT, 264
dwNumVFs
 WD_PCI_SRIOV, 251
dwOffset
 WD_PCI_CONFIG_DUMP, 245
dwOptions
 WD_CARD_CLEANUP, 215
 WD_CARD_REGISTER, 216
 WD_DMA, 221
 WD_EVENT, 225
 WD_GET_DEVICE_PROPERTY, 227
 WD_INTERRUPT, 229
 WD_IPC_REGISTER, 232
 WD_IPC_SEND, 233
 WD_ITEMS, 236
 WD_KERNEL_BUFFER, 238
 WD_PCI_SCAN_CAPS, 248

WD_PCI_SCAN_CARDS, 249
WD_SLEEP, 252
WD_TRANSFER, 253
WDC_INTERRUPT_PARAMS, 260
WDU_GET_DEVICE_DATA, 277
WDU_HALT_TRANSFER, 277
WDU_RESET_DEVICE, 284
WDU_RESET_PIPE, 285
WDU_SELECTIVE_SUSPEND, 286
WDU_SET_INTERFACE, 286
WDU_STREAM, 287
WDU_STREAM_STATUS, 289
WDU_TRANSFER, 290
WDU_WAKEUP, 292
Dword
 WD_TRANSFER, 254
dwPages
 WD_DMA, 221
dwPipeNum
 WDU_HALT_TRANSFER, 277
 WDU_RESET_PIPE, 285
 WDU_STREAM, 287
 WDU_TRANSFER, 290
dwProperty
 WD_GET_DEVICE_PROPERTY, 227
dwRecipientID
 WD_IPC_SEND, 233
dwReserved
 WDU_STREAM, 288
 WDU_STREAM_STATUS, 289
dwReserved1
 WD_ITEMS, 236
dwResult
 WD_KERNEL_PLUGIN_CALL, 241
 WD_PCI_CONFIG_DUMP, 245
dwRxSize
 WDU_STREAM, 288
dwRxTxTimeout
 WDU_STREAM, 288
dwSection
 WD_DEBUG, 218
 WD_DEBUG_ADD, 218
dwSenderId
 WD_EVENT, 225
 WDS_IPC_MSG_RX, 263
dwSlot
 WD_PCI_SLOT, 250
dwSlotFunc
 WD_BUS, 213
dwSubGroupID
 WD_EVENT, 225
 WD_IPC_PROCESS, 231
dwTimeout
 WDU_TRANSFER, 291
dwTransferElementSize
 WD_DMA, 221
dwUniqueID
 WD_EVENT, 225
WD_GET_DEVICE_PROPERTY, 227
WDU_DESCRIPTOR, 276
WDU_GET_DEVICE_DATA, 277
WDU_HALT_TRANSFER, 278
WDU_RESET_DEVICE, 284
WDU_RESET_PIPE, 285
WDU_SELECTIVE_SUSPEND, 286
WDU_SET_INTERFACE, 287
WDU_STREAM, 288
WDU_STREAM_STATUS, 289
WDU_TRANSFER, 291
WDU_WAKEUP, 292
dwVendorId
 WD_PCI_ID, 247
dwVer
 WD_VERSION, 256
dwVerWD
 KP_INIT, 210
Event
 WDC_DEVICE, 258
event_handle_t
 windrvr_events.h, 597
EVENT_HANDLER
 windrvr_events.h, 597
EventAlloc
 windrvr_events.h, 597
EventDup
 windrvr_events.h, 598
EventFree
 windrvr_events.h, 598
EventRegister
 windrvr_events.h, 598
EventUnregister
 windrvr_events.h, 598
FALSE
 kpstdlib.h, 298
fAutoinc
 WD_TRANSFER, 254
fBlocking
 WDU_STREAM, 288
fCheckLockOnly
 WD_CARD_REGISTER, 216
fEnableOk
 WD_INTERRUPT, 229
FILE_ANY_ACCESS
 windrvr.h, 518
FILE_READ_ACCESS
 windrvr.h, 518
FILE_WRITE_ACCESS
 windrvr.h, 518
find_first
 CCString, 205
find_last
 CCString, 205
flsMemory
 WDC_ADDR_DESC, 257
flsRead

WD_PCI_CONFIG_DUMP, 246
fIsRunning
 WDU_STREAM_STATUS, 289
fNotSharable
 WD_ITEMS, 236
Format
 CCString, 205
fRead
 WDU_TRANSFER, 291
free
 kpstdlib.h, 299
fStopped
 WD_INTERRUPT, 230
FUNC_MASK
 windrvr.h, 518
funcCall
 KP_OPEN_CALL, 211
funcClose
 KP_OPEN_CALL, 211
funcEvent
 KP_OPEN_CALL, 211
funcIntAtDpc
 KP_OPEN_CALL, 211
funcIntAtDpcMSI
 KP_OPEN_CALL, 211
funcIntAtIrql
 KP_OPEN_CALL, 212
funcIntAtIrqlMSI
 KP_OPEN_CALL, 212
funcIntDisable
 KP_OPEN_CALL, 212
funcIntEnable
 KP_OPEN_CALL, 212
funcIntHandler
 WDC_INTERRUPT_PARAMS, 260
funcOpen
 KP_INIT, 210
funcOpen_32_64
 KP_INIT, 210
fUseKP
 WDC_INTERRUPT_PARAMS, 260

GET_CAPABILITY_STR
 pci_regs.h, 318
GET_EXTENDED_CAPABILITY_STR
 pci_regs.h, 319
get_os_type
 windrvr.h, 574
GetBuffer
 CCString, 205
GetNumberOfProcessors
 utils.h, 378
GetPageSize
 utils.h, 378

h
 WD_GET_DEVICE_PROPERTY, 227

HANDLER_FUNC
 utils.h, 378

hCard
 WD_CARD_CLEANUP, 215
 WD_CARD_REGISTER, 216
 WD_DMA, 221

hDevice
 WD_GET_DEVICE_PROPERTY, 227

hDma
 WD_DMA, 222

HEADER_TYPE_ALL
 pci_regs.h, 367

HEADER_TYPE_BRIDGE
 pci_regs.h, 367

HEADER_TYPE_BRIDGE_CARDBUS
 pci_regs.h, 367

HEADER_TYPE_CARDBUS
 pci_regs.h, 367

HEADER_TYPE_NORMAL
 pci_regs.h, 367

HEADER_TYPE_NRML_BRIDGE
 pci_regs.h, 367

HEADER_TYPE_NRML_CARDBUS
 pci_regs.h, 367

hEvent
 WD_EVENT, 225
 WDC_DEVICE, 258

hInterrupt
 WD_INTERRUPT, 230
 WD_ITEMS, 236

hIntThread
 WDC_DEVICE, 259

hIpc
 WD_EVENT, 225
 WD_IPC_PROCESS, 231
 WD_IPC_SCAN_PROCS, 232
 WD_IPC_SEND, 234

hKerBuf
 WD_KERNEL_BUFFER, 239

hKernelPlugIn
 WD_EVENT, 225
 WD_KERNEL_PLUGIN, 240
 WD_KERNEL_PLUGIN_CALL, 241

I

 WD_ITEMS, 236

iConfiguration
 WDU_CONFIGURATION_DESCRIPTOR, 268

id
 WDC_DEVICE, 259

idProduct
 WDU_DEVICE_DESCRIPTOR, 272

idVendor
 WDU_DEVICE_DESCRIPTOR, 272

iInterface
 WDU_INTERFACE_DESCRIPTOR, 280

iManufacturer
 WDU_DEVICE_DESCRIPTOR, 272

INFINITE
 utils.h, 377

Init

CCString, 205
INSTALLATION_TYPE_NOT_DETECT_TEXT
 windrvr.h, 518
Int
 WD_ITEMS, 236
 WDC_DEVICE, 259
INT_HANDLER
 windrvr_int_thread.h, 599
INT_HANDLER_FUNC
 windrvr_int_thread.h, 599
INTERRUPT_CE_INT_ID
 windrvr.h, 562
INTERRUPT_CMD_COPY
 windrvr.h, 562
INTERRUPT_CMD_RETURN_VALUE
 windrvr.h, 562
INTERRUPT_DONT_GET_MSI_MESSAGE
 windrvr.h, 562
INTERRUPT_INTERRUPTED
 windrvr.h, 571
INTERRUPT_LATCHED
 windrvr.h, 562
INTERRUPT_LEVEL_SENSITIVE
 windrvr.h, 562
INTERRUPT_MESSAGE
 windrvr.h, 562
INTERRUPT_MESSAGE_X
 windrvr.h, 562
INTERRUPT_RECEIVED
 windrvr.h, 571
INTERRUPT_STOPPED
 windrvr.h, 571
InterruptDisable
 windrvr_int_thread.h, 599
InterruptEnable
 windrvr_int_thread.h, 600
INVALID_HANDLE_VALUE
 windrvr.h, 519
IO
 WD_ITEMS, 236
IOCTL_WD_CARD_CLEANUP_SETUP
 windrvr.h, 519
IOCTL_WD_CARD_REGISTER
 windrvr.h, 519
IOCTL_WD_CARD_UNREGISTER
 windrvr.h, 519
IOCTL_WD_DEBUG
 windrvr.h, 519
IOCTL_WD_DEBUG_ADD
 windrvr.h, 519
IOCTL_WD_DEBUG_DUMP
 windrvr.h, 519
IOCTL_WD_DMA_LOCK
 windrvr.h, 519
IOCTL_WD_DMA_SYNC_CPU
 windrvr.h, 519
IOCTL_WD_DMA_SYNC_IO
 windrvr.h, 519
IOCTL_WD_DMA_TRANSACTION_EXECUTE
 windrvr.h, 520
IOCTL_WD_DMA_TRANSACTION_INIT
 windrvr.h, 520
IOCTL_WD_DMA_TRANSACTION_RELEASE
 windrvr.h, 520
IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK
 windrvr.h, 520
IOCTL_WD_DMA_UNLOCK
 windrvr.h, 520
IOCTL_WD_EVENT_PULL
 windrvr.h, 520
IOCTL_WD_EVENT_REGISTER
 windrvr.h, 520
IOCTL_WD_EVENT_SEND
 windrvr.h, 520
IOCTL_WD_EVENT_UNREGISTER
 windrvr.h, 520
IOCTL_WD_GET_DEVICE_PROPERTY
 windrvr.h, 520
IOCTL_WD_INT_COUNT
 windrvr.h, 521
IOCTL_WD_INT_DISABLE
 windrvr.h, 521
IOCTL_WD_INT_ENABLE
 windrvr.h, 521
IOCTL_WD_INT_WAIT
 windrvr.h, 521
IOCTL_WD_IPC_REGISTER
 windrvr.h, 521
IOCTL_WD_IPC_SCAN_PROCS
 windrvr.h, 521
IOCTL_WD_IPC_SEND
 windrvr.h, 521
IOCTL_WD_IPC_SHARED_INT_DISABLE
 windrvr.h, 521
IOCTL_WD_IPC_SHARED_INT_ENABLE
 windrvr.h, 521
IOCTL_WD_IPC_UNREGISTER
 windrvr.h, 521
IOCTL_WD_KERNEL_BUF_LOCK
 windrvr.h, 522
IOCTL_WD_KERNEL_BUF_UNLOCK
 windrvr.h, 522
IOCTL_WD_KERNEL_PLUGIN_CALL
 windrvr.h, 522
IOCTL_WD_KERNEL_PLUGIN_CLOSE
 windrvr.h, 522
IOCTL_WD_KERNEL_PLUGIN_OPEN
 windrvr.h, 522
IOCTL_WD_LICENSE
 windrvr.h, 522
IOCTL_WD_MULTI_TRANSFER
 windrvr.h, 522
IOCTL_WD_PCI_CONFIG_DUMP
 windrvr.h, 522
IOCTL_WD_PCI_GET_CARD_INFO
 windrvr.h, 522

IOCTL_WD_PCI_SCAN_CAPS
 windrvr.h, 522

IOCTL_WD_PCI_SCAN_CARDS
 windrvr.h, 523

IOCTL_WD_PCI_SRIOV_DISABLE
 windrvr.h, 523

IOCTL_WD_PCI_SRIOV_ENABLE
 windrvr.h, 523

IOCTL_WD_PCI_SRIOV_GET_NUMVFS
 windrvr.h, 523

IOCTL_WD_SLEEP
 windrvr.h, 523

IOCTL_WD_TRANSFER
 windrvr.h, 523

IOCTL_WD_USAGE
 windrvr.h, 523

IOCTL_WD_VERSION
 windrvr.h, 523

IOCTL_WDU_GET_DEVICE_DATA
 windrvr.h, 523

IOCTL_WDU_HALT_TRANSFER
 windrvr.h, 523

IOCTL_WDU_RESET_DEVICE
 windrvr.h, 524

IOCTL_WDU_RESET_PIPE
 windrvr.h, 524

IOCTL_WDU_SELECTIVE_SUSPEND
 windrvr.h, 524

IOCTL_WDU_SET_INTERFACE
 windrvr.h, 524

IOCTL_WDU_STREAM_CLOSE
 windrvr.h, 524

IOCTL_WDU_STREAM_FLUSH
 windrvr.h, 524

IOCTL_WDU_STREAM_GET_STATUS
 windrvr.h, 524

IOCTL_WDU_STREAM_OPEN
 windrvr.h, 524

IOCTL_WDU_STREAM_START
 windrvr.h, 524

IOCTL_WDU_STREAM_STOP
 windrvr.h, 524

IOCTL_WDU_TRANSFER
 windrvr.h, 525

IOCTL_WDU_WAKEUP
 windrvr.h, 525

ipc
 WD_EVENT, 226

IPC_MSG_RX_HANDLER
 wds_lib.h, 474

iProduct
 WDU_DEVICE_DESCRIPTOR, 272

is_empty
 CCString, 206

IsAllocOK
 CCString, 206

iSerialNumber
 WDU_DEVICE_DESCRIPTOR, 272

Item
 WD_CARD, 214

item
 WD_ITEMS, 237

ITEM_BUS
 windrvr.h, 565

ITEM_INTERRUPT
 windrvr.h, 565

ITEM_IO
 windrvr.h, 565

ITEM_MEMORY
 windrvr.h, 565

ITEM_NONE
 windrvr.h, 565

ITEM_TYPE
 windrvr.h, 564

KDBG
 kpstlib.h, 299

KER_BUF_ALLOC_CACHED
 windrvr.h, 572

KER_BUF_ALLOC_CONTIG
 windrvr.h, 572

KER_BUF_ALLOC_NON_CONTIG
 windrvr.h, 572

KER_BUF_GET_EXISTING_BUF
 windrvr.h, 572

KERNEL_DEBUGGER_OFF
 windrvr.h, 564

KERNEL_DEBUGGER_ON
 windrvr.h, 564

kerPlug
 WDC_DEVICE, 259

KP_FUNC_CALL
 wd_kp.h, 386

KP_FUNC_CLOSE
 wd_kp.h, 386

KP_FUNC_EVENT
 wd_kp.h, 387

KP_FUNC_INT_AT_DPC
 wd_kp.h, 387

KP_FUNC_INT_AT_DPC_MSI
 wd_kp.h, 387

KP_FUNC_INT_AT_IRQL
 wd_kp.h, 387

KP_FUNC_INT_AT_IRQL_MSI
 wd_kp.h, 387

KP_FUNC_INT_DISABLE
 wd_kp.h, 387

KP_FUNC_INT_ENABLE
 wd_kp.h, 387

KP_FUNC_OPEN
 wd_kp.h, 388

KP_INIT, 209
 cDriverName, 210
 dwVerWD, 210
 funcOpen, 210
 funcOpen_32_64, 210

KP_Init

kp_pci.c, 611
wd_kp.h, 388
KP_INTERLOCKED
 kpstdlib.h, 299
kp_interlocked_add
 kpstdlib.h, 299
kp_interlocked_decrement
 kpstdlib.h, 300
kp_interlocked_exchange
 kpstdlib.h, 300
kp_interlocked_increment
 kpstdlib.h, 301
kp_interlocked_init
 kpstdlib.h, 301
kp_interlocked_read
 kpstdlib.h, 301
kp_interlocked_set
 kpstdlib.h, 301
kp_interlocked_uninit
 kpstdlib.h, 302
KP_OPEN_CALL, 211
 funcCall, 211
 funcClose, 211
 funcEvent, 211
 funcIntAtDpc, 211
 funcIntAtDpcMSI, 211
 funcIntAtIrql, 212
 funcIntAtIrqlMSI, 212
 funcIntDisable, 212
 funcIntEnable, 212
kp_pci.c, 610, 618
 DRIVER_VER_STR, 611
 KP_Init, 611
 KP_PCI_Call, 612
 KP_PCI_Close, 612
 KP_PCI_Event, 613
 KP_PCI_IntAtDpc, 613
 KP_PCI_IntAtDpcMSI, 614
 KP_PCI_IntAtIrql, 614
 KP_PCI_IntAtIrqlMSI, 615
 KP_PCI_IntDisable, 616
 KP_PCI_IntEnable, 616
 KP_PCI_Open, 617
 KP_PCI_Open_32_64, 617
 PTR32, 611
 USE_MULTI_TRANSFER, 611
KP_PCI_Call
 kp_pci.c, 612
KP_PCI_Close
 kp_pci.c, 612
KP_PCI_Event
 kp_pci.c, 613
KP_PCI_IntAtDpc
 kp_pci.c, 613
KP_PCI_IntAtDpcMSI
 kp_pci.c, 614
KP_PCI_IntAtIrql
 kp_pci.c, 614
KP_PCI_IntAtIrqlMSI
 kp_pci.c, 615
KP_PCI_IntDisable
 kp_pci.c, 616
KP_PCI_IntEnable
 kp_pci.c, 616
KP_PCI_Open
 kp_pci.c, 617
KP_PCI_Open_32_64
 kp_pci.c, 617
KP_SPINLOCK
 kpstdlib.h, 299
kp_spinlock_init
 kpstdlib.h, 302
kp_spinlock_release
 kpstdlib.h, 302
kp_spinlock_uninit
 kpstdlib.h, 303
kp_spinlock_wait
 kpstdlib.h, 303
kpCall
 WD_INTERRUPT, 230
KPRI
 windrvr.h, 525
kpstdlib.h, 296, 304
 __KERNEL__, 297
COPY_FROM_USER_OR_KERNEL, 297
COPY_TO_USER_OR_KERNEL, 298
FALSE, 298
free, 299
KDBG, 299
KP_INTERLOCKED, 299
kp_interlocked_add, 299
kp_interlocked_decrement, 300
kp_interlocked_exchange, 300
kp_interlocked_increment, 301
kp_interlocked_init, 301
kp_interlocked_read, 301
kp_interlocked_set, 301
kp_interlocked_uninit, 302
KP_SPINLOCK, 299
kp_spinlock_init, 302
kp_spinlock_release, 302
kp_spinlock_uninit, 303
kp_spinlock_wait, 303
malloc, 303
NULL, 298
strcpy, 303
TRUE, 299
KPTR
 windrvr.h, 557
Length
 CCString, 206
LNK_CAPS
 pci_regs.h, 366
LNK_CAPS2
 pci_regs.h, 367
LNK_CTL

pci_regs.h, 366
LNK_CTL2
 pci_regs.h, 367
LNK_STS
 pci_regs.h, 366
LNK_STS2
 pci_regs.h, 367

m_buf_size
 CCString, 209
m_str
 CCString, 209
MakeLower
 CCString, 206
MakeUpper
 CCString, 206
malloc
 kpstldlib.h, 303
matchTables
 WD_EVENT, 226
MAX
 windrvr.h, 525
MAX_DESC
 wdc_lib.h, 410
MAX_NAME
 wdc_lib.h, 410
MAX_NAME_DISPLAY
 wdc_lib.h, 411
MAX_PATH
 utils.h, 377
MaxPower
 WDU_CONFIGURATION_DESCRIPTOR, 268

Mem
 WD_ITEMS, 237
METHOD_BUFFERED
 windrvr.h, 525
METHOD_IN_DIRECT
 windrvr.h, 525
METHOD_NEITHER
 windrvr.h, 525
METHOD_OUT_DIRECT
 windrvr.h, 525
Mid
 CCString, 206
MIN
 windrvr.h, 525

NEXT_CAP_PTR
 pci_regs.h, 366
NULL
 kpstldlib.h, 298

operator char *
 CCString, 206
operator PCSTR
 CCString, 206
operator!=
 CCString, 207
operator+

 cstring.h, 295
operator+=
 CCString, 207
operator=
 CCString, 207
operator==
 CCString, 207
operator[]
 CCString, 208
OS_CAN_NOT_DETECT_TEXT
 windrvr.h, 526
OsEventClose
 utils.h, 378
OsEventCreate
 utils.h, 379
OsEventReset
 utils.h, 379
OsEventSignal
 utils.h, 379
OsEventWait
 utils.h, 379
OsMemoryBarrier
 utils.h, 377
OsMutexClose
 utils.h, 380
OsMutexCreate
 utils.h, 380
OsMutexLock
 utils.h, 380
OsMutexUnlock
 utils.h, 381

pActiveAltSetting
 WDU_INTERFACE, 278
pActiveConfig
 WDU_DEVICE, 269
pActiveInterface
 WDU_DEVICE, 269
PAD_TO_64
 WD_KERNEL_PLUGIN, 240
 windrvr.h, 526
 windrvr_usb.h, 602
PAD_TO_64_PTR_ARR
 windrvr_usb.h, 602
pAddr
 WD_ITEMS, 237
 WDC_ADDR_DESC, 257
pAddrDesc
 WDC_DEVICE, 259
Page
 WD_DMA, 222
pAlternateSettings
 WDU_INTERFACE, 278
pBuf
 WD_GET_DEVICE_PROPERTY, 228
 WDU_GET_DEVICE_DATA, 277
pBuffer
 WD_PCI_CONFIG_DUMP, 246
 WD_TRANSFER, 254

WDU_GET_DESCRIPTOR, 276
WDU_TRANSFER, 291
pcBuffer
 WD_DEBUG_ADD, 219
Pci
 WD_EVENT, 226
PCI_ACCESS_ERROR
 windrvr.h, 565
PCI_ACCESS_OK
 windrvr.h, 565
PCI_ACCESS_RESULT
 windrvr.h, 565
PCI_BAD_BUS
 windrvr.h, 565
PCI_BAD_SLOT
 windrvr.h, 565
PCI_BAR0
 pci_regs.h, 366
PCI_BAR1
 pci_regs.h, 366
PCI_BAR2
 pci_regs.h, 366
PCI_BAR3
 pci_regs.h, 366
PCI_BAR4
 pci_regs.h, 366
PCI_BAR5
 pci_regs.h, 366
PCI_BISTR
 pci_regs.h, 366
PCI_CAP
 pci_regs.h, 366
PCI_CAP_EXP_ENDPOINT_SIZEOF_V1
 pci_regs.h, 319
PCI_CAP_EXP_ENDPOINT_SIZEOF_V2
 pci_regs.h, 319
PCI_CAP_ID_AF
 pci_regs.h, 319
PCI_CAP_ID_AGP
 pci_regs.h, 319
PCI_CAP_ID_AG3
 pci_regs.h, 320
PCI_CAP_ID_CCRC
 pci_regs.h, 320
PCI_CAP_ID_CHSWP
 pci_regs.h, 320
PCI_CAP_ID_DBG
 pci_regs.h, 320
PCI_CAP_ID_EXP
 pci_regs.h, 320
PCI_CAP_ID_HT
 pci_regs.h, 320
PCI_CAP_ID_MSI
 pci_regs.h, 320
PCI_CAP_ID_MSIX
 pci_regs.h, 321
PCI_CAP_ID_PCIX
 pci_regs.h, 321
PCI_CAP_ID_PM
 pci_regs.h, 321
PCI_CAP_ID_SATA
 pci_regs.h, 321
PCI_CAP_ID_SECDEV
 pci_regs.h, 321
PCI_CAP_ID_SHPC
 pci_regs.h, 321
PCI_CAP_ID_SLOTID
 pci_regs.h, 321
PCI_CAP_ID_SSVID
 pci_regs.h, 322
PCI_CAP_ID_VNDR
 pci_regs.h, 322
PCI_CAP_ID_VPD
 pci_regs.h, 322
PCI_CAP_LIST_ID
 pci_regs.h, 322
PCI_CAP_LIST_NEXT
 pci_regs.h, 322
PCI_CCBC
 pci_regs.h, 366
PCI_CCR
 pci_regs.h, 366
PCI_CCSC
 pci_regs.h, 366
PCI_CIS
 pci_regs.h, 366
PCI_CLSR
 pci_regs.h, 366
PCI_COMMAND
 pci_regs.h, 322
PCI_COMMAND_FAST_BACK
 pci_regs.h, 322
PCI_COMMAND_INTX_DISABLE
 pci_regs.h, 323
PCI_COMMAND_INVALIDATE
 pci_regs.h, 323
PCI_COMMAND_IO
 pci_regs.h, 323
PCI_COMMAND_MASTER
 pci_regs.h, 323
PCI_COMMAND_MEMORY
 pci_regs.h, 323
PCI_COMMAND_PARITY
 pci_regs.h, 323
PCI_COMMAND_SERR
 pci_regs.h, 323
PCI_COMMAND_SPECIAL
 pci_regs.h, 324
PCI_COMMAND_VGA_PALETTE
 pci_regs.h, 324
PCI_COMMAND_WAIT
 pci_regs.h, 324
PCI_CONFIG_REGS_OFFSET
 pci_regs.h, 365
PCI_CR
 pci_regs.h, 366

PCI DID
 pci_regs.h, 365

PCI_EROM
 pci_regs.h, 366

PCI_EXP_DEVCAP
 pci_regs.h, 324

PCI_EXP_DEVCAP2
 pci_regs.h, 324

PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP
 pci_regs.h, 324

PCI_EXP_DEVCAP2_ARI
 pci_regs.h, 324

PCI_EXP_DEVCAP2_ATOMIC_COMP32
 pci_regs.h, 325

PCI_EXP_DEVCAP2_ATOMIC_COMP64
 pci_regs.h, 325

PCI_EXP_DEVCAP2_ATOMIC_ROUTE
 pci_regs.h, 325

PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP
 pci_regs.h, 325

PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP
 pci_regs.h, 325

PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP
 pci_regs.h, 325

PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT
 pci_regs.h, 325

PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP
 pci_regs.h, 326

PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP
 pci_regs.h, 326

PCI_EXP_DEVCAP2_LTR
 pci_regs.h, 326

PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES
 pci_regs.h, 326

PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR
 pci_regs.h, 326

PCI_EXP_DEVCAP2_OBFF_MASK
 pci_regs.h, 326

PCI_EXP_DEVCAP2_OBFF_MSG
 pci_regs.h, 326

PCI_EXP_DEVCAP2_OBFF_WAKE
 pci_regs.h, 327

PCI_EXP_DEVCAP2_RANGE_A
 pci_regs.h, 327

PCI_EXP_DEVCAP2_RANGE_B
 pci_regs.h, 327

PCI_EXP_DEVCAP2_RANGE_C
 pci_regs.h, 327

PCI_EXP_DEVCAP2_RANGE_D
 pci_regs.h, 327

PCI_EXP_DEVCAP2_TPH_COMP_SUPP
 pci_regs.h, 327

PCI_EXP_DEVCAP_ATN_BUT
 pci_regs.h, 327

PCI_EXP_DEVCAP_ATN_IND
 pci_regs.h, 328

PCI_EXP_DEVCAP_EXT_TAG
 pci_regs.h, 328

PCI_EXP_DEVCAP_FLR
 pci_regs.h, 328

PCI_EXP_DEVCAP_L0S
 pci_regs.h, 328

PCI_EXP_DEVCAP_L1
 pci_regs.h, 328

PCI_EXP_DEVCAP_L1_SHIFT
 pci_regs.h, 328

PCI_EXP_DEVCAP_PAYLOAD
 pci_regs.h, 328

PCI_EXP_DEVCAP_PHANTOM
 pci_regs.h, 329

PCI_EXP_DEVCAP_PHANTOM_SHIFT
 pci_regs.h, 329

PCI_EXP_DEVCAP_PWD_SCL_SHIFT
 pci_regs.h, 329

PCI_EXP_DEVCAP_PWR_IND
 pci_regs.h, 329

PCI_EXP_DEVCAP_PWR_SCL
 pci_regs.h, 329

PCI_EXP_DEVCAP_PWR_VAL
 pci_regs.h, 329

PCI_EXP_DEVCAP_PWR_VAL_SHIFT
 pci_regs.h, 329

PCI_EXP_DEVCAP_RBER
 pci_regs.h, 330

PCI_EXP_DEVCTL
 pci_regs.h, 330

PCI_EXP_DEVCTL2
 pci_regs.h, 330

PCI_EXP_DEVCTL2_ARI
 pci_regs.h, 330

PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK
 pci_regs.h, 330

PCI_EXP_DEVCTL2_ATOMIC_REQ
 pci_regs.h, 330

PCI_EXP_DEVCTL2_COMP_TIMEOUT
 pci_regs.h, 330

PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE
 pci_regs.h, 331

PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK
 pci_regs.h, 331

PCI_EXP_DEVCTL2_IDO_CMP_EN
 pci_regs.h, 331

PCI_EXP_DEVCTL2_IDO_REQ_EN
 pci_regs.h, 331

PCI_EXP_DEVCTL2_LTR_EN
 pci_regs.h, 331

PCI_EXP_DEVCTL2_OBFF_MSGA_EN
 pci_regs.h, 331

PCI_EXP_DEVCTL2_OBFF_MSGB_EN
 pci_regs.h, 331

PCI_EXP_DEVCTL2_OBFF_WAKE_EN
 pci_regs.h, 332

PCI_EXP_DEVCTL_AUX_PME
 pci_regs.h, 332

PCI_EXP_DEVCTL_BCR_FLR
 pci_regs.h, 332

PCI_EXP_DEVCTL_CERE
 pci_regs.h, 332

PCI_EXP_DEVCTL_EXT_TAG
 pci_regs.h, 332

PCI_EXP_DEVCTL_FERE
 pci_regs.h, 332

PCI_EXP_DEVCTL_NFERE
 pci_regs.h, 332

PCI_EXP_DEVCTL_NOSNOOP_EN
 pci_regs.h, 333

PCI_EXP_DEVCTL_PAYLOAD
 pci_regs.h, 333

PCI_EXP_DEVCTL_PAYLOAD_SHIFT
 pci_regs.h, 333

PCI_EXP_DEVCTL_PHANTOM
 pci_regs.h, 333

PCI_EXP_DEVCTL_READRQ
 pci_regs.h, 333

PCI_EXP_DEVCTL_READRQ_1024B
 pci_regs.h, 333

PCI_EXP_DEVCTL_READRQ_128B
 pci_regs.h, 333

PCI_EXP_DEVCTL_READRQ_256B
 pci_regs.h, 334

PCI_EXP_DEVCTL_READRQ_512B
 pci_regs.h, 334

PCI_EXP_DEVCTL_READRQ_SHIFT
 pci_regs.h, 334

PCI_EXP_DEVCTL_RELAX_EN
 pci_regs.h, 334

PCI_EXP_DEVCTL_URRE
 pci_regs.h, 334

PCI_EXP_DEVSTA
 pci_regs.h, 334

PCI_EXP_DEVSTA2
 pci_regs.h, 334

PCI_EXP_DEVSTA_AUXPD
 pci_regs.h, 335

PCI_EXP_DEVSTA_CED
 pci_regs.h, 335

PCI_EXP_DEVSTA_FED
 pci_regs.h, 335

PCI_EXP_DEVSTA_NFED
 pci_regs.h, 335

PCI_EXP_DEVSTA_TRPND
 pci_regs.h, 335

PCI_EXP_DEVSTA_URD
 pci_regs.h, 335

PCI_EXP_FLAGS
 pci_regs.h, 335

PCI_EXP_FLAGS_IRQ
 pci_regs.h, 336

PCI_EXP_FLAGS_SLOT
 pci_regs.h, 336

PCI_EXP_FLAGS_TYPE
 pci_regs.h, 336

PCI_EXP_FLAGS_TYPE_SHIFT
 pci_regs.h, 336

PCI_EXP_FLAGS_VERS
 pci_regs.h, 336

PCI_EXP_LNKCAP
 pci_regs.h, 336

PCI_EXP_LNKCAP2
 pci_regs.h, 336

PCI_EXP_LNKCAP2_CROSSLINK
 pci_regs.h, 337

PCI_EXP_LNKCAP2_SLS_2_5GB
 pci_regs.h, 337

PCI_EXP_LNKCAP2_SLS_5_0GB
 pci_regs.h, 337

PCI_EXP_LNKCAP2_SLS_8_0GB
 pci_regs.h, 337

PCI_EXP_LNKCAP_ASPPMS
 pci_regs.h, 337

PCI_EXP_LNKCAP_ASPPMS_SHIFT
 pci_regs.h, 337

PCI_EXP_LNKCAP_CLKPM
 pci_regs.h, 337

PCI_EXP_LNKCAP_DLLLARC
 pci_regs.h, 338

PCI_EXP_LNKCAP_L0SEL
 pci_regs.h, 338

PCI_EXP_LNKCAP_L0SEL_SHIFT
 pci_regs.h, 338

PCI_EXP_LNKCAP_L1EL
 pci_regs.h, 338

PCI_EXP_LNKCAP_L1EL_SHIFT
 pci_regs.h, 338

PCI_EXP_LNKCAP_LBNC
 pci_regs.h, 338

PCI_EXP_LNKCAP_MLW
 pci_regs.h, 338

PCI_EXP_LNKCAP_MLW_SHIFT
 pci_regs.h, 339

PCI_EXP_LNKCAP_PN
 pci_regs.h, 339

PCI_EXP_LNKCAP_SDERC
 pci_regs.h, 339

PCI_EXP_LNKCAP_SLS
 pci_regs.h, 339

PCI_EXP_LNKCAP_SLS_2_5GB
 pci_regs.h, 339

PCI_EXP_LNKCAP_SLS_5_0GB
 pci_regs.h, 339

PCI_EXP_LNKCTL
 pci_regs.h, 339

PCI_EXP_LNKCTL2
 pci_regs.h, 340

PCI_EXP_LNKCTL2_COMP_SOS
 pci_regs.h, 340

PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL
 pci_regs.h, 340

PCI_EXP_LNKCTL2_ENTER_COMP
 pci_regs.h, 340

PCI_EXP_LNKCTL2_ENTER_MOD_COMP
 pci_regs.h, 340

PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS
 pci_regs.h, 340
PCI_EXP_LNKCTL2_LNK_SPEED_2_5
 pci_regs.h, 340
PCI_EXP_LNKCTL2_LNK_SPEED_5_0
 pci_regs.h, 341
PCI_EXP_LNKCTL2_LNK_SPEED_8_0
 pci_regs.h, 341
PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH
 pci_regs.h, 341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK
 pci_regs.h, 341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT
 pci_regs.h, 341
PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL
 pci_regs.h, 341
PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK
 pci_regs.h, 341
PCI_EXP_LNKCTL_ASPM_L0S
 pci_regs.h, 342
PCI_EXP_LNKCTL_ASPM_L1
 pci_regs.h, 342
PCI_EXP_LNKCTL_ASPMC
 pci_regs.h, 342
PCI_EXP_LNKCTL_CCC
 pci_regs.h, 342
PCI_EXP_LNKCTL_CLKREQ_EN
 pci_regs.h, 342
PCI_EXP_LNKCTL_ES
 pci_regs.h, 342
PCI_EXP_LNKCTL_HAWD
 pci_regs.h, 342
PCI_EXP_LNKCTL_LABIE
 pci_regs.h, 343
PCI_EXP_LNKCTL_LBMIE
 pci_regs.h, 343
PCI_EXP_LNKCTL_LD
 pci_regs.h, 343
PCI_EXP_LNKCTL_RCB
 pci_regs.h, 343
PCI_EXP_LNKCTL_RL
 pci_regs.h, 343
PCI_EXP_LNKSTA
 pci_regs.h, 343
PCI_EXP_LNKSTA2
 pci_regs.h, 343
PCI_EXP_LNKSTA2_CDL
 pci_regs.h, 344
PCI_EXP_LNKSTA2_EQUALIZ_COMP
 pci_regs.h, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH1
 pci_regs.h, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH2
 pci_regs.h, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH3
 pci_regs.h, 344
PCI_EXP_LNKSTA2_LINE_EQ_REQ
 pci_regs.h, 344
PCI_EXP_LNKSTA_CLS
 pci_regs.h, 344
PCI_EXP_LNKSTA_CLS_2_5GB
 pci_regs.h, 345
PCI_EXP_LNKSTA_CLS_5_0GB
 pci_regs.h, 345
PCI_EXP_LNKSTA_CLS_8_0GB
 pci_regs.h, 345
PCI_EXP_LNKSTA_DLLA
 pci_regs.h, 345
PCI_EXP_LNKSTA_LABS
 pci_regs.h, 345
PCI_EXP_LNKSTA_LBMS
 pci_regs.h, 345
PCI_EXP_LNKSTA_LT
 pci_regs.h, 345
PCI_EXP_LNKSTA_NLW
 pci_regs.h, 346
PCI_EXP_LNKSTA_NLW_SHIFT
 pci_regs.h, 346
PCI_EXP_LNKSTA_NLW_X1
 pci_regs.h, 346
PCI_EXP_LNKSTA_NLW_X2
 pci_regs.h, 346
PCI_EXP_LNKSTA_NLW_X4
 pci_regs.h, 346
PCI_EXP_LNKSTA_NLW_X8
 pci_regs.h, 346
PCI_EXP_LNKSTA_SLC
 pci_regs.h, 346
PCI_EXP_RTCAP
 pci_regs.h, 347
PCI_EXP_RTCAP_CRSVIS
 pci_regs.h, 347
PCI_EXP_RTCTL
 pci_regs.h, 347
PCI_EXP_RTCTL_CRSSVE
 pci_regs.h, 347
PCI_EXP_RTCTL_PMEIE
 pci_regs.h, 347
PCI_EXP_RTCTL_SECEE
 pci_regs.h, 347
PCI_EXP_RTCTL_SEFEE
 pci_regs.h, 347
PCI_EXP_RTCTL_SENFEE
 pci_regs.h, 348
PCI_EXP_RTSTA
 pci_regs.h, 348
PCI_EXP_RTSTA_PENDING
 pci_regs.h, 348
PCI_EXP_RTSTA_PME
 pci_regs.h, 348
PCI_EXP_SLTCAP
 pci_regs.h, 348
PCI_EXP_SLTCAP2
 pci_regs.h, 348
PCI_EXP_SLTCAP_AB
 pci_regs.h, 348

PCI_EXP_SLTCAP_AIC_SHIFT
 pci_regs.h, 349
PCI_EXP_SLTCAP_AIP
 pci_regs.h, 349
PCI_EXP_SLTCAP_EIP
 pci_regs.h, 349
PCI_EXP_SLTCAP_HPC
 pci_regs.h, 349
PCI_EXP_SLTCAP_HPS
 pci_regs.h, 349
PCI_EXP_SLTCAP_MRLSP
 pci_regs.h, 349
PCI_EXP_SLTCAP_NCCS
 pci_regs.h, 349
PCI_EXP_SLTCAP_PCP
 pci_regs.h, 350
PCI_EXP_SLTCAP_PIP
 pci_regs.h, 350
PCI_EXP_SLTCAP_PSN
 pci_regs.h, 350
PCI_EXP_SLTCAP SPLS
 pci_regs.h, 350
PCI_EXP_SLTCAP SPLS SHIFT
 pci_regs.h, 350
PCI_EXP_SLTCAP SPLV
 pci_regs.h, 350
PCI_EXP_SLTCAP SPLV SHIFT
 pci_regs.h, 350
PCI_EXP_SLTCTL
 pci_regs.h, 351
PCI_EXP_SLTCTL2
 pci_regs.h, 351
PCI_EXP_SLTCTL_ABPE
 pci_regs.h, 351
PCI_EXP_SLTCTL_AIC
 pci_regs.h, 351
PCI_EXP_SLTCTL_ATTN_IND_BLINK
 pci_regs.h, 351
PCI_EXP_SLTCTL_ATTN_IND_OFF
 pci_regs.h, 351
PCI_EXP_SLTCTL_ATTN_IND_ON
 pci_regs.h, 351
PCI_EXP_SLTCTL_CCIE
 pci_regs.h, 352
PCI_EXP_SLTCTL_DLLSCE
 pci_regs.h, 352
PCI_EXP_SLTCTL_EIC
 pci_regs.h, 352
PCI_EXP_SLTCTL_HPIE
 pci_regs.h, 352
PCI_EXP_SLTCTL_MRLSCE
 pci_regs.h, 352
PCI_EXP_SLTCTL_PCC
 pci_regs.h, 352
PCI_EXP_SLTCTL_PDCE
 pci_regs.h, 352
PCI_EXP_SLTCTL_PFDE
 pci_regs.h, 353

PCI_EXP_SLTCTL_PIC
 pci_regs.h, 353
PCI_EXP_SLTCTL_PIC_SHIFT
 pci_regs.h, 353
PCI_EXP_SLTCTL_PWR_IND_BLINK
 pci_regs.h, 353
PCI_EXP_SLTCTL_PWR_IND_OFF
 pci_regs.h, 353
PCI_EXP_SLTCTL_PWR_IND_ON
 pci_regs.h, 353
PCI_EXP_SLTCTL_PWR_OFF
 pci_regs.h, 353
PCI_EXP_SLTCTL_PWR_ON
 pci_regs.h, 354
PCI_EXP_SLTSTA
 pci_regs.h, 354
PCI_EXP_SLTSTA2
 pci_regs.h, 354
PCI_EXP_SLTSTA_ABP
 pci_regs.h, 354
PCI_EXP_SLTSTA_CC
 pci_regs.h, 354
PCI_EXP_SLTSTA_DLLSC
 pci_regs.h, 354
PCI_EXP_SLTSTA_EIS
 pci_regs.h, 354
PCI_EXP_SLTSTA_MRLSC
 pci_regs.h, 355
PCI_EXP_SLTSTA_MRLSS
 pci_regs.h, 355
PCI_EXP_SLTSTA_PDC
 pci_regs.h, 355
PCI_EXP_SLTSTA_PDS
 pci_regs.h, 355
PCI_EXP_SLTSTA_PFD
 pci_regs.h, 355
PCI_EXP_TYPE_DOWNSTREAM
 pci_regs.h, 355
PCI_EXP_TYPE_ENDPOINT
 pci_regs.h, 355
PCI_EXP_TYPE_LEG_END
 pci_regs.h, 356
PCI_EXP_TYPE_PCI_BRIDGE
 pci_regs.h, 356
PCI_EXP_TYPE_PCIE_BRIDGE
 pci_regs.h, 356
PCI_EXP_TYPE_RC_EC
 pci_regs.h, 356
PCI_EXP_TYPE_RC_END
 pci_regs.h, 356
PCI_EXP_TYPE_ROOT_PORT
 pci_regs.h, 356
PCI_EXP_TYPE_UPSTREAM
 pci_regs.h, 356
PCI_EXT_CAP_ID
 pci_regs.h, 357
PCI_EXT_CAP_IDACS
 pci_regs.h, 357

PCI_EXT_CAP_ID_AMD_XXX
 pci_regs.h, 357

PCI_EXT_CAP_ID_ARI
 pci_regs.h, 357

PCI_EXT_CAP_ID_ATS
 pci_regs.h, 357

PCI_EXT_CAP_ID_CAC
 pci_regs.h, 357

PCI_EXT_CAP_ID_DPA
 pci_regs.h, 357

PCI_EXT_CAP_ID_DPC
 pci_regs.h, 358

PCI_EXT_CAP_ID_DSN
 pci_regs.h, 358

PCI_EXT_CAP_ID_ERR
 pci_regs.h, 358

PCI_EXT_CAP_ID_FRSQ
 pci_regs.h, 358

PCI_EXT_CAP_ID_L1PMS
 pci_regs.h, 358

PCI_EXT_CAP_ID_LNR
 pci_regs.h, 358

PCI_EXT_CAP_ID_LTR
 pci_regs.h, 358

PCI_EXT_CAP_ID_MCAST
 pci_regs.h, 359

PCI_EXT_CAP_ID_MFVC
 pci_regs.h, 359

PCI_EXT_CAP_ID_MPHY
 pci_regs.h, 359

PCI_EXT_CAP_ID_MRIOV
 pci_regs.h, 359

PCI_EXT_CAP_ID_PASID
 pci_regs.h, 359

PCI_EXT_CAP_ID_PMUX
 pci_regs.h, 359

PCI_EXT_CAP_ID_PRI
 pci_regs.h, 359

PCI_EXT_CAP_ID_PTM
 pci_regs.h, 360

PCI_EXT_CAP_ID_PWR
 pci_regs.h, 360

PCI_EXT_CAP_ID_RCEC
 pci_regs.h, 360

PCI_EXT_CAP_ID_RCILC
 pci_regs.h, 360

PCI_EXT_CAP_ID_RCLD
 pci_regs.h, 360

PCI_EXT_CAP_ID_RCRB
 pci_regs.h, 360

PCI_EXT_CAP_ID_REBAR
 pci_regs.h, 360

PCI_EXT_CAP_ID_RTR
 pci_regs.h, 361

PCI_EXT_CAP_ID_SECPCI
 pci_regs.h, 361

PCI_EXT_CAP_ID_SRIOV
 pci_regs.h, 361

PCI_EXT_CAP_ID_TPH
 pci_regs.h, 361

PCI_EXT_CAP_ID_VC
 pci_regs.h, 361

PCI_EXT_CAP_ID_VC9
 pci_regs.h, 361

PCI_EXT_CAP_ID_VNDR
 pci_regs.h, 361

PCI_EXT_CAP_NEXT
 pci_regs.h, 362

PCI_EXT_CAP_VER
 pci_regs.h, 362

PCI_HDR
 pci_regs.h, 366

PCI_HEADER_TYPE
 pci_regs.h, 362

PCI_HEADER_TYPE_BRIDGE
 pci_regs.h, 362

PCI_HEADER_TYPE_CARDBUS
 pci_regs.h, 362

PCI_HEADER_TYPE_NORMAL
 pci_regs.h, 362

PCI_ILR
 pci_regs.h, 366

PCI_IPR
 pci_regs.h, 366

PCI_LTR
 pci_regs.h, 366

PCI_MGR
 pci_regs.h, 366

PCI_MLR
 pci_regs.h, 366

pci_regs.h, 306, 367

 AD_PCI_BAR, 365

 AD_PCI_BAR0, 365

 AD_PCI_BAR1, 365

 AD_PCI_BAR2, 365

 AD_PCI_BAR3, 365

 AD_PCI_BAR4, 365

 AD_PCI_BAR5, 365

 AD_PCI_BARS, 365

 CAP_REG, 366

 DEV_CAPS, 366

 DEV_CAPS2, 367

 DEV_CTL, 366

 DEV_CTL2, 367

 DEV_STS, 366

 DEV_STS2, 367

 GET_CAPABILITY_STR, 318

 GET_EXTENDED_CAPABILITY_STR, 319

 HEADER_TYPE_ALL, 367

 HEADER_TYPE_BRIDGE, 367

 HEADER_TYPE_BRIDGE_CARDBUS, 367

 HEADER_TYPE_CARDBUS, 367

 HEADER_TYPE_NORMAL, 367

 HEADER_TYPE_NRML_BRIDGE, 367

 HEADER_TYPE_NRML_CARDBUS, 367

 LNK_CAPS, 366

LNK_CAPS2, 367
LNK_CTL, 366
LNK_CTL2, 367
LNK_STS, 366
LNK_STS2, 367
NEXT_CAP_PTR, 366
PCI_BAR0, 366
PCI_BAR1, 366
PCI_BAR2, 366
PCI_BAR3, 366
PCI_BAR4, 366
PCI_BAR5, 366
PCI_BISTR, 366
PCI_CAP, 366
PCI_CAP_EXP_ENDPOINT_SIZEOF_V1, 319
PCI_CAP_EXP_ENDPOINT_SIZEOF_V2, 319
PCI_CAP_ID_AF, 319
PCI_CAP_ID_AGP, 319
PCI_CAP_ID_AGPN, 320
PCI_CAP_ID_CCRC, 320
PCI_CAP_ID_CHSWP, 320
PCI_CAP_ID_DBG, 320
PCI_CAP_ID_EXP, 320
PCI_CAP_ID_HT, 320
PCI_CAP_ID_MSI, 320
PCI_CAP_ID_MSIX, 321
PCI_CAP_ID_PCIX, 321
PCI_CAP_ID_PM, 321
PCI_CAP_ID_SATA, 321
PCI_CAP_ID_SECDEV, 321
PCI_CAP_ID_SHPC, 321
PCI_CAP_ID_SLOTID, 321
PCI_CAP_ID_SSVID, 322
PCI_CAP_ID_VNDR, 322
PCI_CAP_ID_VPD, 322
PCI_CAP_LIST_ID, 322
PCI_CAP_LIST_NEXT, 322
PCI_CCBC, 366
PCI_CCR, 366
PCI_CCCSC, 366
PCI_CIS, 366
PCI_CLSR, 366
PCI_COMMAND, 322
PCI_COMMAND_FAST_BACK, 322
PCI_COMMAND_INTX_DISABLE, 323
PCI_COMMAND_INVALIDATE, 323
PCI_COMMAND_IO, 323
PCI_COMMAND_MASTER, 323
PCI_COMMAND_MEMORY, 323
PCI_COMMAND_PARITY, 323
PCI_COMMAND_SERR, 323
PCI_COMMAND_SPECIAL, 324
PCI_COMMAND_VGA_PALETTE, 324
PCI_COMMAND_WAIT, 324
PCI_CONFIG_REGS_OFFSET, 365
PCI_CR, 366
PCI_DID, 365
PCI_EROM, 366
PCI_EXP_DEVCAP, 324
PCI_EXP_DEVCAP2, 324
PCI_EXP_DEVCAP2_128_CAS_COMP_SUPP,
 324
PCI_EXP_DEVCAP2_ARI, 324
PCI_EXP_DEVCAP2_ATOMIC_COMP32, 325
PCI_EXP_DEVCAP2_ATOMIC_COMP64, 325
PCI_EXP_DEVCAP2_ATOMIC_ROUTE, 325
PCI_EXP_DEVCAP2_COMP_TO_DIS_SUPP,
 325
PCI_EXP_DEVCAP2_COMP_TO_RANGES_SUPP,
 325
PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP,
 325
PCI_EXP_DEVCAP2_EE_TLP_PREFIX_SUPP_SHIFT,
 325
PCI_EXP_DEVCAP2_EXT_FMT_FIELD_SUPP,
 326
PCI_EXP_DEVCAP2_EXT_TPH_COMP_SUPP,
 326
PCI_EXP_DEVCAP2_LTR, 326
PCI_EXP_DEVCAP2_MAX_EE_TLP_PREFIXES,
 326
PCI_EXP_DEVCAP2_NO_RO_ENABLED_PR,
 326
PCI_EXP_DEVCAP2_OBFF_MASK, 326
PCI_EXP_DEVCAP2_OBFF_MSG, 326
PCI_EXP_DEVCAP2_OBFF_WAKE, 327
PCI_EXP_DEVCAP2_RANGE_A, 327
PCI_EXP_DEVCAP2_RANGE_B, 327
PCI_EXP_DEVCAP2_RANGE_C, 327
PCI_EXP_DEVCAP2_RANGE_D, 327
PCI_EXP_DEVCAP2_TPH_COMP_SUPP, 327
PCI_EXP_DEVCAP_ATN_BUT, 327
PCI_EXP_DEVCAP_ATN_IND, 328
PCI_EXP_DEVCAP_EXT_TAG, 328
PCI_EXP_DEVCAP_FLR, 328
PCI_EXP_DEVCAP_LOS, 328
PCI_EXP_DEVCAP_L1, 328
PCI_EXP_DEVCAP_L1_SHIFT, 328
PCI_EXP_DEVCAP_PAYLOAD, 328
PCI_EXP_DEVCAP_PHANTOM, 329
PCI_EXP_DEVCAP_PHANTOM_SHIFT, 329
PCI_EXP_DEVCAP_PWD_SCL_SHIFT, 329
PCI_EXP_DEVCAP_PWR_IND, 329
PCI_EXP_DEVCAP_PWR_SCL, 329
PCI_EXP_DEVCAP_PWR_VAL, 329
PCI_EXP_DEVCAP_PWR_VAL_SHIFT, 329
PCI_EXP_DEVCAP_RBER, 330
PCI_EXP_DEVCTL, 330
PCI_EXP_DEVCTL2, 330
PCI_EXP_DEVCTL2_ARI, 330
PCI_EXP_DEVCTL2_ATOMIC_EGRESS_BLOCK,
 330
PCI_EXP_DEVCTL2_ATOMIC_REQ, 330
PCI_EXP_DEVCTL2_COMP_TIMEOUT, 330
PCI_EXP_DEVCTL2_COMP_TIMEOUT_DISABLE,
 331

PCI_EXP_DEVCTL2_EE_TLP_PREFIX_BLOCK,
 331
PCI_EXP_DEVCTL2_IDO_CMP_EN, 331
PCI_EXP_DEVCTL2_IDO_REQ_EN, 331
PCI_EXP_DEVCTL2_LTR_EN, 331
PCI_EXP_DEVCTL2_OBFF_MSGA_EN, 331
PCI_EXP_DEVCTL2_OBFF_MSGB_EN, 331
PCI_EXP_DEVCTL2_OBFF_WAKE_EN, 332
PCI_EXP_DEVCTL_AUX_PME, 332
PCI_EXP_DEVCTL_BCR_FLR, 332
PCI_EXP_DEVCTL_CERE, 332
PCI_EXP_DEVCTL_EXT_TAG, 332
PCI_EXP_DEVCTL_FERE, 332
PCI_EXP_DEVCTL_NFERE, 332
PCI_EXP_DEVCTL_NOSNOOP_EN, 333
PCI_EXP_DEVCTL_PAYLOAD, 333
PCI_EXP_DEVCTL_PAYLOAD_SHIFT, 333
PCI_EXP_DEVCTL_PHANTOM, 333
PCI_EXP_DEVCTL_READRQ, 333
PCI_EXP_DEVCTL_READRQ_1024B, 333
PCI_EXP_DEVCTL_READRQ_128B, 333
PCI_EXP_DEVCTL_READRQ_256B, 334
PCI_EXP_DEVCTL_READRQ_512B, 334
PCI_EXP_DEVCTL_READRQ_SHIFT, 334
PCI_EXP_DEVCTL_RELAX_EN, 334
PCI_EXP_DEVCTL_URRE, 334
PCI_EXP_DEVSTA, 334
PCI_EXP_DEVSTA2, 334
PCI_EXP_DEVSTA_AUXPD, 335
PCI_EXP_DEVSTA_CED, 335
PCI_EXP_DEVSTA_FED, 335
PCI_EXP_DEVSTA_NFED, 335
PCI_EXP_DEVSTA_TRPND, 335
PCI_EXP_DEVSTA_URD, 335
PCI_EXP_FLAGS, 335
PCI_EXP_FLAGS_IRQ, 336
PCI_EXP_FLAGS_SLOT, 336
PCI_EXP_FLAGS_TYPE, 336
PCI_EXP_FLAGS_TYPE_SHIFT, 336
PCI_EXP_FLAGS_VERS, 336
PCI_EXP_LNKCAP, 336
PCI_EXP_LNKCAP2, 336
PCI_EXP_LNKCAP2_CROSSLINK, 337
PCI_EXP_LNKCAP2_SLS_2_5GB, 337
PCI_EXP_LNKCAP2_SLS_5_0GB, 337
PCI_EXP_LNKCAP2_SLS_8_0GB, 337
PCI_EXP_LNKCAP_ASPMs, 337
PCI_EXP_LNKCAP_ASPMs_SHIFT, 337
PCI_EXP_LNKCAP_CLKPM, 337
PCI_EXP_LNKCAP_DLLARc, 338
PCI_EXP_LNKCAP_L0SEL, 338
PCI_EXP_LNKCAP_L0SEL_SHIFT, 338
PCI_EXP_LNKCAP_L1EL, 338
PCI_EXP_LNKCAP_L1EL_SHIFT, 338
PCI_EXP_LNKCAP_LBNC, 338
PCI_EXP_LNKCAP_MLW, 338
PCI_EXP_LNKCAP_MLW_SHIFT, 339
PCI_EXP_LNKCAP_PN, 339
PCI_EXP_LNKCAP_SDERC, 339
PCI_EXP_LNKCAP_SLS, 339
PCI_EXP_LNKCAP_SLS_2_5GB, 339
PCI_EXP_LNKCAP_SLS_5_0GB, 339
PCI_EXP_LNKCTL, 339
PCI_EXP_LNKCTL2, 340
PCI_EXP_LNKCTL2_COMP_SOS, 340
PCI_EXP_LNKCTL2_DEEMPH_LVL_POLL, 340
PCI_EXP_LNKCTL2_ENTER_COMP, 340
PCI_EXP_LNKCTL2_ENTER_MOD_COMP, 340
PCI_EXP_LNKCTL2_HW_AUTO_SPEED_DIS,
 340
PCI_EXP_LNKCTL2_LNK_SPEED_2_5, 340
PCI_EXP_LNKCTL2_LNK_SPEED_5_0, 341
PCI_EXP_LNKCTL2_LNK_SPEED_8_0, 341
PCI_EXP_LNKCTL2_SELECTABLE_DEEMPH,
 341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK,
 341
PCI_EXP_LNKCTL2_TRANS_MARGIN_MASK_SHIFT,
 341
PCI_EXP_LNKCTL2_TRANS_PRESENT_POLL,
 341
PCI_EXP_LNKCTL2_TRGT_LNK_SPEED_MASK,
 341
PCI_EXP_LNKCTL_ASPM_L0S, 342
PCI_EXP_LNKCTL_ASPM_L1, 342
PCI_EXP_LNKCTL_ASPMC, 342
PCI_EXP_LNKCTL_CCC, 342
PCI_EXP_LNKCTL_CLKREQ_EN, 342
PCI_EXP_LNKCTL_ES, 342
PCI_EXP_LNKCTL_HAWD, 342
PCI_EXP_LNKCTL_LABIE, 343
PCI_EXP_LNKCTL_LBMIE, 343
PCI_EXP_LNKCTL_LD, 343
PCI_EXP_LNKCTL_RCB, 343
PCI_EXP_LNKCTL_RL, 343
PCI_EXP_LNKSTA, 343
PCI_EXP_LNKSTA2, 343
PCI_EXP_LNKSTA2_CDL, 344
PCI_EXP_LNKSTA2_EQUALIZ_COMP, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH1, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH2, 344
PCI_EXP_LNKSTA2_EQUALIZ_PH3, 344
PCI_EXP_LNKSTA2_LINE_EQ_REQ, 344
PCI_EXP_LNKSTA_CLS, 344
PCI_EXP_LNKSTA_CLS_2_5GB, 345
PCI_EXP_LNKSTA_CLS_5_0GB, 345
PCI_EXP_LNKSTA_CLS_8_0GB, 345
PCI_EXP_LNKSTA_DLLA, 345
PCI_EXP_LNKSTA_LABS, 345
PCI_EXP_LNKSTA_LBMS, 345
PCI_EXP_LNKSTA_LT, 345
PCI_EXP_LNKSTA_NLW, 346
PCI_EXP_LNKSTA_NLW_SHIFT, 346
PCI_EXP_LNKSTA_NLW_X1, 346
PCI_EXP_LNKSTA_NLW_X2, 346
PCI_EXP_LNKSTA_NLW_X4, 346

PCI_EXP_LNKSTA_NLW_X8, 346
PCI_EXP_LNKSTA_SLC, 346
PCI_EXP_RTCAP, 347
PCI_EXP_RTCAP_CRSVIS, 347
PCI_EXP_RTCTL, 347
PCI_EXP_RTCTL_CRSSVE, 347
PCI_EXP_RTCTL_PMEIE, 347
PCI_EXP_RTCTL_SECEE, 347
PCI_EXP_RTCTL_SEFEE, 347
PCI_EXP_RTCTL_SENFEE, 348
PCI_EXP_RTSTA, 348
PCI_EXP_RTSTA_PENDING, 348
PCI_EXP_RTSTA_PME, 348
PCI_EXP_SLTCAP, 348
PCI_EXP_SLTCAP2, 348
PCI_EXP_SLTCAP_ABP, 348
PCI_EXP_SLTCAP_AIC_SHIFT, 349
PCI_EXP_SLTCAP_AIP, 349
PCI_EXP_SLTCAP_EIP, 349
PCI_EXP_SLTCAP_HPC, 349
PCI_EXP_SLTCAP_HPS, 349
PCI_EXP_SLTCAP_MRLSP, 349
PCI_EXP_SLTCAP_NCCS, 349
PCI_EXP_SLTCAP_PCP, 350
PCI_EXP_SLTCAP_PIP, 350
PCI_EXP_SLTCAP_PSN, 350
PCI_EXP_SLTCAP SPLS, 350
PCI_EXP_SLTCAP SPLS_SHIFT, 350
PCI_EXP_SLTCAP SPLV, 350
PCI_EXP_SLTCAP SPLV_SHIFT, 350
PCI_EXP_SLTCTL, 351
PCI_EXP_SLTCTL2, 351
PCI_EXP_SLTCTL_ABPE, 351
PCI_EXP_SLTCTL_AIC, 351
PCI_EXP_SLTCTL_ATTN_IND_BLINK, 351
PCI_EXP_SLTCTL_ATTN_IND_OFF, 351
PCI_EXP_SLTCTL_ATTN_IND_ON, 351
PCI_EXP_SLTCTL_CCIE, 352
PCI_EXP_SLTCTL_DLLSCE, 352
PCI_EXP_SLTCTL_EIC, 352
PCI_EXP_SLTCTL_HPIE, 352
PCI_EXP_SLTCTL_MRLSCE, 352
PCI_EXP_SLTCTL_PCC, 352
PCI_EXP_SLTCTL_PDCE, 352
PCI_EXP_SLTCTL_PFDE, 353
PCI_EXP_SLTCTL_PIC, 353
PCI_EXP_SLTCTL_PIC_SHIFT, 353
PCI_EXP_SLTCTL_PWR_IND_BLINK, 353
PCI_EXP_SLTCTL_PWR_IND_OFF, 353
PCI_EXP_SLTCTL_PWR_IND_ON, 353
PCI_EXP_SLTCTL_PWR_OFF, 353
PCI_EXP_SLTCTL_PWR_ON, 354
PCI_EXP_SLTSTA, 354
PCI_EXP_SLTSTA2, 354
PCI_EXP_SLTSTA_ABP, 354
PCI_EXP_SLTSTA_CC, 354
PCI_EXP_SLTSTA_DLLSC, 354
PCI_EXP_SLTSTA_EIS, 354
PCI_EXP_SLTSTA_MRLSC, 355
PCI_EXP_SLTSTA_MRLSS, 355
PCI_EXP_SLTSTA_PDC, 355
PCI_EXP_SLTSTA_PDS, 355
PCI_EXP_SLTSTA_PFD, 355
PCI_EXP_TYPE_DOWNSTREAM, 355
PCI_EXP_TYPE_ENDPOINT, 355
PCI_EXP_TYPE_LEG_END, 356
PCI_EXP_TYPE_PCI_BRIDGE, 356
PCI_EXP_TYPE_PCIE_BRIDGE, 356
PCI_EXP_TYPE_RC_EC, 356
PCI_EXP_TYPE_RC_END, 356
PCI_EXP_TYPE_ROOT_PORT, 356
PCI_EXP_TYPE_UPSTREAM, 356
PCI_EXT_CAP_ID, 357
PCI_EXT_CAP_ID_ACSC, 357
PCI_EXT_CAP_ID_AMD_XXX, 357
PCI_EXT_CAP_ID_ARI, 357
PCI_EXT_CAP_ID_ATS, 357
PCI_EXT_CAP_ID_CAC, 357
PCI_EXT_CAP_ID_DPA, 357
PCI_EXT_CAP_ID_DPC, 358
PCI_EXT_CAP_ID_DSM, 358
PCI_EXT_CAP_ID_ERR, 358
PCI_EXT_CAP_ID_FRSQ, 358
PCI_EXT_CAP_ID_L1PMS, 358
PCI_EXT_CAP_ID_LNR, 358
PCI_EXT_CAP_ID_LTR, 358
PCI_EXT_CAP_ID_MCAST, 359
PCI_EXT_CAP_ID_MFVC, 359
PCI_EXT_CAP_ID_MPHY, 359
PCI_EXT_CAP_ID_MRIOV, 359
PCI_EXT_CAP_ID_PASID, 359
PCI_EXT_CAP_ID_PMUX, 359
PCI_EXT_CAP_ID_PRI, 359
PCI_EXT_CAP_ID_PT, 360
PCI_EXT_CAP_ID_PWR, 360
PCI_EXT_CAP_ID_RCEC, 360
PCI_EXT_CAP_ID_RCILC, 360
PCI_EXT_CAP_ID_RCLD, 360
PCI_EXT_CAP_ID_RCRB, 360
PCI_EXT_CAP_ID_REBAR, 360
PCI_EXT_CAP_ID_RTR, 361
PCI_EXT_CAP_ID_SECPCI, 361
PCI_EXT_CAP_ID_SRIOV, 361
PCI_EXT_CAP_ID_TPH, 361
PCI_EXT_CAP_ID_VC, 361
PCI_EXT_CAP_ID_VC9, 361
PCI_EXT_CAP_ID_VNDR, 361
PCI_EXT_CAP_NEXT, 362
PCI_EXT_CAP_VER, 362
PCI_HDR, 366
PCI_HEADER_TYPE, 362
PCI_HEADER_TYPE_BRIDGE, 362
PCI_HEADER_TYPE_CARDBUS, 362
PCI_HEADER_TYPE_NORMAL, 362
PCI_ILR, 366
PCI_IPR, 366

PCI_LTR, 366
PCI_MGR, 366
PCI_MLR, 366
PCI_REV, 366
PCI_SDID, 366
PCI_SR, 366
PCI_SR_CAP_LIST_BIT, 362
PCI_STATUS, 363
PCI_STATUS_66MHZ, 363
PCI_STATUS_CAP_LIST, 363
PCI_STATUS_DETECTED_PARITY, 363
PCI_STATUS_DEVSEL_FAST, 363
PCI_STATUS_DEVSEL_MASK, 363
PCI_STATUS_DEVSEL_MEDIUM, 363
PCI_STATUS_DEVSEL_SHIFT, 364
PCI_STATUS_DEVSEL_SLOW, 364
PCI_STATUS_FAST_BACK, 364
PCI_STATUS_INTERRUPT, 364
PCI_STATUS_PARITY, 364
PCI_STATUS_REC_MASTER_ABORT, 364
PCI_STATUS_REC_TARGET_ABORT, 364
PCI_STATUS_SIG_SYSTEM_ERROR, 364
PCI_STATUS_SIG_TARGET_ABORT, 365
PCI_STATUS_UDF, 365
PCI_SVID, 366
PCI_VID, 365
PCIE_CAP_ID, 366
PCIE_CONFIG_REGS_OFFSET, 366
ROOT_CAPS, 366
ROOT_CTL, 367
ROOT_STS, 367
SLOT_CAPS, 366
SLOT_CAPS2, 367
SLOT_CTL, 366
SLOT_CTL2, 367
SLOT_STS, 366
SLOT_STS2, 367
WDC_PCI_HEADER_TYPE, 367
PCI_REV
 pci_regs.h, 366
PCI_SDID
 pci_regs.h, 366
PCI_SR
 pci_regs.h, 366
PCI_SR_CAP_LIST_BIT
 pci_regs.h, 362
PCI_STATUS
 pci_regs.h, 363
PCI_STATUS_66MHZ
 pci_regs.h, 363
PCI_STATUS_CAP_LIST
 pci_regs.h, 363
PCI_STATUS_DETECTED_PARITY
 pci_regs.h, 363
PCI_STATUS_DEVSEL_FAST
 pci_regs.h, 363
PCI_STATUS_DEVSEL_MASK
 pci_regs.h, 363
PCI_STATUS_DEVSEL_MEDIUM
 pci_regs.h, 363
PCI_STATUS_DEVSEL_SHIFT
 pci_regs.h, 364
PCI_STATUS_DEVSEL_SLOW
 pci_regs.h, 364
PCI_STATUS_FAST_BACK
 pci_regs.h, 364
PCI_STATUS_INTERRUPT
 pci_regs.h, 364
PCI_STATUS_PARITY
 pci_regs.h, 364
PCI_STATUS_REC_MASTER_ABORT
 pci_regs.h, 364
PCI_STATUS_REC_TARGET_ABORT
 pci_regs.h, 364
PCI_STATUS_SIG_SYSTEM_ERROR
 pci_regs.h, 364
PCI_STATUS_SIG_TARGET_ABORT
 pci_regs.h, 365
PCI_STATUS_UDF
 pci_regs.h, 365
pci_strings.h, 373, 374
 PciConfRegData2Str, 373
 PciExpressConfRegData2Str, 374
PCI_SVID
 pci_regs.h, 366
PCI_VID
 pci_regs.h, 365
pciCaps
 WD_PCI_SCAN_CAPS, 248
 WDC_PCI_SCAN_CAPS_RESULT, 261
PciConfRegData2Str
 pci_strings.h, 373
PCIE_CAP_ID
 pci_regs.h, 366
PCIE_CONFIG_REGS_OFFSET
 pci_regs.h, 366
PciEventCreate
 windrvr_events.h, 598
PciExpressConfRegData2Str
 pci_strings.h, 374
pciSlot
 WD_EVENT, 226
 WD_PCI_CARD_INFO, 244
 WD_PCI_CONFIG_DUMP, 246
 WD_PCI_SCAN_CAPS, 248
 WD_PCI_SRIOV, 251
pConfigs
 WDU_DEVICE, 269
PCSTR
 cstring.h, 295
pCtx
 WDC_DEVICE, 259
pData
 WD_KERNEL_PLUGIN_CALL, 241
 WDC_INTERRUPT_PARAMS, 261
pEndpointDescriptors

WDU_ALTERNATE_SETTING, 265
pfDeviceAttach
 WDU_EVENT_TABLE, 274
pfDeviceDetach
 WDU_EVENT_TABLE, 274
pfPowerChange
 WDU_EVENT_TABLE, 275
PHYS_ADDR
 windrvr.h, 557
pInterfaces
 WDU_CONFIGURATION, 266
Pipe0
 WDU_DEVICE, 269
PIPE_TYPE_BULK
 windrvr_usb.h, 604
PIPE_TYPE_CONTROL
 windrvr_usb.h, 604
PIPE_TYPE_INTERRUPT
 windrvr_usb.h, 604
PIPE_TYPE_ISOCRONOUS
 windrvr_usb.h, 604
pKernelAddr
 WD_DMA, 222
 WD_KERNEL_BUFFER, 239
pPhysicalAddr
 WD_DMA_PAGE, 223
 WD_ITEMS, 237
pPipes
 WDU_ALTERNATE_SETTING, 265
pPort
 WD_TRANSFER, 254
pReserved
 WD_ITEMS, 237
pReserved2
 WD_ITEMS, 237
PRI64
 windrvr.h, 526
print2wstr
 utils.h, 381
PrintDbgMessage
 utils.h, 381
procInfo
 WD_IPC_REGISTER, 232
 WD_IPC_SCAN_PROCS, 233
 WDS_IPC_SCAN_RESULT, 264
PTR32
 kp_pci.c, 611
ptr32
 windrvr_32bit.h, 594
pTransAddr
 WD_ITEMS, 237
pTransCmds
 WDC_INTERRUPT_PARAMS, 261
pUserAddr
 WD_DMA, 222
 WD_KERNEL_BUFFER, 239
pUserData
 WDU_EVENT_TABLE, 275

pUserDirectAddr
 WD_ITEMS, 238
pUserDirectMemAddr
 WDC_ADDR_DESC, 257
PWDC_DEVICE
 wdc_defs.h, 401

qwBytes
 WD_ITEMS, 238
 WD_KERNEL_BUFFER, 239
 WDC_ADDR_DESC, 257
qwMsgData
 WD_EVENT, 226
 WD_IPC_SEND, 234
 WDS_IPC_MSG_RX, 263
Qword
 WD_TRANSFER, 254

REGKEY_BUFSIZE
 windrvr.h, 526
reserved
 WDC_ADDR_DESC, 257
RM_BYTE
 windrvr.h, 573
RM_DWORD
 windrvr.h, 573
RM_QWORD
 windrvr.h, 573
RM_SBYTE
 windrvr.h, 573
RM_SDWORD
 windrvr.h, 573
RM_SQWORD
 windrvr.h, 573
RM_SWORD
 windrvr.h, 573
RM_WORD
 windrvr.h, 573
ROOT_CAPS
 pci_regs.h, 366
ROOT_CTL
 pci_regs.h, 367
ROOT_STS
 pci_regs.h, 367
RP_BYTE
 windrvr.h, 573
RP_DWORD
 windrvr.h, 573
RP_QWORD
 windrvr.h, 573
RP_SBYTE
 windrvr.h, 573
RP_SDWORD
 windrvr.h, 573
RP_SQWORD
 windrvr.h, 573
RP_SWORD
 windrvr.h, 573
RP_WORD

windrvr.h, 573
S_ALL
 windrvr.h, 564
S_CARD_REG
 windrvr.h, 564
S_DMA
 windrvr.h, 564
S_EVENT
 windrvr.h, 564
S_INT
 windrvr.h, 564
S_IO
 windrvr.h, 564
S_IPC
 windrvr.h, 564
S_KER_BUF
 windrvr.h, 564
S_KER_DRV
 windrvr.h, 564
S_KER_PLUG
 windrvr.h, 564
S_LICENSE
 windrvr.h, 564
S_MEM
 windrvr.h, 564
S_MISC
 windrvr.h, 564
S_PCI
 windrvr.h, 564
S_PNP
 windrvr.h, 564
S_USB
 windrvr.h, 564
SAFE_STRING
 windrvr.h, 526
searchId
 WD_PCI_SCAN_CARDS, 249
SetupPacket
 WDU_TRANSFER, 291
SIZE_OF_WD_DMA
 windrvr.h, 526
SIZE_OF_WD_EVENT
 windrvr.h, 526
SLEEP_BUSY
 windrvr.h, 561
SLEEP_NON_BUSY
 windrvr.h, 561
SleepWrapper
 utils.h, 382
slot
 WDC_DEVICE, 259
SLOT_CAPS
 pci_regs.h, 366
SLOT_CAPS2
 pci_regs.h, 367
SLOT_CTL
 pci_regs.h, 366
SLOT_CTL2
 pci_regs.h, 367
 pci_regs.h, 367
SLOT_STS
 pci_regs.h, 366
SLOT_STS2
 pci_regs.h, 367
snprintf
 utils.h, 377
sprintf
 CCString, 208
Stat2Str
 status_strings.h, 375
status_strings.h, 375
 Stat2Str, 375
strcmp
 CCString, 208
strcpy
 kpstlplib.h, 303
strcmp
 CCString, 208
 cstring.h, 295
 windrvr.h, 526
StrRemove
 CCString, 208
StrReplace
 CCString, 208
substr
 CCString, 208
tolower
 CCString, 208
tolower_copy
 CCString, 208
toupper
 CCString, 209
toupper_copy
 CCString, 209
trim
 CCString, 209
TRUE
 kpstlplib.h, 299
type
 WDU_PIPE_INFO, 284
u
 WD_EVENT, 226
u16
 wd_types.h, 392
u32
 wd_types.h, 392
u64
 wd_types.h, 393
u8
 wd_types.h, 393
UINT32
 windrvr.h, 557
UINT64
 windrvr.h, 557
UNUSED_VAR
 windrvr.h, 527

UPRI
 windrvr.h, 527

UPTR
 windrvr.h, 558

Usb
 WD_EVENT, 226

USB_ABORT_PIPE
 windrvr_usb.h, 604

USB_BULK_INT_URB_SIZE_OVERRIDE_128K
 windrvr_usb.h, 604

USB_DIR
 windrvr_usb.h, 604

USB_DIR_IN
 windrvr_usb.h, 604

USB_DIR_IN_OUT
 windrvr_usb.h, 604

USB_DIR_OUT
 windrvr_usb.h, 604

USB_FULL_TRANSFER
 windrvr_usb.h, 604

USB_ISOCH_ASAP
 windrvr_usb.h, 604

USB_ISOCH_FULL_PACKETS_ONLY
 windrvr_usb.h, 604

USB_ISOCH_NOASAP
 windrvr_usb.h, 604

USB_ISOCH_RESET
 windrvr_usb.h, 604

USB_PIPE_TYPE
 windrvr_usb.h, 604

USB_SHORT_TRANSFER
 windrvr_usb.h, 604

USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL
 windrvr_usb.h, 604

USB_TRANSFER_HALT
 windrvr_usb.h, 604

UsbEventCreate
 windrvr_events.h, 598

USE_MULTI_TRANSFER
 kp_pci.c, 611

UtilGetFileName
 utils.h, 382

UtilGetFileSize
 utils.h, 382

UtilGetStringFromUser
 utils.h, 383

utils.h, 376, 384

 GetNumberOfProcessors, 378

 GetPageSize, 378

 HANDLER_FUNC, 378

 INFINITE, 377

 MAX_PATH, 377

 OsEventClose, 378

 OsEventCreate, 379

 OsEventReset, 379

 OsEventSignal, 379

 OsEventWait, 379

 OsMemoryBarrier, 377

OsmutexClose, 380

OsmutexCreate, 380

OsmutexLock, 380

OsmutexUnlock, 381

print2wstr, 381

PrintDbgMessage, 381

SleepWrapper, 382

sprintf, 377

UtilGetFileName, 382

UtilGetFileSize, 382

UtilGetStringFromUser, 383

vPrintDbgMessage, 383

vsnprintf, 377

va_copy
 windrvr.h, 527

vPrintDbgMessage
 utils.h, 383

vsnprintf
 utils.h, 377

vsprintf
 CCString, 209

WD_ACCEPT_CONTROL
 windrvr.h, 571

WD_ACKNOWLEDGE
 windrvr.h, 571

WD_ACTIONS_ALL
 windrvr.h, 527

WD_ACTIONS_POWER
 windrvr.h, 527

WD_BUS, 212

 dwBusNum, 213

 dwBusType, 213

 dwDomainNum, 213

 dwSlotFunc, 213

WD_BUS_EISA
 windrvr.h, 562

WD_BUS_ISA
 windrvr.h, 562

WD_BUS_PCI
 windrvr.h, 562

WD_BUS_TYPE
 windrvr.h, 558

WD_BUS_UNKNOWN
 windrvr.h, 562

WD_BUS_USB
 windrvr.h, 562

WD_BUS_V30
 windrvr.h, 558

WD_CANT_OBTAIN_PDO
 windrvr.h, 567

WD_CARD, 213

 dwItems, 214

 Item, 214

WD_CARD_CLEANUP, 214

 Cmd, 214

 dwCmds, 215

 dwOptions, 215

hCard, 215
WD_CARD_ITEMS
 windrvr.h, 562
WD_CARD_REGISTER, 215
 Card, 216
 cDescription, 216
 cName, 216
 dwOptions, 216
 fCheckLockOnly, 216
 hCard, 216
WD_CARD_REGISTER_V118
 windrvr.h, 558
WD_CARD_V118
 windrvr.h, 558
WD_CardCleanupSetup
 windrvr.h, 527
WD_CardRegister
 windrvr.h, 528
WD_CardUnregister
 windrvr.h, 529
WD_Close
 wd_log.h, 390
 windrvr.h, 529
WD_CloseLocal
 windrvr.h, 529
WD_CloseLog
 wd_log.h, 390
WD_CPU_SPEC
 windrvr.h, 529
WD_CTL_CODE
 windrvr.h, 529
WD_CTL_DECODE_FUNC
 windrvr.h, 530
WD_CTL_DECODE_TYPE
 windrvr.h, 530
WD_CTL_IS_64BIT_AWARE
 windrvr.h, 530
WD_DATA_MISMATCH
 windrvr.h, 566
WD_DATA_MODEL
 windrvr.h, 530
WD_DEBUG, 217
 dwBufferSize, 217
 dwCmd, 217
 dwLevel, 217
 dwLevelMessageBox, 218
 dwSection, 218
WD_Debug
 windrvr.h, 530
WD_DEBUG_ADD, 218
 dwLevel, 218
 dwSection, 218
 pcBuffer, 219
WD_DEBUG_ADD_V503
 windrvr.h, 558
WD_DEBUG_DUMP, 219
 cBuffer, 219
WD_DEBUG_DUMP_V40
 windrvr.h, 558
WD_DEBUG_V40
 windrvr.h, 558
WD_DebugAdd
 windrvr.h, 530
WD_DebugDump
 windrvr.h, 531
WD_DEFAULT_DRIVER_NAME
 windrvr.h, 531
WD_DEFAULT_DRIVER_NAME_BASE
 windrvr.h, 531
WD_DEVICE_NOT_FOUND
 windrvr.h, 567
WD_DEVICE_PCI
 windrvr.h, 571
WD_DEVICE_REGISTRY_PROPERTY
 windrvr_usb.h, 604
WD_DEVICE_USB
 windrvr.h, 571
WD_DMA, 219
 DMATransactionCallback, 220
 DMATransactionCallbackCtx, 220
 dwAlignment, 220
 dwBytes, 220
 dwBytesTransferred, 221
 dwMaxTransferSize, 221
 dwOptions, 221
 dwPages, 221
 dwTransferElementSize, 221
 hCard, 221
 hDma, 222
 Page, 222
 pKernelAddr, 222
 pUserAddr, 222
WD_DMA_OPTIONS
 windrvr.h, 565
WD_DMA_PAGE, 222
 dwBytes, 222
 pPhysicalAddr, 223
WD_DMA_PAGE_V80
 windrvr.h, 558
WD_DMA_PAGES
 windrvr.h, 560
WD_DMA_V80
 windrvr.h, 558
WD_DMALOCK
 windrvr.h, 531
WD_DMASyncCpu
 windrvr.h, 533
WD_DMASyncclo
 windrvr.h, 533
WD_DMATransactionExecute
 windrvr.h, 534
WD_DMATransactionInit
 windrvr.h, 534
WD_DMATransactionRelease
 windrvr.h, 535
WD_DMATransactionUninit

windrivr.h, 535
WD_DMATransferCompletedAndCheck
 windrvr.h, 535
WD_DMALock
 windrvr.h, 536
WD_DRIVER_NAME
 windrvr.h, 536
WD_DRIVER_NAME_PREFIX
 windrvr.h, 536
WD_DriverName
 windrvr.h, 574
WD_ERROR_CODES
 windrvr.h, 566
WD_EVENT, 223
 cardId, 224
 dwAction, 224
 dwEventId, 224
 dwEventType, 224
 dwGroupID, 224
 dwMsgID, 224
 dwNumMatchTables, 224
 dwOptions, 225
 dwSenderId, 225
 dwSubGroupID, 225
 dwUniqueId, 225
 hEvent, 225
 hIpc, 225
 hKernelPlugIn, 225
 Ipc, 226
 matchTables, 226
 Pci, 226
 pciSlot, 226
 qwMsgData, 226
 u, 226
 Usb, 226
WD_EVENT_ACTION
 windrvr.h, 570
WD_EVENT_OPTION
 windrvr.h, 571
WD_EVENT_TYPE
 windrvr.h, 558
WD_EVENT_TYPE_IPC
 windrvr.h, 561
WD_EVENT_TYPE_PCI
 windrvr.h, 561
WD_EVENT_TYPE_UNKNOWN
 windrvr.h, 561
WD_EVENT_TYPE_USB
 windrvr.h, 561
WD_EVENT_V121
 windrvr.h, 559
WD_EventPull
 windrvr.h, 536
WD_EventRegister
 windrvr.h, 536
WD_EventSend
 windrvr.h, 536
WD_EventUnregister
 windrvr.h, 537
WD_FAILED_ENABLING_INTERRUPT
 windrvr.h, 567
WD_FAILED_KERNEL_MAPPING
 windrvr.h, 567
WD_FAILED_USER_MAPPING
 windrvr.h, 567
WD_FORCE_CLEANUP
 windrvr.h, 563
WD_FUNCTION
 wd_log.h, 390
 windrvr.h, 537
WD_GET_DEVICE_PROPERTY, 226
 dwBytes, 227
 dwOptions, 227
 dwProperty, 227
 dwUniqueId, 227
 h, 227
 hDevice, 227
 pBuf, 228
WD_GET_DEVICE_PROPERTY_OPTION
 windrvr.h, 571
WD_GetDeviceProperty
 windrvr.h, 537
WD_INCORRECT_VERSION
 windrvr.h, 568
WD_INSERT
 windrvr.h, 570
WD_INSUFFICIENT_RESOURCES
 windrvr.h, 566
WD_IntCount
 windrvr.h, 537
WD_IntDisable
 windrvr.h, 537
WD_IntEnable
 windrvr.h, 538
WD_INTERRUPT, 228
 Cmd, 228
 dwCmds, 229
 dwCounter, 229
 dwEnabledIntType, 229
 dwLastMessage, 229
 dwLost, 229
 dwOptions, 229
 fEnableOk, 229
 fStopped, 230
 hInterrupt, 230
 kpCall, 230
WD_INTERRUPT_NOT_ENABLED
 windrvr.h, 567
WD_INTERRUPT_V91
 windrvr.h, 559
WD_INTERRUPT_WAIT_RESULT
 windrvr.h, 571
WD_IntWait
 windrvr.h, 539
WD_INVALID_32BIT_APP
 windrvr.h, 568

WD_INVALID_HANDLE
windrvr.h, 566

WD_INVALID_IOCTL
windrvr.h, 568

WD_INVALID_PARAMETER
windrvr.h, 568

WD_INVALID_PIPE_NUMBER
windrvr.h, 566

WD_IPC_ALL_MSG
windrvr.h, 539

WD_IPC_MAX_PROCS
windrvr.h, 562

WD_IPC_MULTICAST
windrvr.h, 563

WD_IPC_MULTICAST_MSG
windrvr.h, 571

WD_IPC_PROCESS, 230
cProcessName, 231
dwGroupID, 231
dwSubGroupID, 231
hlpc, 231

WD_IPC_PROCESS_V121
windrvr.h, 559

WD_IPC_REGISTER, 231
dwOptions, 232
procInfo, 232

WD_IPC_REGISTER_V121
windrvr.h, 559

WD_IPC_SCAN_PROCS, 232
dwNumProcs, 232
hlpc, 232
procInfo, 233

WD_IPC_SCAN_PROCS_V121
windrvr.h, 559

WD_IPC_SEND, 233
dwMsgID, 233
dwOptions, 233
dwRecipientID, 233
hlpc, 234
qwMsgData, 234

WD_IPC_SEND_V121
windrvr.h, 559

WD_IPC_SUBGROUP_MULTICAST
windrvr.h, 563

WD_IPC_UID_UNICAST
windrvr.h, 563

WD_IPC_UNICAST_MSG
windrvr.h, 571

WD_IpcRegister
windrvr.h, 539

WD_IpcScanProcs
windrvr.h, 540

WD_IpcSend
windrvr.h, 540

WD_IpcUnRegister
windrvr.h, 540

WD_IRP_CANCELED
windrvr.h, 567

WD_ITEM_MEM_ALLOW_CACHE
windrvr.h, 572

WD_ITEM_MEM_DO_NOT_MAP_KERNEL
windrvr.h, 572

WD_ITEM_MEM_OPTIONS
windrvr.h, 571

WD_ITEM_MEM_USER_MAP
windrvr.h, 572

WD_ITEMS, 234
Bus, 235
dwBar, 235
dwBytes, 235
dwInterrupt, 235
dwOptions, 236
dwReserved1, 236
fNotSharable, 236
hInterrupt, 236
I, 236
Int, 236
IO, 236
item, 237
Mem, 237
pAddr, 237
pPhysicalAddr, 237
pReserved, 237
pReserved2, 237
pTransAddr, 237
pUserDirectAddr, 238
qwBytes, 238

WD_ITEMS_V118
windrvr.h, 559

WD_KER_BUF_OPTION
windrvr.h, 572

WD_KERNEL_BUFFER, 238
dwOptions, 238
hKerBuf, 239
pKernelAddr, 239
pUserAddr, 239
qwBytes, 239

WD_KERNEL_BUFFER_V121
windrvr.h, 559

WD_KERNEL_PLUGIN, 239
cDriverName, 240
cDriverPath, 240
hKernelPlugIn, 240
PAD_TO_64, 240

WD_KERNEL_PLUGIN_CALL, 240
dwMessage, 241
dwResult, 241
hKernelPlugIn, 241
pData, 241

WD_KERNEL_PLUGIN_CALL_V40
windrvr.h, 559

WD_KERNEL_PLUGIN_V40
windrvr.h, 559

WD_KernelBufLock
windrvr.h, 541

WD_KernelBufUnlock

windrvr.h, 541
WD_KernelPlugInCall
 windrvr.h, 543
WD_KernelPlugInClose
 windrvr.h, 543
WD_KernelPlugInOpen
 windrvr.h, 545
WD_KERPLUG_FAILURE
 windrvr.h, 567
wd_kp.h, 385, 388
 __KERNEL__, 386
 __KERPLUG__, 386
 KP_FUNC_CALL, 386
 KP_FUNC_CLOSE, 386
 KP_FUNC_EVENT, 387
 KP_FUNC_INT_AT_DPC, 387
 KP_FUNC_INT_AT_DPC_MSI, 387
 KP_FUNC_INT_AT_IRQL, 387
 KP_FUNC_INT_AT_IRQL_MSI, 387
 KP_FUNC_INT_DISABLE, 387
 KP_FUNC_INT_ENABLE, 387
 KP_FUNC_OPEN, 388
 KP_Init, 388
WD_LICENSE, 241
 cLicense, 242
WD_License
 windrvr.h, 545
WD_LICENSE_LENGTH
 windrvr.h, 546
WD_LICENSE_V122
 windrvr.h, 559
wd_log.h, 389, 392
 WD_Close, 390
 WD_CloseLog, 390
 WD_FUNCTION, 390
 WD_LogAdd, 390
 WD_LogStart, 391
 WD_LogStop, 391
 WD_Open, 390
 WD_OpenLog, 391
 WdFunctionLog, 391
WD_LogAdd
 wd_log.h, 390
WD_LogStart
 wd_log.h, 391
WD_LogStop
 wd_log.h, 391
WD_MAJOR_VER
 wd_ver.h, 394
WD_MAJOR_VER_STR
 wd_ver.h, 394
WD_MATCH_EXCLUDE
 windrvr.h, 561
WD_MAX_DRIVER_NAME_LENGTH
 windrvr.h, 546
WD_MAX_KP_NAME_LENGTH
 windrvr.h, 546
WD_MINOR_VER
 wd_ver.h, 394
WD_MINOR_VER_STR
 wd_ver.h, 394
WD_MORE_PROCESSING_REQUIRED
 windrvr.h, 568
WD_MultiTransfer
 windrvr.h, 546
WD_NO_DEVICE_OBJECT
 windrvr.h, 568
WD_NO_EVENTS
 windrvr.h, 568
WD_NO_LICENSE
 windrvr.h, 566
WD_NO_RESOURCES_ON_DEVICE
 windrvr.h, 568
WD_NOT_IMPLEMENTED
 windrvr.h, 567
WD_OBSOLETE
 windrvr.h, 570
WD_Open
 wd_log.h, 390
 windrvr.h, 547
WD_OpenLocal
 windrvr.h, 547
WD_OpenLog
 wd_log.h, 391
WD_OpenStreamLocal
 windrvr.h, 547
WD_OPERATION_ALREADY_DONE
 windrvr.h, 567
WD_OPERATION_FAILED
 windrvr.h, 568
WD_OS_INFO, 242
 cBuild, 242
 cCsdVersion, 242
 cCurrentVersion, 242
 cInstallationType, 243
 cProdName, 243
 dwMajorVersion, 243
 dwMinorVersion, 243
WD_PCI_CAP, 243
 dwCapId, 244
 dwCapOffset, 244
WD_PCI_CAP_ID_ALL
 windrvr.h, 561
WD_PCI_CARD_INFO, 244
 Card, 244
 pciSlot, 244
WD_PCI_CARD_INFO_V118
 windrvr.h, 559
WD_PCI_CARDS
 windrvr.h, 563
WD_PCI_CONFIG_DUMP, 245
 dwBytes, 245
 dwOffset, 245
 dwResult, 245
 flsRead, 246
 pBuffer, 246

pciSlot, 246
WD_PCI_CONFIG_DUMP_V30
 windrvr.h, 560
WD_PCI_ID, 246
 dwDeviceId, 246
 dwVendorId, 247
WD_PCI_MAX_CAPS
 windrvr.h, 563
WD_PCI_SCAN_BY_TOPOLOGY
 windrvr.h, 572
WD_PCI_SCAN_CAPS, 247
 dwCapId, 247
 dwNumCaps, 247
 dwOptions, 248
 pciCaps, 248
 pciSlot, 248
WD_PCI_SCAN_CAPS_BASIC
 windrvr.h, 572
WD_PCI_SCAN_CAPS_EXTENDED
 windrvr.h, 572
WD_PCI_SCAN_CAPS_OPTIONS
 windrvr.h, 572
WD_PCI_SCAN_CAPS_V118
 windrvr.h, 560
WD_PCI_SCAN_CARDS, 248
 cardId, 249
 cardSlot, 249
 dwCards, 249
 dwOptions, 249
 searchId, 249
WD_PCI_SCAN_CARDS_V124
 windrvr.h, 560
WD_PCI_SCAN_DEFAULT
 windrvr.h, 572
WD_PCI_SCAN_INCLUDE_DOMAINS
 windrvr.h, 572
WD_PCI_SCAN_OPTIONS
 windrvr.h, 572
WD_PCI_SCAN_REGISTERED
 windrvr.h, 572
WD_PCI_SLOT, 250
 dwBus, 250
 dwDomain, 250
 dwFunction, 250
 dwSlot, 250
WD_PCI_SRIOV, 251
 dwNumVFs, 251
 pciSlot, 251
WD_PCI_SRIOV_V122
 windrvr.h, 560
WD_PciConfigDump
 windrvr.h, 547
WD_PciGetCardInfo
 windrvr.h, 548
WD_PciScanCaps
 windrvr.h, 549
WD_PciScanCards
 windrvr.h, 549
WD_PciSriovDisable
 windrvr.h, 550
WD_PciSriovEnable
 windrvr.h, 550
WD_PciSriovGetNumVFs
 windrvr.h, 551
WD_POWER_CHANGED_D0
 windrvr.h, 570
WD_POWER_CHANGED_D1
 windrvr.h, 570
WD_POWER_CHANGED_D2
 windrvr.h, 570
WD_POWER_CHANGED_D3
 windrvr.h, 570
WD_POWER_SYSTEM_HIBERNATE
 windrvr.h, 571
WD_POWER_SYSTEM_SHUTDOWN
 windrvr.h, 571
WD_POWER_SYSTEM_SLEEPING1
 windrvr.h, 570
WD_POWER_SYSTEM_SLEEPING2
 windrvr.h, 571
WD_POWER_SYSTEM_SLEEPING3
 windrvr.h, 571
WD_POWER_SYSTEM_WORKING
 windrvr.h, 570
WD_PROCESS_NAME_LENGTH
 windrvr.h, 551
WD_PROD_NAME
 windrvr.h, 551
WD_READ_WRITE_CONFLICT
 windrvr.h, 566
WD_REMOVE
 windrvr.h, 570
WD_RESOURCE_OVERLAP
 windrvr.h, 567
WD_SET_CONFIGURATION_FAILED
 windrvr.h, 567
WD_SharedIntDisable
 windrvr.h, 551
WD_SharedIntEnable
 windrvr.h, 551
WD_SLEEP, 251
 dwMicroSeconds, 252
 dwOptions, 252
WD_Sleep
 windrvr.h, 552
WD_SLEEP_V40
 windrvr.h, 560
WD_STATUS_INVALID_WD_HANDLE
 windrvr.h, 566
WD_STATUS_SUCCESS
 windrvr.h, 566
WD_StreamClose
 windrvr.h, 552
WD_StreamOpen
 windrvr.h, 552
WD_SUB_MINOR_VER

wd_ver.h, 394
WD_SUB_MINOR_VER_STR
 wd_ver.h, 394
WD_SYSTEM_INTERNAL_ERROR
 windrvr.h, 566
WD_TIME_OUT_EXPIRED
 windrvr.h, 567
WD_TOO_MANY_HANDLES
 windrvr.h, 568
WD_TRANSFER, 252
 Byte, 253
 cmdTrans, 253
 Data, 253
 dwBytes, 253
 dwOptions, 253
 Dword, 254
 fAutoinc, 254
 pBuffer, 254
 pPort, 254
 Qword, 254
 Word, 254
WD_Transfer
 windrvr.h, 553
WD_TRANSFER_CMD
 windrvr.h, 572
WD_TRANSFER_V61
 windrvr.h, 560
WD_TRY AGAIN
 windrvr.h, 568
WD_TYPE
 windrvr.h, 553
wd_types.h, 392, 393
 u16, 392
 u32, 392
 u64, 393
 u8, 393
WD_UGetDeviceData
 windrvr.h, 553
WD_UHaltTransfer
 windrvr.h, 553
WD_UNKNOWN_PIPE_TYPE
 windrvr.h, 566
WD_UResetDevice
 windrvr.h, 553
WD_UResetPipe
 windrvr.h, 554
WD_USAGE, 255
 applications_num, 255
 devices_num, 255
WD_Usage
 windrvr.h, 554
WD_USB_CYCLE_PORT
 windrvr.h, 561
WD_USB_DESCRIPTOR_ERROR
 windrvr.h, 567
WD_USB_HARD_RESET
 windrvr.h, 561
WD_USB_MAX_ALT_SETTINGS
 windrvr_usb.h, 602
WD_USB_MAX_ENDPOINTS
 windrvr_usb.h, 602
WD_USB_MAX_INTERFACES
 windrvr_usb.h, 602
WD_USB_MAX_PIPE_NUMBER
 windrvr_usb.h, 602
WD_USBD_STATUS_BABBLE_DETECTED
 windrvr.h, 569
WD_USBD_STATUS_BAD_START_FRAME
 windrvr.h, 569
WD_USBD_STATUS_BTSTUFF
 windrvr.h, 569
WD_USBD_STATUS_BUFFER_OVERRUN
 windrvr.h, 569
WD_USBD_STATUS_BUFFER_TOO_SMALL
 windrvr.h, 570
WD_USBD_STATUS_BUFFER_UNDERRUN
 windrvr.h, 569
WD_USBD_STATUS_CANCELED
 windrvr.h, 569
WD_USBD_STATUS_CRC
 windrvr.h, 569
WD_USBD_STATUS_DATA_BUFFER_ERROR
 windrvr.h, 569
WD_USBD_STATUS_DATA_OVERRUN
 windrvr.h, 569
WD_USBD_STATUS_DATA_TOGGLE_MISMATCH
 windrvr.h, 569
WD_USBD_STATUS_DATA_UNDERRUN
 windrvr.h, 569
WD_USBD_STATUS_DEV_NOT_RESPONDING
 windrvr.h, 569
WD_USBD_STATUS_DEVICE_GONE
 windrvr.h, 570
WD_USBD_STATUS_ENDPOINT_HALTED
 windrvr.h, 569
WD_USBD_STATUS_ERROR
 windrvr.h, 568
WD_USBD_STATUS_ERROR_BUSY
 windrvr.h, 569
WD_USBD_STATUS_ERROR_SHORT_TRANSFER
 windrvr.h, 569
WD_USBD_STATUS_FIFO
 windrvr.h, 569
WD_USBD_STATUS_FRAME_CONTROL_NOT_OWNED
 windrvr.h, 570
WD_USBD_STATUS_FRAME_CONTROL_OWNED
 windrvr.h, 570
WD_USBD_STATUS_HALTED
 windrvr.h, 568
WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR
 windrvr.h, 570
WD_USBD_STATUS_INAVLID_PIPE_FLAGS
 windrvr.h, 570
WD_USBD_STATUS_INSUFFICIENT_RESOURCES
 windrvr.h, 570
WD_USBD_STATUS_INTERFACE_NOT_FOUND

windrvr.h, 570
WD_USBD_STATUS_INTERNAL_HC_ERROR
 windrvr.h, 569
WD_USBD_STATUS_INVALID_PARAMETER
 windrvr.h, 569
WD_USBD_STATUS_INVALID_PIPE_HANDLE
 windrvr.h, 569
WD_USBD_STATUS_INVALID_URB_FUNCTION
 windrvr.h, 569
WD_USBD_STATUS_ISO_NA_LATE_USBPORT
 windrvr.h, 570
WD_USBD_STATUS_ISO_NOT_ACCESSED_BY_HW
 windrvr.h, 570
WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE
 windrvr.h, 570
WD_USBD_STATUS_ISO_TD_ERROR
 windrvr.h, 570
WD_USBD_STATUS_ISOCH_REQUEST_FAILED
 windrvr.h, 570
WD_USBD_STATUS_NO_BANDWIDTH
 windrvr.h, 569
WD_USBD_STATUS_NO_MEMORY
 windrvr.h, 569
WD_USBD_STATUS_NOT_ACCESSED
 windrvr.h, 569
WD_USBD_STATUS_NOT_SUPPORTED
 windrvr.h, 570
WD_USBD_STATUS_PENDING
 windrvr.h, 568
WD_USBD_STATUS_PID_CHECK_FAILURE
 windrvr.h, 569
WD_USBD_STATUS_REQUEST_FAILED
 windrvr.h, 569
WD_USBD_STATUS_RESERVED1
 windrvr.h, 569
WD_USBD_STATUS_RESERVED2
 windrvr.h, 569
WD_USBD_STATUS_SET_CONFIG_FAILED
 windrvr.h, 570
WD_USBD_STATUS_STALL_PID
 windrvr.h, 569
WD_USBD_STATUS_STATUS_NOT_MAPPED
 windrvr.h, 570
WD_USBD_STATUS_SUCCESS
 windrvr.h, 568
WD_USBD_STATUS_TIMEOUT
 windrvr.h, 570
WD_USBD_STATUS_UNEXPECTED_PID
 windrvr.h, 569
WD_USBD_STATUS_XACT_ERROR
 windrvr.h, 569
WD_USelectiveSuspend
 windrvr.h, 554
WD_USetInterface
 windrvr.h, 554
WD_UStreamClose
 windrvr.h, 554
WD_UStreamFlush
 windrvr.h, 554
 windrvr.h, 554
WD_UStreamGetStatus
 windrvr.h, 555
WD_UStreamOpen
 windrvr.h, 555
WD_UStreamRead
 windrvr.h, 555
WD_UStreamStart
 windrvr.h, 555
WD_UStreamStop
 windrvr.h, 555
WD_UStreamWrite
 windrvr.h, 555
WD_UTransfer
 windrvr.h, 556
WD_UWakeup
 windrvr.h, 556
WD_VER
 wd_ver.h, 395
wd_ver.h, 393, 395
 COPYRIGHTS_FULL_STR, 394
 COPYRIGHTS_YEAR_STR, 394
 WD_MAJOR_VER, 394
 WD_MAJOR_VER_STR, 394
 WD_MINOR_VER, 394
 WD_MINOR_VER_STR, 394
 WD_SUB_MINOR_VER, 394
 WD_SUB_MINOR_VER_STR, 394
 WD_VER, 395
 WD_VER_BETA_STR, 395
 WD_VER_ITOA, 395
 WD_VERSION_MAC_STR, 395
 WD_VERSION_STR, 395
WD_VER_BETA_STR
 wd_ver.h, 395
WD_VER_ITOA
 wd_ver.h, 395
WD_VER_STR
 windrvr.h, 556
WD_VERSION, 255
 cVer, 255
 dwVer, 256
WD_Version
 windrvr.h, 556
WD_VERSION_MAC_STR
 wd_ver.h, 395
WD_VERSION_STR
 wd_ver.h, 395
WD_VERSION_STR_LENGTH
 windrvr.h, 557
WD_VERSION_V30
 windrvr.h, 560
WD_WINDRIVER_NOT_FOUND
 windrvr.h, 568
WD_WINDRIVER_STATUS_ERROR
 windrvr.h, 566
WD_WRONG_UNIQUE_ID
 windrvr.h, 567

WD_ZERO_PACKET_SIZE
windrvr.h, 566

WDC_AD_CFG_SPACE
wdc_lib.h, 411

WDC_ADDR_DESC, 256
dwAddrSpace, 256
dwItemIndex, 256
flsMemory, 257
pAddr, 257
pUserDirectMemAddr, 257
qwBytes, 257
reserved, 257

WDC_ADDR_IS_MEM
wdc_defs.h, 397

WDC_ADDR_MODE
wdc_lib.h, 419

WDC_ADDR_MODE_TO_SIZE
wdc_lib.h, 411

WDC_ADDR_RW_DEFAULT
wdc_lib.h, 420

WDC_ADDR_RW_NO_AUTOINC
wdc_lib.h, 420

WDC_ADDR_RW_OPTIONS
wdc_lib.h, 419

WDC_ADDR_SIZE
wdc_lib.h, 419

WDC_ADDR_SIZE_TO_MODE
wdc_lib.h, 411

WDC_AddrSpacelsActive
wdc_lib.h, 420

WDC_CallKerPlug
wdc_lib.h, 420

WDC_CardCleanupSetup
wdc_lib.h, 421

WDC_DBG_DBM_ERR
wdc_lib.h, 411

WDC_DBG_DBM_FILE_ERR
wdc_lib.h, 411

WDC_DBG_DBM_FILE_TRACE
wdc_lib.h, 411

WDC_DBG_DBM_TRACE
wdc_lib.h, 412

WDC_DBG_DEFAULT
wdc_lib.h, 412

WDC_DBG_FILE_ERR
wdc_lib.h, 412

WDC_DBG_FILE_TRACE
wdc_lib.h, 412

WDC_DBG_FULL
wdc_lib.h, 412

WDC_DBG_LEVEL_ERR
wdc_lib.h, 412

WDC_DBG_LEVEL_TRACE
wdc_lib.h, 412

WDC_DBG_NONE
wdc_lib.h, 413

WDC_DBG_OPTIONS
wdc_lib.h, 419

WDC_DBG_OUT_DBM
wdc_lib.h, 413

WDC_DBG_OUT_FILE
wdc_lib.h, 413

wdc_defs.h, 396, 402
PWDC_DEVICE, 401

WDC_ADDR_IS_MEM, 397

WDC_DEVICE, 401

WDC_GET_ADDR_DESC, 397

WDC_GET_ADDR_SPACE_SIZE, 397

WDC_GET_CARD_HANDLE, 398

WDC_GET_ENABLED_INT_LAST_MSG, 398

WDC_GET_ENABLED_INT_TYPE, 399

WDC_GET_INT_OPTIONS, 399

WDC_GET_KP_HANDLE, 399

WDC_GET_PCARD, 400

WDC_GET_PPCI_ID, 400

WDC_GET_PPCI_SLOT, 400

WDC_INT_IS_MSI, 401

WDC_IS_KP, 401

WDC_MEM_DIRECT_ADDR, 401

WDC_DEVICE, 257
cardReg, 258
dwNumAddrSpaces, 258
Event, 258
hEvent, 258
hIntThread, 259
id, 259
Int, 259
kerPlug, 259
pAddrDesc, 259
pCtx, 259
slot, 259
wdc_defs.h, 401

WDC_DEVICE_HANDLE
wdc_lib.h, 419

WDC_DIRECTION
wdc_lib.h, 420

WDC_DMABufGet
wdc_lib.h, 421

WDC_DMABufUnlock
wdc_lib.h, 422

WDC_DMAContigBufLock
wdc_lib.h, 423

WDC_DMAGetGlobalHandle
wdc_lib.h, 413

WDC_DMAReservedBufLock
wdc_lib.h, 423

WDC_DMASGBufLock
wdc_lib.h, 424

WDC_DMASyncCpu
wdc_lib.h, 425

WDC_DMASyncclo
wdc_lib.h, 425

WDC_DMATransactionContigInit
wdc_lib.h, 425

WDC_DMATransactionExecute
wdc_lib.h, 426

WDC_DMATransactionRelease
 wdc_lib.h, 427
WDC_DMATransactionSGInit
 wdc_lib.h, 427
WDC_DMATransactionUninit
 wdc_lib.h, 428
WDC_DMATransferCompletedAndCheck
 wdc_lib.h, 428
WDC_DriverClose
 wdc_lib.h, 429
WDC_DriverOpen
 wdc_lib.h, 430
WDC_DRV_OPEN_ALL
 wdc_lib.h, 413
WDC_DRV_OPEN_BASIC
 wdc_lib.h, 413
WDC_DRV_OPEN_CHECK_VER
 wdc_lib.h, 413
WDC_DRV_OPEN_DEFAULT
 wdc_lib.h, 414
WDC_DRV_OPEN_KP
 wdc_lib.h, 414
WDC_DRV_OPEN_OPTIONS
 wdc_lib.h, 419
WDC_DRV_OPEN_REG_LIC
 wdc_lib.h, 414
WDC_Err
 wdc_lib.h, 432
WDC_EventIsRegistered
 wdc_lib.h, 432
WDC_EventRegister
 wdc_lib.h, 433
WDC_EventUnregister
 wdc_lib.h, 435
WDC_GET_ADDR_DESC
 wdc_defs.h, 397
WDC_GET_ADDR_SPACE_SIZE
 wdc_defs.h, 397
WDC_GET_CARD_HANDLE
 wdc_defs.h, 398
WDC_GET_ENABLED_INT_LAST_MSG
 wdc_defs.h, 398
WDC_GET_ENABLED_INT_TYPE
 wdc_defs.h, 399
WDC_GET_INT_OPTIONS
 wdc_defs.h, 399
WDC_GET_KP_HANDLE
 wdc_defs.h, 399
WDC_GET_PCARD
 wdc_defs.h, 400
WDC_GET_PPCI_ID
 wdc_defs.h, 400
WDC_GET_PPCI_SLOT
 wdc_defs.h, 400
WDC_GetBusType
 wdc_lib.h, 436
WDC_GetDevContext
 wdc_lib.h, 436
WDC_GetWDHandle
 wdc_lib.h, 437
WDC_INT_IS_MSI
 wdc_defs.h, 401
WDC_IntDisable
 wdc_lib.h, 437
WDC_IntEnable
 wdc_lib.h, 437
WDC_INTERRUPT_PARAMS, 260
 dwNumCmds, 260
 dwOptions, 260
 funcIntHandler, 260
 fUseKP, 260
 pData, 261
 pTransCmds, 261
WDC_IntIsEnabled
 wdc_lib.h, 439
WDC_IntType2Str
 wdc_lib.h, 439
WDC_IS_KP
 wdc_defs.h, 401
WDC_IsaDeviceClose
 wdc_lib.h, 440
WDC_IsaDeviceOpen
 wdc_lib.h, 440
WDC_KernelPlugInOpen
 wdc_lib.h, 440
wdc_lib.h, 403, 466
 MAX_DESC, 410
 MAX_NAME, 410
 MAX_NAME_DISPLAY, 411
 WDC_AD_CFG_SPACE, 411
 WDC_ADDR_MODE, 419
 WDC_ADDR_MODE_TO_SIZE, 411
 WDC_ADDR_RW_DEFAULT, 420
 WDC_ADDR_RW_NO_AUTOINC, 420
 WDC_ADDR_RW_OPTIONS, 419
 WDC_ADDR_SIZE, 419
 WDC_ADDR_SIZE_TO_MODE, 411
 WDC_AddrSpacelsActive, 420
 WDC_CallKerPlug, 420
 WDC_CardCleanupSetup, 421
 WDC_DBG_DBM_ERR, 411
 WDC_DBG_DBM_FILE_ERR, 411
 WDC_DBG_DBM_FILE_TRACE, 411
 WDC_DBG_DBM_TRACE, 412
 WDC_DBG_DEFAULT, 412
 WDC_DBG_FILE_ERR, 412
 WDC_DBG_FILE_TRACE, 412
 WDC_DBG_FULL, 412
 WDC_DBG_LEVEL_ERR, 412
 WDC_DBG_LEVEL_TRACE, 412
 WDC_DBG_NONE, 413
 WDC_DBG_OPTIONS, 419
 WDC_DBG_OUT_DBM, 413
 WDC_DBG_OUT_FILE, 413
 WDC_DEVICE_HANDLE, 419
 WDC_DIRECTION, 420

WDC_DMABufGet, 421
WDC_DMABufUnlock, 422
WDC_DMAContigBufLock, 423
WDC_DMAGetGlobalHandle, 413
WDC_DMAReservedBufLock, 423
WDC_DMASGBufLock, 424
WDC_DMASyncCpu, 425
WDC_DMASyncIo, 425
WDC_DMATransactionContigInit, 425
WDC_DMATransactionExecute, 426
WDC_DMATransactionRelease, 427
WDC_DMATransactionSGInit, 427
WDC_DMATransactionUninit, 428
WDC_DMATransferCompletedAndCheck, 428
WDC_DriverClose, 429
WDC_DriverOpen, 430
WDC_DRV_OPEN_ALL, 413
WDC_DRV_OPEN_BASIC, 413
WDC_DRV_OPEN_CHECK_VER, 413
WDC_DRV_OPEN_DEFAULT, 414
WDC_DRV_OPEN_KP, 414
WDC_DRV_OPEN_OPTIONS, 419
WDC_DRV_OPEN_REG_LIC, 414
WDC_Err, 432
WDC_EventIsRegistered, 432
WDC_EventRegister, 433
WDC_EventUnregister, 435
WDC_GetBusType, 436
WDC_GetDevContext, 436
WDC_GetWDHandle, 437
WDC_IntDisable, 437
WDC_IntEnable, 437
WDC_IntIsEnabled, 439
WDC_IntType2Str, 439
WDC_IsaDeviceClose, 440
WDC_IsaDeviceOpen, 440
WDC_KernelPlugInOpen, 440
WDC_MODE_16, 419
WDC_MODE_32, 419
WDC_MODE_64, 419
WDC_MODE_8, 419
WDC_MultiTransfer, 441
WDC_PciDeviceClose, 441
WDC_PciDeviceOpen, 442
WDC_PciGetDeviceInfo, 444
WDC_PciGetExpressGen, 446
WDC_PciGetExpressGenBySlot, 446
WDC_PciGetExpressOffset, 446
WDC_PciGetHeaderType, 447
WDC_PciReadCfg, 447
WDC_PciReadCfg16, 447
WDC_PciReadCfg32, 448
WDC_PciReadCfg64, 448
WDC_PciReadCfg8, 448
WDC_PciReadCfgBySlot, 449
WDC_PciReadCfgBySlot16, 449
WDC_PciReadCfgBySlot32, 450
WDC_PciReadCfgBySlot64, 450
WDC_PciReadCfgBySlot8, 450
WDC_PciScanCaps, 451
WDC_PciScanCapsBySlot, 451
WDC_PciScanDevices, 452
WDC_PciScanDevicesByTopology, 452
WDC_PciScanExtCaps, 453
WDC_PciScanRegisteredDevices, 453
WDC_PciSriovDisable, 453
WDC_PciSriovEnable, 454
WDC_PciSriovGetNumVFs, 455
WDC_PciWriteCfg, 455
WDC_PciWriteCfg16, 456
WDC_PciWriteCfg32, 456
WDC_PciWriteCfg64, 456
WDC_PciWriteCfg8, 457
WDC_PciWriteCfgBySlot, 457
WDC_PciWriteCfgBySlot16, 458
WDC_PciWriteCfgBySlot32, 458
WDC_PciWriteCfgBySlot64, 458
WDC_PciWriteCfgBySlot8, 459
WDC_READ, 420
WDC_READ_WRITE, 420
WDC_ReadAddr16, 459
WDC_ReadAddr32, 460
WDC_ReadAddr64, 460
WDC_ReadAddr8, 460
WDC_ReadAddrBlock, 461
WDC_ReadAddrBlock16, 414
WDC_ReadAddrBlock32, 414
WDC_ReadAddrBlock64, 415
WDC_ReadAddrBlock8, 415
WDC_ReadMem16, 415
WDC_ReadMem32, 415
WDC_ReadMem64, 416
WDC_ReadMem8, 416
WDC_SetDebugOptions, 461
WDC_SIZE_16, 416
WDC_SIZE_32, 416
WDC_SIZE_64, 416
WDC_SIZE_8, 416
WDC_Sleep, 463
WDC_SLEEP_BUSY, 417
WDC_SLEEP_NON_BUSY, 417
WDC_SLEEP_OPTIONS, 419
WDC_Trace, 464
WDC_Version, 464
WDC_WRITE, 420
WDC_WriteAddr16, 464
WDC_WriteAddr32, 465
WDC_WriteAddr64, 465
WDC_WriteAddr8, 465
WDC_WriteAddrBlock, 466
WDC_WriteAddrBlock16, 417
WDC_WriteAddrBlock32, 417
WDC_WriteAddrBlock64, 417
WDC_WriteAddrBlock8, 418
WDC_WriteMem16, 418
WDC_WriteMem32, 418

WDC_WriteMem64, 418
WDC_WriteMem8, 418
WDC_MEM_DIRECT_ADDR
 wdc_defs.h, 401
WDC_MODE_16
 wdc_lib.h, 419
WDC_MODE_32
 wdc_lib.h, 419
WDC_MODE_64
 wdc_lib.h, 419
WDC_MODE_8
 wdc_lib.h, 419
WDC_MultiTransfer
 wdc_lib.h, 441
WDC_PCI_HEADER_TYPE
 pci_regs.h, 367
WDC_PCI_SCAN_CAPS_RESULT, 261
 dwNumCaps, 261
 pciCaps, 261
WDC_PCI_SCAN_RESULT, 262
 deviceId, 262
 deviceSlot, 262
 dwNumDevices, 262
WDC_PciDeviceClose
 wdc_lib.h, 441
WDC_PciDeviceOpen
 wdc_lib.h, 442
WDC_PciGetDeviceInfo
 wdc_lib.h, 444
WDC_PciGetExpressGen
 wdc_lib.h, 446
WDC_PciGetExpressGenBySlot
 wdc_lib.h, 446
WDC_PciGetExpressOffset
 wdc_lib.h, 446
WDC_PciGetHeaderType
 wdc_lib.h, 447
WDC_PciReadCfg
 wdc_lib.h, 447
WDC_PciReadCfg16
 wdc_lib.h, 447
WDC_PciReadCfg32
 wdc_lib.h, 448
WDC_PciReadCfg64
 wdc_lib.h, 448
WDC_PciReadCfg8
 wdc_lib.h, 448
WDC_PciReadCfgBySlot
 wdc_lib.h, 449
WDC_PciReadCfgBySlot16
 wdc_lib.h, 449
WDC_PciReadCfgBySlot32
 wdc_lib.h, 450
WDC_PciReadCfgBySlot64
 wdc_lib.h, 450
WDC_PciReadCfgBySlot8
 wdc_lib.h, 450
WDC_PciScanCaps
 wdc_lib.h, 451
WDC_PciScanCapsBySlot
 wdc_lib.h, 451
WDC_PciScanDevices
 wdc_lib.h, 452
WDC_PciScanDevicesByTopology
 wdc_lib.h, 452
WDC_PciScanExtCaps
 wdc_lib.h, 453
WDC_PciScanRegisteredDevices
 wdc_lib.h, 453
WDC_PciSriovDisable
 wdc_lib.h, 453
WDC_PciSriovEnable
 wdc_lib.h, 454
WDC_PciSriovGetNumVFs
 wdc_lib.h, 455
WDC_PciWriteCfg
 wdc_lib.h, 455
WDC_PciWriteCfg16
 wdc_lib.h, 456
WDC_PciWriteCfg32
 wdc_lib.h, 456
WDC_PciWriteCfg64
 wdc_lib.h, 456
WDC_PciWriteCfg8
 wdc_lib.h, 457
WDC_PciWriteCfgBySlot
 wdc_lib.h, 457
WDC_PciWriteCfgBySlot16
 wdc_lib.h, 458
WDC_PciWriteCfgBySlot32
 wdc_lib.h, 458
WDC_PciWriteCfgBySlot64
 wdc_lib.h, 458
WDC_PciWriteCfgBySlot8
 wdc_lib.h, 459
WDC_READ
 wdc_lib.h, 420
WDC_READ_WRITE
 wdc_lib.h, 420
WDC_ReadAddr16
 wdc_lib.h, 459
WDC_ReadAddr32
 wdc_lib.h, 460
WDC_ReadAddr64
 wdc_lib.h, 460
WDC_ReadAddr8
 wdc_lib.h, 460
WDC_ReadAddrBlock
 wdc_lib.h, 461
WDC_ReadAddrBlock16
 wdc_lib.h, 414
WDC_ReadAddrBlock32
 wdc_lib.h, 414
WDC_ReadAddrBlock64
 wdc_lib.h, 415
WDC_ReadAddrBlock8

wdc_lib.h, 415
WDC_ReadMem16
 wdc_lib.h, 415
WDC_ReadMem32
 wdc_lib.h, 415
WDC_ReadMem64
 wdc_lib.h, 416
WDC_ReadMem8
 wdc_lib.h, 416
WDC_SetDebugOptions
 wdc_lib.h, 461
WDC_SIZE_16
 wdc_lib.h, 416
WDC_SIZE_32
 wdc_lib.h, 416
WDC_SIZE_64
 wdc_lib.h, 416
WDC_SIZE_8
 wdc_lib.h, 416
WDC_Sleep
 wdc_lib.h, 463
WDC_SLEEP_BUSY
 wdc_lib.h, 417
WDC_SLEEP_NON_BUSY
 wdc_lib.h, 417
WDC_SLEEP_OPTIONS
 wdc_lib.h, 419
WDC_Trace
 wdc_lib.h, 464
WDC_Version
 wdc_lib.h, 464
WDC_WRITE
 wdc_lib.h, 420
WDC_WriteAddr16
 wdc_lib.h, 464
WDC_WriteAddr32
 wdc_lib.h, 465
WDC_WriteAddr64
 wdc_lib.h, 465
WDC_WriteAddr8
 wdc_lib.h, 465
WDC_WriteAddrBlock
 wdc_lib.h, 466
WDC_WriteAddrBlock16
 wdc_lib.h, 417
WDC_WriteAddrBlock32
 wdc_lib.h, 417
WDC_WriteAddrBlock64
 wdc_lib.h, 417
WDC_WriteAddrBlock8
 wdc_lib.h, 418
WDC_WriteMem16
 wdc_lib.h, 418
WDC_WriteMem32
 wdc_lib.h, 418
WDC_WriteMem64
 wdc_lib.h, 418
WDC_WriteMem8
 wdc_lib.h, 418
wdc_lib.h, 418
WdDevicePropertyAddress
 windrvr_usb.h, 605
WdDevicePropertyBootConfiguration
 windrvr_usb.h, 605
WdDevicePropertyBootConfigurationTranslated
 windrvr_usb.h, 605
WdDevicePropertyBusNumber
 windrvr_usb.h, 605
WdDevicePropertyBusTypeGuid
 windrvr_usb.h, 605
WdDevicePropertyClassGuid
 windrvr_usb.h, 605
WdDevicePropertyClassName
 windrvr_usb.h, 605
WdDevicePropertyCompatibleIDs
 windrvr_usb.h, 605
WdDevicePropertyDeviceDescription
 windrvr_usb.h, 605
WdDevicePropertyDriverKeyName
 windrvr_usb.h, 605
WdDevicePropertyEnumeratorName
 windrvr_usb.h, 605
WdDevicePropertyFriendlyName
 windrvr_usb.h, 605
WdDevicePropertyHardwareID
 windrvr_usb.h, 605
WdDevicePropertyInstallState
 windrvr_usb.h, 605
WdDevicePropertyLegacyBusType
 windrvr_usb.h, 605
WdDevicePropertyLocationInformation
 windrvr_usb.h, 605
WdDevicePropertyManufacturer
 windrvr_usb.h, 605
WdDevicePropertyPhysicalDeviceObjectName
 windrvr_usb.h, 605
WdDevicePropertyRemovalPolicy
 windrvr_usb.h, 605
WdDevicePropertyUINumber
 windrvr_usb.h, 605
WdFunctionLog
 wd_log.h, 391
WDS_IPC_MSG_RX, 263
 dwMsgID, 263
 dwSenderUID, 263
 qwMsgData, 263
WDS_IPC_SCAN_RESULT, 264
 dwNumProcs, 264
 proInfo, 264
WDS_IpcMulticast
 wds_lib.h, 474
WDS_IpcRegister
 wds_lib.h, 475
WDS_IpcScanProcs
 wds_lib.h, 476
WDS_IpcSubGroupMulticast
 wds_lib.h, 478

WDS_IpcUidUnicast
 wds_lib.h, 478
WDS_IpcUnRegister
 wds_lib.h, 480
WDS_IsIpcRegistered
 wds_lib.h, 481
WDS_IsSharedIntsEnabledLocally
 wds_lib.h, 482
wds_lib.h, 472, 485
 IPC_MSG_RX_HANDLER, 474
 WDS_IpcMulticast, 474
 WDS_IpcRegister, 475
 WDS_IpcScanProcs, 476
 WDS_IpcSubGroupMulticast, 478
 WDS_IpcUidUnicast, 478
 WDS_IpcUnRegister, 480
 WDS_IsIpcRegistered, 481
 WDS_IsSharedIntsEnabledLocally, 482
 WDS_SharedBufferAlloc, 482
 WDS_SharedBufferFree, 483
 WDS_SharedBufferGet, 483
 WDS_SharedBufferGetGlobalHandle, 473
 WDS_SharedIntDisableGlobal, 483
 WDS_SharedIntDisableLocal, 484
 WDS_SharedIntEnable, 484
WDS_SharedBufferAlloc
 wds_lib.h, 482
WDS_SharedBufferFree
 wds_lib.h, 483
WDS_SharedBufferGet
 wds_lib.h, 483
WDS_SharedBufferGetGlobalHandle
 wds_lib.h, 473
WDS_SharedIntDisableGlobal
 wds_lib.h, 483
WDS_SharedIntDisableLocal
 wds_lib.h, 484
WDS_SharedIntEnable
 wds_lib.h, 484
WDU_ALTERNATE_SETTING, 264
 Descriptor, 265
 pEndpointDescriptors, 265
 pPipes, 265
WDU_ATTACH_CALLBACK
 wdu_lib.h, 488
WDU_CONFIG_DESC_TYPE
 windrvr_usb.h, 602
WDU_CONFIGURATION, 265
 Descriptor, 266
 dwNumInterfaces, 266
 pInterfaces, 266
WDU_CONFIGURATION_DESCRIPTOR, 266
 bConfigurationValue, 267
 bDescriptorType, 267
 bLength, 267
 bmAttributes, 267
 bNumInterfaces, 267
 iConfiguration, 268
 MaxPower, 268
 wTotalLength, 268
WDU_DETACH_CALLBACK
 wdu_lib.h, 489
WDU_DEVICE, 268
 Descriptor, 269
 pActiveConfig, 269
 pActiveInterface, 269
 pConfigs, 269
 Pipe0, 269
WDU_DEVICE_DESC_TYPE
 windrvr_usb.h, 602
WDU_DEVICE_DESCRIPTOR, 270
 bcdDevice, 270
 bcdUSB, 270
 bDescriptorType, 271
 bDeviceClass, 271
 bDeviceProtocol, 271
 bDeviceSubClass, 271
 bLength, 271
 bMaxPacketSize0, 271
 bNumConfigurations, 271
 idProduct, 272
 idVendor, 272
 iManufacturer, 272
 iProduct, 272
 iSerialNumber, 272
WDU_DEVICE_HANDLE
 wdu_lib.h, 489
WDU_DIR
 windrvr_usb.h, 605
WDU_DIR_IN
 windrvr_usb.h, 605
WDU_DIR_IN_OUT
 windrvr_usb.h, 605
WDU_DIR_OUT
 windrvr_usb.h, 605
WDU_DRIVER_HANDLE
 wdu_lib.h, 489
WDU_ENDPOINT_ADDRESS_MASK
 windrvr_usb.h, 602
WDU_ENDPOINT_DESC_TYPE
 windrvr_usb.h, 602
WDU_ENDPOINT_DESCRIPTOR, 272
 bDescriptorType, 273
 bEndpointAddress, 273
 bInterval, 273
 bLength, 273
 bmAttributes, 274
 wMaxPacketSize, 274
WDU_ENDPOINT_DIRECTION_IN
 windrvr_usb.h, 603
WDU_ENDPOINT_DIRECTION_MASK
 windrvr_usb.h, 603
WDU_ENDPOINT_DIRECTION_OUT
 windrvr_usb.h, 603
WDU_ENDPOINT_TYPE_MASK
 windrvr_usb.h, 603

WDU_EVENT_TABLE, 274
 pfDeviceAttach, 274
 pfDeviceDetach, 274
 pfPowerChange, 275
 pUserData, 275
WDU_GET_DESCRIPTOR, 275
 bIndex, 275
 bType, 275
 dwUniqueId, 276
 pBuffer, 276
 wLanguage, 276
 wLength, 276
WDU_GET_DEVICE_DATA, 276
 dwBytes, 276
 dwOptions, 277
 dwUniqueId, 277
 pBuf, 277
WDU_GET_MAX_PACKET_SIZE
 windrvr_usb.h, 603
WDU_GetDeviceAddr
 wdi_lib.h, 490
WDU_GetDeviceInfo
 wdi_lib.h, 490
WDU_GetDeviceRegistryProperty
 wdi_lib.h, 491
WDU_GetLangIDs
 wdi_lib.h, 491
WDUGetStringDesc
 wdi_lib.h, 492
WDU_HALT_TRANSFER, 277
 dwOptions, 277
 dwPipeNum, 277
 dwUniqueId, 278
WDU_HaltTransfer
 wdi_lib.h, 493
WDU_Init
 wdi_lib.h, 493
WDU_INTERFACE, 278
 dwNumAltSettings, 278
 pActiveAltSetting, 278
 pAlternateSettings, 278
WDU_INTERFACE_DESC_TYPE
 windrvr_usb.h, 603
WDU_INTERFACE_DESCRIPTOR, 279
 bAlternateSetting, 279
 bDescriptorType, 279
 bInterfaceClass, 280
 bInterfaceNumber, 280
 bInterfaceProtocol, 280
 bInterfaceSubClass, 280
 bLength, 280
 bNumEndpoints, 280
 iInterface, 280
WDU_LANGID
 wdi_lib.h, 489
 wdi_lib.h, 486, 504
 WDU_ATTACH_CALLBACK, 488
 WDU_DETACH_CALLBACK, 489
 WDU_DEVICE_HANDLE, 489
 WDU_DRIVER_HANDLE, 489
 WDU_GetDeviceAddr, 490
 WDU_GetDeviceInfo, 490
 WDU_GetDeviceRegistryProperty, 491
 WDU_GetLangIDs, 491
 WDU_GetStringDesc, 492
 WDU_HaltTransfer, 493
 WDU_Init, 493
 WDU_LANGID, 489
 WDU_POWER_CHANGE_CALLBACK, 489
 WDU_PutDeviceInfo, 494
 WDU_ResetDevice, 494
 WDU_ResetPipe, 494
 WDU_SelectiveSuspend, 495
 WDU_SetConfig, 495
 WDU_SetInterface, 495
 WDU_STREAM_HANDLE, 490
 WDU_StreamClose, 495
 WDU_StreamFlush, 496
 WDU_StreamGetStatus, 497
 WDU_StreamOpen, 497
 WDU_StreamRead, 499
 WDU_StreamStart, 500
 WDU_StreamStop, 501
 WDU_StreamWrite, 501
 WDU_Transfer, 502
 WDU_TransferBulk, 503
 WDU_TransferDefaultPipe, 503
 WDU_TransferInterrupt, 503
 WDU_TransferIsoch, 503
 WDU_Uninit, 504
 WDU_Wakeup, 504
WDU_MATCH_TABLE, 281
 bDeviceClass, 281
 bDeviceSubClass, 281
 bInterfaceClass, 282
 bInterfaceProtocol, 282
 bInterfaceSubClass, 282
 wProductId, 282
 wVendorId, 282
WDU_PIPE_INFO, 282
 direction, 283
 dwInterval, 283
 dwMaximumPacketSize, 283
 dwNumber, 283
 type, 284
WDU_POWER_CHANGE_CALLBACK
 wdi_lib.h, 489
WDU_PutDeviceInfo
 wdi_lib.h, 494
WDU_REGISTER_DEVICES_HANDLE
 windrvr_usb.h, 603
WDU_RESET_DEVICE, 284
 dwOptions, 284
 dwUniqueId, 284
WDU_RESET_PIPE, 285
 dwOptions, 285

dwPipeNum, 285
dwUniqueId, 285
WDU_ResetDevice
 wdu_lib.h, 494
WDU_ResetPipe
 wdu_lib.h, 494
WDU_SELECTIVE_SUSPEND, 285
 dwOptions, 286
 dwUniqueId, 286
WDU_SELECTIVE_SUSPEND_CANCEL
 windrvr_usb.h, 606
WDU_SELECTIVE_SUSPEND_OPTIONS
 windrvr_usb.h, 605
WDU_SELECTIVE_SUSPEND_SUBMIT
 windrvr_usb.h, 605
WDU_SelectiveSuspend
 wdu_lib.h, 495
WDU_SET_INTERFACE, 286
 dwAlternateSetting, 286
 dwInterfaceNum, 286
 dwOptions, 286
 dwUniqueId, 287
WDU_SetConfig
 wdu_lib.h, 495
WDU_SetInterface
 wdu_lib.h, 495
WDU_STREAM, 287
 dwBufferSize, 287
 dwOptions, 287
 dwPipeNum, 287
 dwReserved, 288
 dwRxSize, 288
 dwRxTxTimeout, 288
 dwUniqueId, 288
 fBlocking, 288
WDU_STREAM_HANDLE
 wdu_lib.h, 490
WDU_STREAM_STATUS, 288
 dwBytesInBuffer, 289
 dwLastError, 289
 dwOptions, 289
 dwReserved, 289
 dwUniqueId, 289
 flsRunning, 289
WDU_StreamClose
 wdu_lib.h, 495
WDU_StreamFlush
 wdu_lib.h, 496
WDU_StreamGetStatus
 wdu_lib.h, 497
WDU_StreamOpen
 wdu_lib.h, 497
WDU_StreamRead
 wdu_lib.h, 499
WDU_StreamStart
 wdu_lib.h, 500
WDU_StreamStop
 wdu_lib.h, 501
WDU_StreamWrite
 wdu_lib.h, 501
WDU_STRING_DESC_STRING
 windrvr_usb.h, 603
WDU_TRANSFER, 289
 dwBufferSize, 290
 dwBytesTransferred, 290
 dwOptions, 290
 dwPipeNum, 290
 dwTimeout, 291
 dwUniqueId, 291
 fRead, 291
 pBuffer, 291
 SetupPacket, 291
WDU_Transfer
 wdu_lib.h, 502
WDU_TransferBulk
 wdu_lib.h, 503
WDU_TransferDefaultPipe
 wdu_lib.h, 503
WDU_TransferInterrupt
 wdu_lib.h, 503
WDU_TransferIsoch
 wdu_lib.h, 503
WDU_Uninit
 wdu_lib.h, 504
WDU_WAKEUP, 291
 dwOptions, 292
 dwUniqueId, 292
WDU_Wakeup
 wdu_lib.h, 504
WDU_WAKEUP_DISABLE
 windrvr_usb.h, 606
WDU_WAKEUP_ENABLE
 windrvr_usb.h, 606
WDU_WAKEUP_OPTIONS
 windrvr_usb.h, 606
WIN32
 windrvr.h, 557
WINAPI
 windrvr.h, 557
windrvr.h, 506, 576
 In, 516
 Inout, 516
 Out, 516
 Outptr, 516
 __ALIGN_DOWN, 516
 __ALIGN_UP, 516
 __FUNCTION__, 516
BYTE, 557
BZERO, 517
check_secureBoot_enabled, 573
CMD_END, 573
CMD_MASK, 573
CMD_NONE, 573
CTL_CODE, 517
D_ERROR, 564
D_INFO, 564

D_OFF, 564
D_TRACE, 564
D_WARN, 564
DEBUG_CLEAR_BUFFER, 564
DEBUG_CLOCK_RESET, 564
DEBUG_COMMAND, 563
DEBUG_DUMP_CLOCK_OFF, 564
DEBUG_DUMP_CLOCK_ON, 564
DEBUG_DUMP_SEC_OFF, 564
DEBUG_DUMP_SEC_ON, 564
DEBUG_LEVEL, 564
DEBUG_SECTION, 564
DEBUG_SET_BUFFER, 563
DEBUG_SET_FILTER, 563
DEBUG_STATUS, 563
DEBUG_USER_BUF_LEN, 517
DLLCALLCONV, 517
DMA_ADDR, 557
DMA_ADDRESS_WIDTH_MASK, 517
DMA_ALLOW_64BIT_ADDRESS, 565
DMA_ALLOW_CACHE, 565
DMA_ALLOW_NO_HCARD, 565
DMA_BIT_MASK, 517
DMA_DIRECTION_MASK, 517
DMA_DISABLE_MERGE_ADJACENT_PAGES, 566
DMA_FROM_DEVICE, 565
DMA_GET_EXISTING_BUF, 565
DMA_GET_PREALLOCATED_BUFFERS_ONLY, 566
DMA_GPUDIRECT, 566
DMA_KBUF_ALLOC_SPECIFY_ADDRESS_WIDTH, 566
DMA_KBUF_BELOW_16M, 565
DMA_KERNEL_BUFFER_ALLOC, 565
DMA_KERNEL_ONLY_MAP, 565
DMA_LARGE_BUFFER, 565
DMA_OPTIONS_ADDRESS_WIDTH_SHIFT, 517
DMA_OPTIONS_ALL, 518
DMA_READ_FROM_DEVICE, 518
DMA_RESERVED_MEM, 565
DMA_TO_DEVICE, 565
DMA_TO_FROM_DEVICE, 565
DMA_TRANSACTION, 566
DMA_TRANSACTION_CALLBACK, 557
DMA_WRITE_TO_DEVICE, 518
FILE_ANY_ACCESS, 518
FILE_READ_ACCESS, 518
FILE_WRITE_ACCESS, 518
FUNC_MASK, 518
get_os_type, 574
INSTALLATION_TYPE_NOT_DETECT_TEXT, 518
INTERRUPT_CE_INT_ID, 562
INTERRUPT_CMD_COPY, 562
INTERRUPT_CMD_RETURN_VALUE, 562
INTERRUPT_DONT_GET_MSI_MESSAGE, 562
INTERRUPT_INTERRUPTED, 571
INTERRUPT_LATCHED, 562
INTERRUPT_LEVEL_SENSITIVE, 562
INTERRUPT_MESSAGE, 562
INTERRUPT_MESSAGE_X, 562
INTERRUPT_RECEIVED, 571
INTERRUPT_STOPPED, 571
INVALID_HANDLE_VALUE, 519
IOCTL_WD_CARD_CLEANUP_SETUP, 519
IOCTL_WD_CARD_REGISTER, 519
IOCTL_WD_CARD_UNREGISTER, 519
IOCTL_WD_DEBUG, 519
IOCTL_WD_DEBUG_ADD, 519
IOCTL_WD_DEBUG_DUMP, 519
IOCTL_WD_DMA_LOCK, 519
IOCTL_WD_DMA_SYNC_CPU, 519
IOCTL_WD_DMA_SYNC_IO, 519
IOCTL_WD_DMA_TRANSACTION_EXECUTE, 520
IOCTL_WD_DMA_TRANSACTION_INIT, 520
IOCTL_WD_DMA_TRANSACTION_RELEASE, 520
IOCTL_WD_DMA_TRANSFER_COMPLETED_AND_CHECK, 520
IOCTL_WD_DMA_UNLOCK, 520
IOCTL_WD_EVENT_PULL, 520
IOCTL_WD_EVENT_REGISTER, 520
IOCTL_WD_EVENT_SEND, 520
IOCTL_WD_EVENT_UNREGISTER, 520
IOCTL_WD_GET_DEVICE_PROPERTY, 520
IOCTL_WD_INT_COUNT, 521
IOCTL_WD_INT_DISABLE, 521
IOCTL_WD_INT_ENABLE, 521
IOCTL_WD_INT_WAIT, 521
IOCTL_WD_IPC_REGISTER, 521
IOCTL_WD_IPC_SCAN_PROCS, 521
IOCTL_WD_IPC_SEND, 521
IOCTL_WD_IPC_SHARED_INT_DISABLE, 521
IOCTL_WD_IPC_SHARED_INT_ENABLE, 521
IOCTL_WD_IPC_UNREGISTER, 521
IOCTL_WD_KERNEL_BUF_LOCK, 522
IOCTL_WD_KERNEL_BUF_UNLOCK, 522
IOCTL_WD_KERNEL_PLUGIN_CALL, 522
IOCTL_WD_KERNEL_PLUGIN_CLOSE, 522
IOCTL_WD_KERNEL_PLUGIN_OPEN, 522
IOCTL_WD_LICENSE, 522
IOCTL_WD_MULTI_TRANSFER, 522
IOCTL_WD_PCI_CONFIG_DUMP, 522
IOCTL_WD_PCI_GET_CARD_INFO, 522
IOCTL_WD_PCI_SCAN_CAPS, 522
IOCTL_WD_PCI_SCAN_CARDS, 523
IOCTL_WD_PCI_SRIOV_DISABLE, 523
IOCTL_WD_PCI_SRIOV_ENABLE, 523
IOCTL_WD_PCI_SRIOV_GET_NUMVFS, 523
IOCTL_WD_SLEEP, 523
IOCTL_WD_TRANSFER, 523
IOCTL_WD_USAGE, 523
IOCTL_WD_VERSION, 523
IOCTL_WDU_GET_DEVICE_DATA, 523
IOCTL_WDU_HALT_TRANSFER, 523

IOCTL_WDU_RESET_DEVICE, 524
IOCTL_WDU_RESET_PIPE, 524
IOCTL_WDU_SELECTIVE_SUSPEND, 524
IOCTL_WDU_SET_INTERFACE, 524
IOCTL_WDU_STREAM_CLOSE, 524
IOCTL_WDU_STREAM_FLUSH, 524
IOCTL_WDU_STREAM_GET_STATUS, 524
IOCTL_WDU_STREAM_OPEN, 524
IOCTL_WDU_STREAM_START, 524
IOCTL_WDU_STREAM_STOP, 524
IOCTL_WDU_TRANSFER, 525
IOCTL_WDU_WAKEUP, 525
ITEM_BUS, 565
ITEM_INTERRUPT, 565
ITEM_IO, 565
ITEM_MEMORY, 565
ITEM_NONE, 565
ITEM_TYPE, 564
KER_BUF_ALLOC_CACHED, 572
KER_BUF_ALLOC_CONTIG, 572
KER_BUF_ALLOC_NON_CONTIG, 572
KER_BUF_GET_EXISTING_BUF, 572
KERNEL_DEBUGGER_OFF, 564
KERNEL_DEBUGGER_ON, 564
KPRI, 525
KPTR, 557
MAX, 525
METHOD_BUFFERED, 525
METHOD_IN_DIRECT, 525
METHOD_NEITHER, 525
METHOD_OUT_DIRECT, 525
MIN, 525
OS_CAN_NOT_DETECT_TEXT, 526
PAD_TO_64, 526
PCI_ACCESS_ERROR, 565
PCI_ACCESS_OK, 565
PCI_ACCESS_RESULT, 565
PCI_BAD_BUS, 565
PCI_BAD_SLOT, 565
PHYS_ADDR, 557
PRI64, 526
REGKEY_BUFSIZE, 526
RM_BYTE, 573
RM_DWORD, 573
RM_QWORD, 573
RM_SBYTE, 573
RM_SDWORD, 573
RM_SQWORD, 573
RM_SWORD, 573
RM_WORD, 573
RP_BYTE, 573
RP_DWORD, 573
RP_QWORD, 573
RP_SBYTE, 573
RP_SDWORD, 573
RP_SQWORD, 573
RP_SWORD, 573
RP_WORD, 573
S_ALL, 564
S_CARD_REG, 564
S_DMA, 564
S_EVENT, 564
S_INT, 564
S_IO, 564
S_IPC, 564
S_KER_BUF, 564
S_KER_DRV, 564
S_KER_PLUG, 564
S_LICENSE, 564
S_MEM, 564
S_MISC, 564
S_PCI, 564
S_PNP, 564
S_USB, 564
SAFE_STRING, 526
SIZE_OF_WD_DMA, 526
SIZE_OF_WD_EVENT, 526
SLEEP_BUSY, 561
SLEEP_NON_BUSY, 561
strcmp, 526
UINT32, 557
UINT64, 557
UNUSED_VAR, 527
UPRI, 527
UPTR, 558
va_copy, 527
WD_ACCEPT_CONTROL, 571
WD_ACKNOWLEDGE, 571
WD_ACTIONS_ALL, 527
WD_ACTIONS_POWER, 527
WD_BUS_EISA, 562
WD_BUS_ISA, 562
WD_BUS_PCI, 562
WD_BUS_TYPE, 558
WD_BUS_UNKNOWN, 562
WD_BUS_USB, 562
WD_BUS_V30, 558
WD_CANT_OBTAIN_PDO, 567
WD_CARD_ITEMS, 562
WD_CARD_REGISTER_V118, 558
WD_CARD_V118, 558
WD_CardCleanupSetup, 527
WD_CardRegister, 528
WD_CardUnregister, 529
WD_Close, 529
WD_CloseLocal, 529
WD_CPU_SPEC, 529
WD_CTL_CODE, 529
WD_CTL_DECODE_FUNC, 530
WD_CTL_DECODE_TYPE, 530
WD_CTL_IS_64BIT_AWARE, 530
WD_DATA_MISMATCH, 566
WD_DATA_MODEL, 530
WD_Debug, 530
WD_DEBUG_ADD_V503, 558
WD_DEBUG_DUMP_V40, 558

WD_DEBUG_V40, 558
WD_DebugAdd, 530
WD_DebugDump, 531
WD_DEFAULT_DRIVER_NAME, 531
WD_DEFAULT_DRIVER_NAME_BASE, 531
WD_DEVICE_NOT_FOUND, 567
WD_DEVICE_PCI, 571
WD_DEVICE_USB, 571
WD_DMA_OPTIONS, 565
WD_DMA_PAGE_V80, 558
WD_DMA_PAGES, 560
WD_DMA_V80, 558
WD_DMALock, 531
WD_DMASyncCpu, 533
WD_DMASynclo, 533
WD_DMATransactionExecute, 534
WD_DMATransactionInit, 534
WD_DMATransactionRelease, 535
WD_DMATransactionUninit, 535
WD_DMATransferCompletedAndCheck, 535
WD_DMAUnlock, 536
WD_DRIVER_NAME, 536
WD_DRIVER_NAME_PREFIX, 536
WD_DriverName, 574
WD_ERROR_CODES, 566
WD_EVENT_ACTION, 570
WD_EVENT_OPTION, 571
WD_EVENT_TYPE, 558
WD_EVENT_TYPE_IPC, 561
WD_EVENT_TYPE_PCI, 561
WD_EVENT_TYPE_UNKNOWN, 561
WD_EVENT_TYPE_USB, 561
WD_EVENT_V121, 559
WD_EventPull, 536
WD_EventRegister, 536
WD_EventSend, 536
WD_EventUnregister, 537
WD_FAILED_ENABLING_INTERRUPT, 567
WD_FAILED_KERNEL_MAPPING, 567
WD_FAILED_USER_MAPPING, 567
WD_FORCE_CLEANUP, 563
WD_FUNCTION, 537
WD_GET_DEVICE_PROPERTY_OPTION, 571
WD_GetDeviceProperty, 537
WD_INCORRECT_VERSION, 568
WD_INSERT, 570
WD_INSUFFICIENT_RESOURCES, 566
WD_IntCount, 537
WD_IntDisable, 537
WD_IntEnable, 538
WD_INTERRUPT_NOT_ENABLED, 567
WD_INTERRUPT_V91, 559
WD_INTERRUPT_WAIT_RESULT, 571
WD_IntWait, 539
WD_INVALID_32BIT_APP, 568
WD_INVALID_HANDLE, 566
WD_INVALID_IOCTL, 568
WD_INVALID_PARAMETER, 568
WD_INVALID_PIPE_NUMBER, 566
WD_IPC_ALL_MSG, 539
WD_IPC_MAX_PROCS, 562
WD_IPC_MULTICAST, 563
WD_IPC_MULTICAST_MSG, 571
WD_IPC_PROCESS_V121, 559
WD_IPC_REGISTER_V121, 559
WD_IPC_SCAN_PROCS_V121, 559
WD_IPC_SEND_V121, 559
WD_IPC_SUBGROUP_MULTICAST, 563
WD_IPC_UID_UNICAST, 563
WD_IPC_UNICAST_MSG, 571
WD_IpcRegister, 539
WD_IpcScanProcs, 540
WD_IpcSend, 540
WD_IpcUnRegister, 540
WD_IRP_CANCELED, 567
WD_ITEM_MEM_ALLOW_CACHE, 572
WD_ITEM_MEM_DO_NOT_MAP_KERNEL, 572
WD_ITEM_MEM_OPTIONS, 571
WD_ITEM_MEM_USER_MAP, 572
WD_ITEMS_V118, 559
WD_KER_BUF_OPTION, 572
WD_KERNEL_BUFFER_V121, 559
WD_KERNEL_PLUGIN_CALL_V40, 559
WD_KERNEL_PLUGIN_V40, 559
WD_KernelBufLock, 541
WD_KernelBufUnlock, 541
WD_KernelPlugInCall, 543
WD_KernelPlugInClose, 543
WD_KernelPlugInOpen, 545
WD_KERPLUG_FAILURE, 567
WD_License, 545
WD_LICENSE_LENGTH, 546
WD_LICENSE_V122, 559
WD_MATCH_EXCLUDE, 561
WD_MAX_DRIVER_NAME_LENGTH, 546
WD_MAX_KP_NAME_LENGTH, 546
WD_MORE_PROCESSING_REQUIRED, 568
WD_MultiTransfer, 546
WD_NO_DEVICE_OBJECT, 568
WD_NO_EVENTS, 568
WD_NO_LICENSE, 566
WD_NO_RESOURCES_ON_DEVICE, 568
WD_NOT_IMPLEMENTED, 567
WD_OBSOLETE, 570
WD_Open, 547
WD_OpenLocal, 547
WD_OpenStreamLocal, 547
WD_OPERATION_ALREADY_DONE, 567
WD_OPERATION_FAILED, 568
WD_PCI_CAP_ID_ALL, 561
WD_PCI_CARD_INFO_V118, 559
WD_PCI_CARDS, 563
WD_PCI_CONFIG_DUMP_V30, 560
WD_PCI_MAX_CAPS, 563
WD_PCI_SCAN_BY_TOPOLOGY, 572
WD_PCI_SCAN_CAPS_BASIC, 572

WD_PCI_SCAN_CAPS_EXTENDED, 572
WD_PCI_SCAN_CAPS_OPTIONS, 572
WD_PCI_SCAN_CAPS_V118, 560
WD_PCI_SCAN_CARDS_V124, 560
WD_PCI_SCAN_DEFAULT, 572
WD_PCI_SCAN_INCLUDE_DOMAINS, 572
WD_PCI_SCAN_OPTIONS, 572
WD_PCI_SCAN_REGISTERED, 572
WD_PCI_SRIOV_V122, 560
WD_PciConfigDump, 547
WD_PciGetCardInfo, 548
WD_PciScanCaps, 549
WD_PciScanCards, 549
WD_PciSriovDisable, 550
WD_PciSriovEnable, 550
WD_PciSriovGetNumVFs, 551
WD_POWER_CHANGED_D0, 570
WD_POWER_CHANGED_D1, 570
WD_POWER_CHANGED_D2, 570
WD_POWER_CHANGED_D3, 570
WD_POWER_SYSTEM_HIBERNATE, 571
WD_POWER_SYSTEM_SHUTDOWN, 571
WD_POWER_SYSTEM_SLEEPING1, 570
WD_POWER_SYSTEM_SLEEPING2, 571
WD_POWER_SYSTEM_SLEEPING3, 571
WD_POWER_SYSTEM_WORKING, 570
WD_PROCESS_NAME_LENGTH, 551
WD_PROD_NAME, 551
WD_READ_WRITE_CONFLICT, 566
WD_REMOVE, 570
WD_RESOURCE_OVERLAP, 567
WD_SET_CONFIGURATION_FAILED, 567
WD_SharedIntDisable, 551
WD_SharedIntEnable, 551
WD_Sleep, 552
WD_SLEEP_V40, 560
WD_STATUS_INVALID_WD_HANDLE, 566
WD_STATUS_SUCCESS, 566
WD_StreamClose, 552
WD_StreamOpen, 552
WD_SYSTEM_INTERNAL_ERROR, 566
WD_TIME_OUT_EXPIRED, 567
WD_TOO_MANY_HANDLES, 568
WD_Transfer, 553
WD_TRANSFER_CMD, 572
WD_TRANSFER_V61, 560
WD_TRY AGAIN, 568
WD_TYPE, 553
WD_UGetDeviceData, 553
WD_UHaltTransfer, 553
WD_UNKNOWN_PIPE_TYPE, 566
WD_UResetDevice, 553
WD_UResetPipe, 554
WD_Usage, 554
WD_USB_CYCLE_PORT, 561
WD_USB_DESCRIPTOR_ERROR, 567
WD_USB_HARD_RESET, 561
WD_USBD_STATUS_BABBLE_DETECTED, 569
WD_USBD_STATUS_BAD_START_FRAME, 569
WD_USBD_STATUS_BTSTUFF, 569
WD_USBD_STATUS_BUFFER_OVERRUN, 569
WD_USBD_STATUS_BUFFER_TOO_SMALL, 570
WD_USBD_STATUS_BUFFER_UNDERRUN, 569
WD_USBD_STATUS_CANCELED, 569
WD_USBD_STATUS_CRC, 569
WD_USBD_STATUS_DATA_BUFFER_ERROR, 569
WD_USBD_STATUS_DATA_OVERRUN, 569
WD_USBD_STATUS_DATA_TOGGLE_MISMATCH, 569
WD_USBD_STATUS_DATA_UNDERRUN, 569
WD_USBD_STATUS_DEV_NOT_RESPONDING, 569
WD_USBD_STATUS_DEVICE_GONE, 570
WD_USBD_STATUS_ENDPOINT_HALTED, 569
WD_USBD_STATUS_ERROR, 568
WD_USBD_STATUS_ERROR_BUSY, 569
WD_USBD_STATUS_ERROR_SHORT_TRANSFER, 569
WD_USBD_STATUS_FIFO, 569
WD_USBD_STATUS_FRAME_CONTROL_NOT OWNED, 570
WD_USBD_STATUS_FRAME_CONTROL_OWNED, 570
WD_USBD_STATUS_HALTED, 568
WD_USBD_STATUS_INAVLID_CONFIGURATION_DESCRIPTOR, 570
WD_USBD_STATUS_INAVLID_PIPE_FLAGS, 570
WD_USBD_STATUS_INSUFFICIENT_RESOURCES, 570
WD_USBD_STATUS_INTERFACE_NOT_FOUND, 570
WD_USBD_STATUS_INTERNAL_HC_ERROR, 569
WD_USBD_STATUS_INVALID_PARAMETER, 569
WD_USBD_STATUS_INVALID_PIPE_HANDLE, 569
WD_USBD_STATUS_INVALID_URB_FUNCTION, 569
WD_USBD_STATUS_ISO_NA_LATE_USBPORT, 570
WD_USBD_STATUS_ISO_NOT_ACCESSED_BY_HW, 570
WD_USBD_STATUS_ISO_NOT_ACCESSED_LATE, 570
WD_USBD_STATUS_ISO_TD_ERROR, 570
WD_USBD_STATUS_ISOCH_REQUEST FAILED, 570
WD_USBD_STATUS_NO_BANDWIDTH, 569
WD_USBD_STATUS_NO_MEMORY, 569
WD_USBD_STATUS_NOT_ACCESSED, 569
WD_USBD_STATUS_NOT_SUPPORTED, 570
WD_USBD_STATUS_PENDING, 568
WD_USBD_STATUS_PID_CHECK_FAILURE, 569
WD_USBD_STATUS_REQUEST FAILED, 569
WD_USBD_STATUS_RESERVED1, 569

WD_USBD_STATUS_RESERVED2, 569
WD_USBD_STATUS_SET_CONFIG_FAILED, 570
WD_USBD_STATUS_STALL_PID, 569
WD_USBD_STATUS_STATUS_NOT_MAPPED, 570
WD_USBD_STATUS_SUCCESS, 568
WD_USBD_STATUS_TIMEOUT, 570
WD_USBD_STATUS_UNEXPECTED_PID, 569
WD_USBD_STATUS_XACT_ERROR, 569
WD_USelectiveSuspend, 554
WD_USetInterface, 554
WD_UStreamClose, 554
WD_UStreamFlush, 554
WD_UStreamGetStatus, 555
WD_UStreamOpen, 555
WD_UStreamRead, 555
WD_UStreamStart, 555
WD_UStreamStop, 555
WD_UStreamWriter, 555
WD_UTransfer, 556
WD_UWakeup, 556
WD_VER_STR, 556
WD_Version, 556
WD_VERSION_STR_LENGTH, 557
WD_VERSION_V30, 560
WD_WINDRIVER_NOT_FOUND, 568
WD_WINDRIVER_STATUS_ERROR, 566
WD_WRONG_UNIQUE_ID, 567
WD_ZERO_PACKET_SIZE, 566
WIN32, 557
WINAPI, 557
WM_BYTE, 573
WM_DWORD, 573
WM_QWORD, 573
WM_SBYTE, 573
WM_SDWORD, 573
WM_SQWORD, 573
WM_SWORD, 573
WM_WORD, 573
WORD, 560
WP_BYTE, 573
WP_DWORD, 573
WP_QWORD, 573
WP_SBYTE, 573
WP_SDWORD, 573
WP_SQWORD, 573
WP_SWORD, 573
WP_WORD, 573
windrvr_32bit.h, 594
 ptr32, 594
windrvr_events.h, 597, 598
 event_handle_t, 597
 EVENT_HANDLER, 597
 EventAlloc, 597
 EventDup, 598
 EventFree, 598
 EventRegister, 598
 EventUnregister, 598
PciEventCreate, 598
UsbEventCreate, 598
windrvr_int_thread.h, 599, 600
 INT_HANDLER, 599
 INT_HANDLER_FUNC, 599
 InterruptDisable, 599
 InterruptEnable, 600
windrvr_usb.h, 600, 606
 PAD_TO_64, 602
 PAD_TO_64_PTR_ARR, 602
 PIPE_TYPE_BULK, 604
 PIPE_TYPE_CONTROL, 604
 PIPE_TYPE_INTERRUPT, 604
 PIPE_TYPE_ISOCHRONOUS, 604
 USB_ABORT_PIPE, 604
 USB_BULK_INT_URB_SIZE_OVERRIDE_128K, 604
 USB_DIR, 604
 USB_DIR_IN, 604
 USB_DIR_IN_OUT, 604
 USB_DIR_OUT, 604
 USB_FULL_TRANSFER, 604
 USB_ISOCH_ASAP, 604
 USB_ISOCH_FULL_PACKETS_ONLY, 604
 USB_ISOCH_NOASAP, 604
 USB_ISOCH_RESET, 604
 USB_PIPE_TYPE, 604
 USB_SHORT_TRANSFER, 604
 USB_STREAM_OVERWRITE_BUFFER_WHEN_FULL, 604
 USB_TRANSFER_HALT, 604
 WD_DEVICE_REGISTRY_PROPERTY, 604
 WD_USB_MAX_ALT_SETTINGS, 602
 WD_USB_MAX_ENDPOINTS, 602
 WD_USB_MAX_INTERFACES, 602
 WD_USB_MAX_PIPE_NUMBER, 602
 WdDevicePropertyAddress, 605
 WdDevicePropertyBootConfiguration, 605
 WdDevicePropertyBootConfigurationTranslated, 605
 WdDevicePropertyBusNumber, 605
 WdDevicePropertyBusTypeGuid, 605
 WdDevicePropertyClassGuid, 605
 WdDevicePropertyClassName, 605
 WdDevicePropertyCompatibleIDs, 605
 WdDevicePropertyDeviceDescription, 605
 WdDevicePropertyDriverKeyName, 605
 WdDevicePropertyEnumeratorName, 605
 WdDevicePropertyFriendlyName, 605
 WdDevicePropertyHardwareID, 605
 WdDevicePropertyInstallState, 605
 WdDevicePropertyLegacyBusType, 605
 WdDevicePropertyLocationInformation, 605
 WdDevicePropertyManufacturer, 605
 WdDevicePropertyPhysicalDeviceObjectName, 605
 WdDevicePropertyRemovalPolicy, 605
 WdDevicePropertyUINumber, 605

WDU_CONFIG_DESC_TYPE, 602
WDU_DEVICE_DESC_TYPE, 602
WDU_DIR, 605
WDU_DIR_IN, 605
WDU_DIR_IN_OUT, 605
WDU_DIR_OUT, 605
WDU_ENDPOINT_ADDRESS_MASK, 602
WDU_ENDPOINT_DESC_TYPE, 602
WDU_ENDPOINT_DIRECTION_IN, 603
WDU_ENDPOINT_DIRECTION_MASK, 603
WDU_ENDPOINT_DIRECTION_OUT, 603
WDU_ENDPOINT_TYPE_MASK, 603
WDU_GET_MAX_PACKET_SIZE, 603
WDU_INTERFACE_DESC_TYPE, 603
WDU_REGISTER_DEVICES_HANDLE, 603
WDU_SELECTIVE_SUSPEND_CANCEL, 606
WDU_SELECTIVE_SUSPEND_OPTIONS, 605
WDU_SELECTIVE_SUSPEND_SUBMIT, 605
WDU_STRING_DESC_STRING, 603
WDU_WAKEUP_DISABLE, 606
WDU_WAKEUP_ENABLE, 606
WDU_WAKEUP_OPTIONS, 606
wLanguage
 WDU_GET_DESCRIPTOR, 276
wLength
 WDU_GET_DESCRIPTOR, 276
WM_BYTE
 windrvr.h, 573
WM_DWORD
 windrvr.h, 573
WM_QWORD
 windrvr.h, 573
WM_SBYTE
 windrvr.h, 573
WM_SDWORD
 windrvr.h, 573
WM_SQWORD
 windrvr.h, 573
WM_SWORD
 windrvr.h, 573
WM_WORD
 windrvr.h, 573
wMaxPacketSize
 WDU_ENDPOINT_DESCRIPTOR, 274
WORD
 windrvr.h, 560
Word
 WD_TRANSFER, 254
WP_BYTE
 windrvr.h, 573
WP_DWORD
 windrvr.h, 573
WP_QWORD
 windrvr.h, 573
WP_SBYTE
 windrvr.h, 573
WP_SDWORD
 windrvr.h, 573
WP_SQWORD
 windrvr.h, 573
WP_SWORD
 windrvr.h, 573
WP_WORD
 windrvr.h, 573
wProductId
 WDU_MATCH_TABLE, 282
wTotalLength
 WDU_CONFIGURATION_DESCRIPTOR, 268
wVendorId
 WDU_MATCH_TABLE, 282