WinDriver™ PCI/ISA Low-Level API Reference

Jungo Connectivity Ltd.

Version 14.1.1

WinDriver™ PCI/ISA Low-Level API Reference

Table of Contents

List of Figures

Chapter 1
Overview

The present guide provides an overview of the low-level `WD_xxx` PCI/ISA WinDriver APIs. While you are still free to use these APIs, it is recommended, instead, to use the high-level **WDC** library APIs, which provide convenient wrappers to the low-level APIs. The **WDC** APIs are described in detail in the **WinDriver PCI Manual** (**wdpci_man.pdf**).

This function reference is C oriented. The WinDriver C# APIs have been implemented as closely as possible to the C APIs, therefore .NET programmers can also use this reference to better understand the WinDriver APIs for their selected development language. For the exact API implementation and usage examples for your selected language, refer to the WinDriver .NET source code.

Chapter 2
WD_xxx PCI/ISA Functions

This chapter describes the basic WD_xxx() PCI/ISA WinDriver functions.

2.1. API Calling Sequence — PCI/ISA

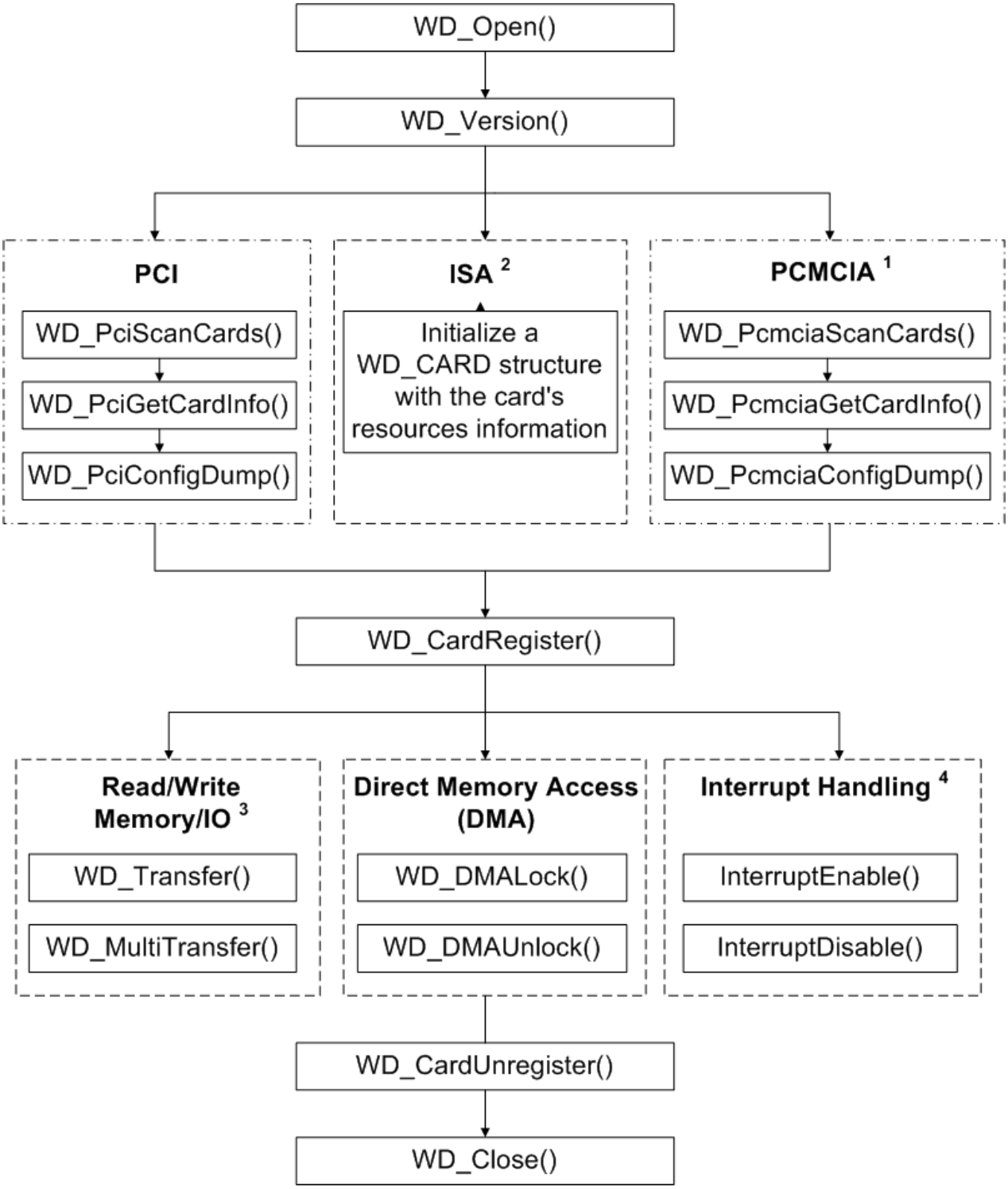
Figure 2.1 demonstrates typical calling sequences using the basic WinDriver WD_xxx() PCI/ISA functions.

1. Memory addresses can be accessed directly, using the user-mode mapping of the address returned by WD_CardRegister() [2.6] (or the kernel mapping, when accessing memory from the Kernel PlugIn). Direct memory access is more efficient than using WD_Transfer() [2.9].

2. It is possible (although not recommended) to replace the use of the high-level InterruptEnable() convenience function [2.17] with the low-level WD_IntEnable() [3.2], WD_IntWait() [3.3] and WD_IntCount() [3.4] functions, and replace the call to InterruptDisable() [2.18] with a call to the low-level WD_IntDisable() function [3.5].
   For more information on the low-level WinDriver interrupt handling API, refer to Chapter 3 of the manual.

3. WinDriver's general-use APIs, such as WD_DebugAdd() [5.6] or WD_Sleep() [5.8], can be called anywhere between the calls to WD_Open() and WD_Close(). For more details, refer to Chapter 5.

Figure 2.1. WinDriver PCI/ISA Calling Sequence



2.2. WD_PciScanCards()

Purpose

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.

Prototype

```
DWORD WD_PciScanCards(
    HANDLE hWD,
    WD_PCI_SCAN_CARDS *pPciScan);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pPciScan	WD_PCI_SCAN_CARDS*	
• searchId	WD_PCI_ID	
* dwVendorId	DWORD	Input
* dwDeviceId	DWORD	Input
• dwCards	DWORD	Output
• cardId	WD_PCI_ID[WD_PCI_CARDS]	
* dwVendorId	DWORD	Output
* dwDeviceId	DWORD	Output
• cardSlot	WD_PCI_SLOT[WD_PCI_CARDS]	
* dwBus	DWORD	Output
* dwSlot	DWORD	Output
* dwFunction	DWORD	Output
• dwOptions	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pPciScan	Pointer to a PCI bus scan information structure:
• searchId	PCI card ID information structure:
* dwVendorId	The vendor ID of the PCI cards to detect. If 0, the function will search for all PCI card vendor IDs.
* dwDeviceId	The card ID of the PCI cards to detect. If 0, the function will search for all PCI card IDs. If both **`dwVendorId`** and **`dwDeviceId`** are set to 0 the function will return information regarding all connected PCI cards.
• dwCards	Number of cards detected for the specified search criteria set in the **`searchId`** field
• cardId	Array of PCI card ID information structures for the detected PCI cards that match the search criteria set in the **`searchId`** field:
* dId.dwVendorId	Vendor ID
* dId.dwDeviceId	Card ID
• cardSlot	Array of PCI slot information structures for the detected PCI cards that match the search criteria set in the **`searchId`** field:
* dwBus	Bus number (0 based)
* dwSlot	Slot number (0 based)
* dwFunction	Function number (0 based)
• dwOptions	PCI bus scan options. Can be set to any of the **`WD_PCI_SCAN_OPTIONS`** flags: • **`WD_PCI_SCAN_DEFAULT`** — Scan all PCI buses and slots. This is the default scan option. • **`WD_PCI_SCAN_BY_TOPOLOGY`** — Scan the PCI bus by topology. • **`WD_PCI_SCAN_REGISTERED`** — Scan all PCI buses and slots but only look for devices that have been registered to work with WinDriver.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
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");
```

2.3. WD_PciScanCaps()

Purpose

Scans the specified PCI capabilities group of the given PCI slot for the specified capability (or for all capabilities).

Prototype

```
DWORD WD_PciScanCaps(
    HANDLE hWD,
    WD_PCI_SCAN_CAPS *pPciScanCaps);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pPciScanCaps	WD_PCI_SCAN_CAPS*	
• pciSlot	WD_PCI_SLOT	
* dwBus	DWORD	Input
* dwSlot	DWORD	Input
* dwFunction	DWORD	Input
• dwCapId	DWORD	Input
• dwOptions	DWORD	Input
• dwNumCaps	DWORD	Output
• pciCaps	WD_PCI_CAP[WD_PCI_MAX_CAPS]	
* dwCapId	DWORD	Output
* dwCapOffset	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pPciScanCaps	Pointer to a PCI capabilities scan structure:
• pciSlot	PCI slot information structure
* dwBus	Bus number (0 based)
* dwSlot	Slot number (0 based)
* dwFunction	Function number (0 based)
• dwCapId	ID of the PCI capability for which to search, or `WD_PCI_CAP_ID_ALL` to search for all PCI capabilities
• dwOptions	PCI capabilities scan options. Can be set to any of the **`WD_PCI_SCAN_CAPS_OPTIONS`** flags: • **`WD_PCI_SCAN_CAPS_BASIC`** — Scan the basic PCI capabilities. This is the default scan option. • **`WD_PCI_SCAN_CAPS_EXTENDED`** — Scan the extended (PCI Express) PCI capabilities.
• dwNumCaps	Number of PCI capabilities that match the search criteria set in the **`dwCapId`** and **`dwOptions`** fields
• pciCaps	Array of PCI capability structures for PCI capabilities that match the search criteria set in the **`dwCapId`** and **`dwOptions`** fields
* dwCapId	PCI capability ID
* dwCapOffset	PCI capability register offset

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
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);
}
```

2.4. WD_PciGetCardInfo()

Purpose

Retrieves PCI device's resource information (i.e., Memory ranges, I/O ranges, Interrupt lines).

Prototype

```
DWORD WD_PciGetCardInfo(
    HANDLE hWD,
    WD_PCI_CARD_INFO *pPciCard);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pPciCard	WD_PCI_CARD_INFO*	Input
• pciSlot	WD_PCI_SLOT	Input
* dwBus	DWORD	Input
* dwSlot	DWORD	Input
* dwFunction	DWORD	Input
• Card	WD_CARD	Input
* dwItems	DWORD	Output
* Item	WD_ITEMS[WD_CARD_ITEMS]	Input

Name	Type	Input/Output
• item	DWORD	Output
• fNotSharable	DWORD	Output
• I	union	Input
* Mem	struct	Input
• pPhysicalAddr	PHYS_ADDR	Output
• qwBytes	UINT64	Output
• pTransAddr	KPTR	N/A
• pUserDirectAddr	UPTR	N/A
• dwBar	DWORD	Output
• dwOptions	DWORD	N/A
• pReserved	KPTR	N/A
* IO	struct	Input
• pAddr	KPTR	Output
• dwBytes	DWORD	Output
• dwBar	DWORD	Output
* Int	struct	Input
• dwInterrupt	DWORD	Output
• dwOptions	DWORD	Output
• hInterrupt	DWORD	N/A
• dwReserved1	DWORD	N/A
• pReserved2	KPTR	N/A
* Bus	WD_BUS	Input
• dwBusType	WD_BUS_TYPE	Output
• dwBusNum	DWORD	Output
• dwSlotFunc	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pPciCard	Pointer to a PCI card information structure:
• pciSlot	PCI slot information structure:
* dwBus	PCI bus number (0 based)
* dwSlot	PCI slot number (0 based)
* dwFunction	PCI function number (0 based)
• Card	Card information structure:
* dwItems	Number of items detected on the card
* Item	Card items information structure:
• item	Item type — `ITEM_MEMORY`, `ITEM_IO`, `ITEM_INTERRUPT`, or `ITEM_BUS`
• fNotSharable	• **1** — Non-sharable resource; should be locked for exclusive use • **0** — Sharable resource Note: You can modify the value of this field before registering the card and its resources using `WD_CardRegister()` [2.6].
• I	Specific data according to the item type (**`pPciCard->Card.Item.item`**)
* Mem	Memory-item descriptor (type=`ITEM_MEMORY`)
• pPhysicalAddr	First address of the physical memory range.
• qwBytes	Length of the memory range, in bytes
• dwBar	Base Address Register (BAR) number of the memory range
* IO	I/O-item descriptor (type=`ITEM_IO`)
• pAddr	First address of the I/O range
• dwBytes	Length of the I/O range, in bytes
• dwBar	Base Address Register (BAR) number for the I/O range
* Int	Interrupt-item descriptor (type=`ITEM_INTERRUPT`)
• dwInterrupt	Physical interrupt request (IRQ) number

Name	Description
• dwOptions	Interrupt options bit-mask, which can consist of a combination of any of the following flags for indicating the interrupt types supported by the device. The default interrupt type, when none of the following flags is set, is legacy edge-triggered interrupts. • **`INTERRUPT_MESSAGE_X`** — Indicates that the hardware supports Extended Message-Signaled Interrupts (MSI-X). This option is applicable only to PCI cards on Linux — see information in the **WinDriver PCI Manual**. • **`INTERRUPT_MESSAGE`** — On Linux, indicates that the hardware supports Message-Signaled Interrupts (MSI). On Windows, indicates that the hardware supports MSI or MSI-X. This option is applicable only to PCI cards on Linux and Windows 7 and higher — see information in the **WinDriver PCI Manual**. • **`INTERRUPT_LEVEL_SENSITIVE`** — Indicates that the hardware supports level-sensitive interrupts.
* Bus	Bus-item descriptor (type=`ITEM_BUS`)
• dwBusType	The card's bus type. For a PCI card the bus type is **`WD_BUS_PCI`**.
• dwBusNum	The card's bus number
• dwSlotFunc	A combination of the card's bus slot and function numbers: the lower three bits represent the function number and the remaining bits represent the slot number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot number of 0x10 (remaining bits: 10000).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- For PCI devices, if the I/O, memory and IRQ information is available from the Plug-and-Play Manager, the information is obtained from there. Otherwise, the information is read directly from the PCI configuration registers. Note: for Windows, you must have an **inf** file installed.

- If the IRQ is obtained from the Plug-and-Play Manager, it is mapped and therefore may differ from the physical IRQ.

Example

```
WD_PCI_CARD_INFO pciCardInfo;
WD_CARD Card;

BZERO(pciCardInfo);
pciCardInfo.pciSlot = pciSlot;
WD_PciGetCardInfo(hWD, &pciCardInfo);
if (pciCardInfo.Card.dwItems!=0) /* At least one item was found */
{
    Card = pciCardInfo.Card;
}
else
{
    printf("Failed fetching PCI card information\n");
}
```

2.5. WD_PciConfigDump()

Purpose

Reads/writes from/to the PCI configuration space of a selected PCI card or the extended configuration space of a selected PCI Express card.

Access to the PCI Express extended configuration space is supported on target platforms that support such access (e.g., Windows and Linux). For such platforms, all *PCI* references in the following documentation include PCI Express as well.

Prototype

```
DWORD WD_PciConfigDump(
    HANDLE hWD,
    WD_PCI_CONFIG_DUMP *pConfig);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pConfig	WD_PCI_CONFIG_DUMP*	
• pciSlot	WD_PCI_SLOT	
* dwBus	DWORD	Input
* dwSlot	DWORD	Input
* dwFunction	DWORD	Input
• pBuffer	PVOID	Input/Output
• dwOffset	DWORD	Input
• dwBytes	DWORD	Input
• fIsRead	DWORD	Input
• dwResult	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pConfig	Pointer to a PCI configuration space information structure:
• pciSlot	PCI slot information structure:
* dwBus	PCI bus number (0 based)
* dwSlot	PCI slot number (0 based)
* dwFunction	PCI function number (0 based)
• pBuffer	A pointer to the data that is read from the PCI configuration space (if **`fIsRead`** is `TRUE`) or a pointer to the data to write to the PCI configuration space (if **`fIsRead`** is `FALSE`)
• dwOffset	The offset of the specific register(s) in the PCI configuration space to read/write from/to
• dwBytes	Number of bytes to read/write
• fIsRead	If **`TRUE`** — read from the PCI configuration space; If **`FALSE`** — write to the PCI configuration space
• dwResult	Result of the PCI configuration space read/write. Can be any of the following `PCI_ACCESS_RESULT` enumeration values: • **`PCI_ACCESS_OK`** — read/write succeeded • **`PCI_ACCESS_ERROR`** — read/write failed • **`PCI_BAD_BUS`** — the specified bus does not exist • **`PCI_BAD_SLOT`** — the specified slot or function does not exist

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_PCI_CONFIG_DUMP pciConfig;
DWORD dwStatus;
WORD aBuffer[2];

BZERO(pciConfig);
pciConfig.pciSlot.dwBus = 0;
pciConfig.pciSlot.dwSlot = 3;
pciConfig.pciSlot.dwFunction = 0;
pciConfig.pBuffer = aBuffer;
pciConfig.dwOffset = 0;
pciConfig.dwBytes = sizeof(aBuffer);
pciConfig.fIsRead = TRUE;

dwStatus = WD_PciConfigDump(hWD, &pciConfig);
if (dwStatus)
{
    printf("WD_PciConfigDump failed: %s\n", Stat2Str(dwStatus));
}
else
{
    printf("Card in Bus 0, Slot 3, Funcion 0 has Vendor ID %x "
        "Device ID %x\n", aBuffer[0], aBuffer[1]);
}
```

2.6. WD_CardRegister()

Purpose

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`() [2.17] and/or `WD_IntEnable`() [3.2].

Prototype

```
DWORD WD_CardRegister(
    HANDLE hWD,
    WD_CARD_REGISTER *pCardReg);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pCardReg	WD_CARD_REGISTER*	
• Card	WD_CARD	
* dwItems	DWORD	Input
* Item	WD_ITEMS[WD_CARD_ITEMS]	
• item	DWORD	Input
• fNotSharable	DWORD	Input
• I	union	
* Mem	struct	
• pPhysicalAddr	PHYS_ADDR	Input
• qwBytes	UINT64	Input
• pTransAddr	KPTR	Output

Name	Type	Input/Output
• pUserDirectAddr	UPTR	Output
• dwBar	DWORD	Input
• dwOptions	DWORD	Input
• pReserved	KPTR	N/A
* IO	struct	
• pAddr	KPTR	Input
• dwBytes	DWORD	Input
• dwBar	DWORD	Input
* Int	struct	
• dwInterrupt	DWORD	Input
• dwOptions	DWORD	Input
• hInterrupt	DWORD	Output
• dwReserved1	DWORD	N/A
• pReserved2	KPTR	N/A
* Bus	WD_BUS	
• dwBusType	WD_BUS_TYPE	Input
• dwBusNum	DWORD	Input
• dwSlotFunc	DWORD	Input
• fCheckLockOnly	DWORD	Input
• hCard	DWORD	Output
• dwOptions	DWORD	Input
• cName	CHAR[32]	Input
• cDescription	CHAR[100]	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pCardReg	Pointer to a card registration information structure:
• Card	Card information structure. For a PCI device it is recommended to pass the card structure received from a previous call to `WD_PciGetCardInfo()` [2.4] — refer to Remark 1 in this section.
* dwItems	Number of items detected on the card
* Item	Card items information structure:

Name	Description
• item	Item type — `ITEM_MEMORY`, `ITEM_IO`, `ITEM_INTERRUPT`, or `ITEM_BUS`
• fNotSharable	• **1** — Non-sharable resource; should be locked for exclusive use • **0** — Sharable resource
• I	Specific data according to the item type (**`pCardReg->Card.Item.item`**)
* Mem	Memory-item descriptor (type=`ITEM_MEMORY`)
• pPhysicalAddr	First address of the physical memory range. For PCI, this field is ignored and the physical address is retrieved directly from the card, because the 32-bit field size cannot correctly store 64-bit addresses.
• qwBytes	Length (in bytes) of the memory range
• pTransAddr	Kernel-mode mapping of the memory range's physical base address. This address should be used when setting the memory address in calls to `WD_Transfer`() [2.9] or `WD_MultiTransfer`() [2.10] or when accessing memory directly from a Kernel PlugIn driver.
• pUserDirectAddr	User-mode mapping of the memory range's physical base address. This address should be used for accessing a memory address directly from a user-mode process.
• dwBar	Base Address Register number of PCI card
• dwOptions	A bit-mask of memory-item registration flags — a combination of any of the of the following `WD_ITEM_MEM_OPTIONS` enumeration values: • **`WD_ITEM_MEM_DO_NOT_MAP_KERNEL`**: Avoid mapping the item's physical memory to the kernel address space (`I.Mem.pTransAddr` not set); map the memory only to the user-mode virtual address space (mapped base address: `I.Mem.pUserDirectAddr`). For more information, refer to Remark 2 in this section. NOTE: This flag is applicable only to memory items. • **`WD_ITEM_MEM_ALLOW_CACHE`** (Windows): Map the item's physical memory (base address: `I.Mem.pPhysicalAddr`) as cached. NOTE: This flag is applicable only to memory items that pertain to the host's RAM, as opposed to local memory on the card.
* IO	I/O-item descriptor (type=`ITEM_IO`)
• pAddr	First address of the I/O range
• dwBytes	Length of the I/O range, in bytes
• dwBar	Base Address Register (BAR) number for the I/O range

Name	Description
* Int	Interrupt-item descriptor (type=`ITEM_INTERRUPT`)
• dwInterrupt	Physical interrupt request (IRQ) number
• dwOptions	Interrupt options bit-mask, which can consist of a combination of any of the following flags: **Interrupt type flags:** NOTE: In the call to `WD_CardRegister`() the interrupt type flags are appliable only to ISA devices. For Plug-and-Play hardware (PCI) the function retrieves the supported interrupt types independently. • **`INTERRUPT_LEVEL_SENSITIVE`** — indicates that the device supports level-sensitive interrupts. • **`INTERRUPT_LATCHED`** — indicates that the device supports legacy edge-triggered interrupts. The value of this flag is zero, therefore it is applicable only when no other interrupt flag is set. **Miscellaneous interrupt flags:**
• hInterrupt	An interrupt handle to be used in calls to `InterruptEnable`() [2.17] or `WD_IntEnable`() [3.2].
* Bus	Bus-item descriptor (type=`ITEM_BUS`)
• dwBusType	The card's bus type — any of the following `WD_BUS_TYPE` enumeration values: • **`WD_BUS_PCI`** — PCI bus • **`WD_BUS_ISA`** — ISA bus • **`WD_BUS_EISA`** — EISA bus
• dwBusNum	Bus number
• dwSlotFunc	A combination of the card's bus slot/socket and function numbers: the lower three bits represent the function number and the remaining bits represent the slot/socket number. For example: a value of 0x80 (<=> 10000000 binary) corresponds to a function number of 0 (lower 3 bits: 000) and a slot/socket number of 0x10 (remaining bits: 10000).
• fCheckLockOnly	Set to **`TRUE`** to check if the defined card resources can be locked (in which case **`hCard`** will be set to 1), or whether they are already locked for exclusive use. When this flag is set to `TRUE` the function doesn't actually locking the specified resources.
• hCard	Handle to the card resources defined in the **`Card`** field. This handle should later be passed to `WD_CardUnregister`() [2.8] in order to free the resources. If the card registration fails, this field is set to 0. When **`fCheckLockOnly`** is set to `TRUE`, `hCard` is set to 1 if the card's resources can be be locked successfully (i.e., the resources aren't currently locked for exclusive use), or 0 otherwise.
• dwOptions	Should always be set to zero
• cName	Card name (optional)

Name	Description
• cDescription	Card description (optional)

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

1. For PCI the `cardReg.Card` input resources information should be retrieved from the Plug-and-Play Manager via `WD_PciGetCardInfo`() [2.4].

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`() [2.4] 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.

Example

```
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;
}
```

2.7. WD_CardCleanupSetup()

Purpose

Sets a list of transfer cleanup commands to be performed for the specified card on any of the following occasions:

- The application 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.

Prototype

```
DWORD WD_CardCleanupSetup(
    HANDLE hWD,
    WD_CARD_CLEANUP *pCardCleanup)
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pCardCleanup	WD_CARD_CLEANUP*	
• hCard	DWORD	Input
• Cmds	WD_TRANSFER*	Input
• dwCmds	DWORD	Input
• dwOptions	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pCardCleanup	Pointer to a card clean-up information structure:
• hCard	Handle to the relevant card as received from `WD_CardRegister()` [2.6]
• Cmds	Pointer to an array of cleanup transfer commands to be performed
• dwCmds	Number of cleanup commands in the **`Cmds`** array
• bForceCleanup	If **`0`**: The cleanup transfer commands (`Cmd`) will be performed in either of the following cases: • When the application exist abnormally. • When the application exits normally but without calling `WD_CardUnregister()` [2.8] to unregister the card. If the **`WD_FORCE_CLEANUP`** flag is set: The cleanup transfer commands will be performed both in the two cases described above, as well as in the following case: • When `WD_CardUnregister()` [2.8] is called to unregister the card.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

You should call this function right after calling `WD_CardRegister()` [2.6].

Example

```
WD_CARD_CLEANUP cleanup;
BZERO(cleanup);

/* Set-up the cleanup struct with the cleanup information */

dwStatus = WD_CardCleanupSetup(hWD, &cleanup);
if (dwStatus)
{
    printf("WD_CardCleanupSetup failed: %s\n", Stat2Str(dwStatus));
}
```

2.8. WD_CardUnregister()

Purpose

Unregisters a device and frees the resources allocated to it.

Prototype

```
DWORD WD_CardUnregister(
    HANDLE hWD,
    WD_CARD_REGISTER *pCardReg);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pCardReg	WD_CARD_REGISTER*	
• Card	WD_CARD	N/A
• fCheckLockOnly	DWORD	N/A
• hCard	DWORD	Input
• dwOptions	DWORD	N/A
• cName	CHAR[32]	N/A
• cDescription	CHAR[100]	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pCardReg	Pointer to a card registration information structure
• hCard	Handle to the card to unregister, as received from `WD_CardRegister()` [2.6].

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_CardUnregister(hWD, &cardReg);
```

2.9. WD_Transfer()

Purpose

Executes a single read/write instruction to an I/O port or to a memory address.

Prototype

```
DWORD WD_Transfer(
    HANDLE hWD,
    WD_TRANSFER *pTrans);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pTrans	WD_TRANSFER*	
• cmdTrans	DWORD	Input
• pPort	KPTR	Input
• dwBytes	DWORD	Input
• fAutoinc	DWORD	Input
• dwOptions	DWORD	Input
• Data	union	
* Byte	BYTE	Input/Output
* Word	WORD	Input/Output
* Dword	UINT32	Input/Output
* Qword	UINT64	Input/Output
* pBuffer	PVOID	Input/Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pTrans	Pointer to a transfer information structure:
• cmdTrans	A value indicating the type of transfer to perform — see definition of the `WD_TRANSFER_CMD` enumeration in **windrvr.h**. In calls to `WD_Transfer()` the transfer command should be a read/write transfer command that conforms to the following format: **<dir><p>_[S]<size>** Explanation: **<dir>**: **R** for read, **W** for write **<p>**: **P** for I/O, **M** for memory **<S>**: signifies a string (block) transfer, as opposed to a single transfer **<size>**: BYTE, WORD, DWORD or QWORD .
• pPort	The memory or I/O address to access. For a memory transfer, use the kernel-mapping of the base address received from `WD_CardRegister()` [2.6] in `pCardReg->Card.Item[i]I.Mem.pTransAddr` (where 'i' is the index of the relevant memory item) + the desired offset from the beginning of the address range. For an I/O transfer, use the base address received from `WD_CardRegister()` [2.6] in `pCardReg->Card.Item[i]I.IO.pAddr` (where 'i' is the index of the relevant I/O item) + the desired offset from the beginning of the address range.
• dwBytes	Used in string transfers — number of bytes to transfer
• fAutoinc	Used for string transfers. If **TRUE**, I/O or memory address should be incremented for transfer. If **FALSE**, all data is transferred to the same port/address.
• dwOptions	Must be set to 0
• Data	The transfer data — input for read transfers, output for write transfers:
* Byte	Used for 8-bit transfers
* Word	Used for 16-bit transfers
* Dword	Used for 32-bit transfers
* Qword	Used for 64-bit transfers
* pBuffer	Used for string (block) transfers

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- 64-bit data transfers (QWORD) are available only for memory string transfers.
  Such transfers require 64-bit enabled PCI device, a 64-bit PCI bus, and an x86 CPU running under any of the operating systems supported by WinDriver. (Note: 64-bit data transfers performed with `WD_Transfer`() do not require 64-bit operating system/CPU).

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

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

Example

```
WD_TRANSFER Trans;
BYTE read_data;

BZERO(Trans);
Trans.cmdTrans = RP_BYTE; /* Read Port BYTE */
Trans.pPort = 0x210;
WD_Transfer(hWD, &Trans);
read_data = Trans.Data.Byte;
```

2.10. WD_MultiTransfer()

Purpose

Executes multiple read/write instructions to I/O ports and/or memory addresses.

Prototype

```
DWORD WD_MultiTransfer(
    HANDLE hWD,
    WD_TRANSFER *pTransferArray,
    DWORD dwNumTransfers);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pTransferArray	WD_TRANSFER*	
• cmdTrans	DWORD	Input
• pPort	KPTR	Input
• dwBytes	DWORD	Input
• fAutoinc	DWORD	Input
• dwOptions	DWORD	Input
• Data	union	
* Byte	BYTE	Input/Output
* Word	WORD	Input/Output
* Dword	UINT32	Input/Output
* Qword	UINT64	Input/Output
* pBuffer	PVOID	Input/Output
dwNumTransfers	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pTransferArray	Pointer to a beginning of an array of transfer information structures:
• cmdTrans	A value indicating the type of transfer to perform — see definition of the `WD_TRANSFER_CMD` enumeration in **windrvr.h**. In calls to `WD_MultiTransfer()` the transfer command should be a read/write transfer command that conforms to the following format: **`<dir><p>_[S]<size>`** Explanation: **`<dir>`**: **`R`** for read, **`W`** for write **`<p>`**: **`P`** for I/O, **`M`** for memory **`<S>`**: signifies a string (block) transfer, as opposed to a single transfer **`<size>`**: BYTE, WORD, DWORD or QWORD .
• pPort	The memory or I/O address to access. For a memory transfer, use the kernel-mapping of the base address received from `WD_CardRegister()` [2.6] in `pCardReg->Card.Item[i]I.Mem.pTransAddr` (where 'i' is the index of the relevant memory item) + the desired offset from the beginning of the address range. For an I/O transfer, use the base address received from `WD_CardRegister()` [2.6] in `pCardReg->Card.Item[i]I.IO.pAddr` (where 'i' is the index of the relevant I/O item) + the desired offset from the beginning of the address range.
• dwBytes	Used in string transfers — number of bytes to transfer
• fAutoinc	Used for string transfers. If **`TRUE`**, I/O or memory address should be incremented for transfer. If **`FALSE`**, all data is transferred to the same port/address.
• dwOptions	Must be set to 0
• Data	The transfer data — input for read transfers, output for write transfers:
* Byte	Used for 8-bit transfers
* Word	Used for 16-bit transfers
* Dword	Used for 32-bit transfers
* Qword	Used for 64-bit transfers
* pBuffer	Used for string (block) transfers
dwNumTransfers	The number of transfers to perform (the **`pTransferArray`** array should contain at least `dwNumTransfers` elements)

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- See `WD_Transfer()` [2.9] remarks.
- This function is not supported in Visual Basic.

Example

```
WD_TRANSFER Trans[4];
DWORD dwResult;
char *cData = "Message to send\n";

BZERO(Trans);
Trans[0].cmdTrans = WP_WORD;  /* Write Port WORD */
Trans[0].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;
```

2.11. WD_DMALock()

Purpose

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 user-mode, 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.

Prototype

```
DWORD WD_DMALock(
    HANDLE hWD,
    WD_DMA *pDma);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDma	WD_DMA*	
• hDma	DWORD	Output
• pUserAddr	PVOID	Input/Output
• pKernelAddr	KPTR	Output
• dwBytes	DWORD	Input
• dwOptions	DWORD	Input
• dwPages	DWORD	Input/Output
• hCard	DWORD	Input
• Page	WD_DMA_PAGE[WD_DMA_PAGES]	
* pPhysicalAddr	DMA_ADDR	Output
* dwBytes	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDma	Pointer to a DMA information structure:
• hDma	DMA buffer handle, which should be passed to `WD_DMAUnlock()` [2.12] when unlocking the DMA buffer, or 0 if the DMA lock failed
• pUserAddr	Virtual user-mode mapped DMA buffer address. Input in the case of Scatter/Gather and output in the case of contiguous buffer DMA.
• pKernelAddr	Kernel-mode mapped DMA buffer address. Relevant only for contiguous-buffer DMA (`dwOptions | DMA_KERNEL_BUFFER_ALLOC`) and for Scatter/Gather DMA on Windows.
• dwBytes	The size of the DMA buffer, in bytes

Name	Description
• dwOptions	DMA options bit-mask, which can consist of a combination of any of the following flags: • **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_KERNEL_BUFFER_ALLOC**: Allocate a contiguous DMA buffer in the physical memory. The default behavior (when this flag is not set) is to allocate a Scatter/Gather DMA buffer. • **DMA_KBUF_BELOW_16M**: Allocate the physical DMA buffer within the first 16MB of the main memory. This flag is applicable only to contiguous-buffer DMA, i.e., only when the DMA_KERNEL_BUFFER_ALLOC flag is also set. • **DMA_LARGE_BUFFER**: Enable locking of a large DMA buffer — dwBytes > 1MB. This flag is applicable only to Scatter/Gather DMA (i.e., when the DMA_KERNEL_BUFFER_ALLOC flag is not set). • **DMA_ALLOW_CACHE**: Allow caching of the DMA buffer. • **DMA_KERNEL_ONLY_MAP**: Do not map the allocated DMA buffer to the user mode (i.e., map it to kernel-mode only). This flag is applicable only in cases where the DMA_KERNEL_BUFFER_ALLOC flag is applicable — see above. • **DMA_ALLOW_64BIT_ADDRESS**: Allow allocation of 64-bit DMA addresses. This flag is supported on Windows and and Linux.
• dwPages	The number of pages allocated for the DMA buffer. For contiguous-buffer DMA this field is always set to 1. For a large (> 1MB) Scatter/Gather DMA buffer (dwOptions | DMA_LARGE_BUFFER) this field is used both as an input and an output parameter (otherwise only output) — see remark below for details.
• hCard	Handle to a card for which the DMA buffer is locked, as received from WD_CardRegister() [2.6].
• Page	Array of DMA page structures:
* pPhysicalAddr	Pointer to the physical address of the beginning of the page
* dwBytes	Page size, in bytes

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A].

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`() [2.9] / `WD_MultiTransfer`() [2.10], 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`() [2.6] 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].I.Mem.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.

Example

The following code demonstrates Scatter/Gather DMA allocation:

```
WD_DMA dma;
DWORD dwStatus;
PVOID pBuffer = malloc(20000);

BZERO(dma);
dma.dwBytes = 20000;
dma.pUserAddr = pBuffer;
dma.dwOptions = fIsRead ? DMA_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. */
}
```

2.12. WD_DMAUnlock()

Purpose

Unlocks a DMA buffer.

Prototype

```
DWORD WD_DMAUnlock(
    HANDLE hWD,
    WD_DMA *pDMA);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDma	WD_DMA*	
• hDma	DWORD	Input
• pUserAddr	PVOID	N/A
• pKernelAddr	KPTR	N/A
• dwBytes	DWORD	N/A
• dwOptions	DWORD	N/A
• dwPages	DWORD	N/A
• hCard	DWORD	N/A
• Page	WD_DMA_PAGE[WD_DMA_PAGES]	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDMA	Pointer to a DMA information structure:
• hDma	DMA buffer handle, received from `WD_DMALock()` [2.11].

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_DMAUnlock(hWD, &dma);
```

2.13. WD_KernelBufLock()

Purpose

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).

Prototype

```
DWORD WD_KernelBufLock(
    HANDLE hWD,
    WD_KERNEL_BUFFER *pKerBuf);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pKerBuf	WD_KERNEL_BUFFER*	
• hKerBuf	DWORD	Output
• dwOptions	DWORD	Input
• qwBytes	UINT64	Input
• pKernelAddr	KPTR	Output
• pUserAddr	UPTR	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pKerBuf	Pointer to a KERNEL_BUFFER information structure:
• hKerBuf	Kernel buffer handle which should be passed to `WD_KernelBufUnlock()` [2.14] when freeing the kernel buffer, or 0 if the kernel buffer lock failed
• dwOptions	Kernel buffer options bit-mask, which can consist of a combination of any of the following flags: • **`ALLOCATE_CONTIG_BUFFER`**:Allocates a physically contiguous buffer • **`ALLOCATE_CACHED_BUFFER`**:Allocates a cached buffer
• qwBytes	The size of the kernel buffer, in bytes.
• pKernelAddr	Pointer to the physical address of the beginning of the buffer

Name	Description
• pUserAddr	Pointer to the virtual address of the beginning of the buffer

Return Value

Returns WD_STATUS_SUCCESS (0) on success, or an appropriate error code otherwise [A].

Example

The following code demonstrates allocation of contiguous cached buffer:

```
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. */
}
```

2.14. WD_KernelBufUnlock()

Purpose

Frees kernel buffer.

Prototype

```
DWORD WD_KernelBufUnlock(
    HANDLE hWD,
    WD_KERNEL_BUFFER *pKerBuf);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pKerBuf	WD_KERNEL_BUFFER*	
• hKerBuf	DWORD	Input
• dwOptions	DWORD	N/A
• qwBytes	UINT64	N/A
• pKernelAddr	KPTR	N/A
• pUserAddr	UPTR	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pKerBuf	Pointer to a KERNEL_BUFFER information structure:
• hKerBuf	Kernel buffer handle

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- This API is currently not support on WinCE.

Example

```
WD_KernelBufUnlock(hWD, &buf);
```

2.15. WD_DMASyncCpu()

Purpose

Synchronizes the cache of all CPUs with the DMA buffer, by flushing the data from the CPU caches.

This function should be called before performing a DMA transfer (see Remarks below).

Prototype

```
DWORD WD_DMASyncCpu(
    HANDLE hWD,
    WD_DMA *pDMA);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDMA	WD_DMA*	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDMA	Pointer to a DMA information structure, received from a previous call to `WD_DMALock()` [2.11]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- An asynchronous DMA read or write operation accesses data in memory, not in the processor (CPU) cache, which resides between the CPU and the host's physical memory. Unless the CPU cache has been flushed, by calling `WD_DMASyncCpu()`, just before a read transfer, the data transferred into system memory by the DMA operation could be overwritten with stale data if the CPU cache is flushed later. Unless the CPU cache has been flushed by calling `WD_DMASyncCpu()` just before a write transfer, the data in the CPU cache might be more up-to-date than the copy in memory.

Example

```
WD_DMASyncCpu(hWD, &dma);
```

2.16. WD_DMASyncIo()

Purpose

Synchronizes the I/O caches with the DMA buffer, by flushing the data from the I/O caches and updating the CPU caches.

This function should be called after performing a DMA transfer (see Remarks below).

Prototype

```
DWORD WD_DMASyncIo(
    HANDLE hWD,
    WD_DMA *pDMA);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDMA	WD_DMA*	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDMA	Pointer to a DMA information structure, received from a previous call to `WD_DMALock()` [2.11]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- After a DMA transfer has been completed, the data can still be in the I/O cache, which resides between the host's physical memory and the bus-master DMA device, but not yet in the host's main memory. If the CPU accesses the memory, it might read the wrong data from the CPU cache. To ensure a consistent view of the memory for the CPU, you should call `WD_DMASyncIo()` after a DMA transfer in order to flush the data from the I/O cache and update the CPU cache with the new data. The function also flushes additional caches and buffers between the device and memory, such as caches associated with bus extenders or bridges.

Example

```
WD_DMASyncIo(hWD, &dma);
```

2.17. InterruptEnable()

Purpose

A convenient function for setting up interrupt handling.

Prototype

```
DWORD InterruptEnable(
    HANDLE *phThread,
    HANDLE hWD,
    WD_INTERRUPT *pInt,
    INT_HANDLER func,
    PVOID pData);
```

Parameters

Name	Type	Input/Output
phThread	HANDLE*	Output
hWD	HANDLE	Input
pInt	WD_INTERRUPT*	
• hInterrupt	DWORD	Input
• dwOptions	DWORD	Input
• Cmd	WD_TRANSFER*	Input
• dwCmds	DWORD	Input
• kpCall	WD_KERNEL_PLUGIN_CALL	
* hKernelPlugIn	DWORD	Input
* dwMessage	DWORD	N/A
* pData	PVOID	N/A
* dwResult	DWORD	N/A
• fEnableOk	DWORD	N/A
• dwCounter	DWORD	N/A
• dwLost	DWORD	N/A
• fStopped	DWORD	N/A
• dwLastMessage	DWORD	N/A
• dwEnabledIntType	DWORD	Output
func	typedef void (*INT_HANDLER)(PVOID pData);	Input
pData	PVOID	Input

Description

Name	Description
phThread	Pointer to the handle to the spawned interrupt thread, which should be passed to `InterruptDisable`() [2.18] when disabling the interrupt
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open`() [5.2]
pInt	Pointer to an interrupt information structure:
• hInterrupt	Internal interrupt handle, as received from `WD_CardRegister`() [2.6] in `I.Int.hInterrupt`
• dwOptions	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 (**`func`**). 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: • **`INTERRUPT_MESSAGE_X`**: Extended Message-Signaled Interrupts (MSI-X) • **`INTERRUPT_MESSAGE`**: Message-Signaled Interrupts (MSI) • **`INTERRUPT_LEVEL_SENSITIVE`** — Legacy level-sensitive interrupts

Name	Description
• Cmd	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. NOTE: • Memory allocated for the transfer commands must remain available until the interrupts are disabled . • When handling level-sensitive interrupts (such as PCI interrupts) in the user mode (without a Kernel PlugIn driver), you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received. The commands in the array can be either of the following: • A read/write transfer command that conforms to the following format: **`<dir><p>_[S]<size>`** — see the description of `pTrans->cmdTrans` in Section 2.9. • **`CMD_MASK`**: an interrupt mask command for determining the source of the interrupt: When this command is set, upon the arrival of an interrupt in the kernel WinDriver masks the value of the previous read command in the `WD_TRANSFER` commands array with the mask that is set in the relevant `Data` field union member of the mask transfer command. For example, for a `Cmd WD_TRANSFER` array, if `Cmd[i-1].cmdTrans` is `RM_BYTE`, WinDriver performs the following mask: `Cmd[i-1].Data.Byte & Cmd[i].Data.Byte`. If the mask is successful, the driver claims ownership of the interrupt and when the control is returned to the user mode, the interrupt handler routine that was passed to the interrupt enable function is invoked; otherwise, the driver rejects ownership of the interrupt, the interrupt handler routine is not invoked and the subsequent transfer commands in the array are not executed. (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.) NOTE: A `CMD_MASK` command must be preceded by a read transfer command (`RM_XXX` / `RP_XXX`).
• dwCmds	Number of transfer commands in the **`Cmd`** array
• kpCall	Kernel PlugIn message information structure:
* hKernelPlugIn	Handle to Kernel PlugIn returned from `WD_KernelPlugInOpen()` [6.1].

Name	Description
• dwEnabledIntType	Updated by the function to indicate the type of interrupt enabled for the device. Can be set to any of the following values: • **`INTERRUPT_MESSAGE_X`**: Extended Message-Signaled Interrupts (MSI-X). * • **`INTERRUPT_MESSAGE`**: Message-Signaled Interrupts (MSI). * • **`INTERRUPT_LEVEL_SENSITIVE`**: Legacy level-sensitive interrupts. • **`0`**: Default interrupt type — Legacy edge-triggered interrupts. **NOTE:** * The Windows APIs do not distinguish between MSI and MSI-X; therefore, on this OS the WinDriver functions set the **`INTERRUPT_MESSAGE`** flag for both MSI and MSI-X. ** This field normally relevant only in the case of PCI devices that support more than one type of interrupt.
func	The interrupt handler routine, which will be called once for every interrupt occurrence. (Note: The function type `INT_HANDLER` is defined in **windrvr_int_thread.h**.)
pData	Pointer to the data to be passed as the argument to the interrupt handler routine (**`func`**)

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- You can view the implementation of this function in **WinDriver/src/wdapi/windrvr_int_thread.c**.
- `InterruptEnable`() uses the low-level `WD_IntEnable`() [3.2], `WD_IntWait`() [3.3] and `WD_IntCount`() [3.4] functions and replaces the need to call these functions separately, although this is still possible.
- 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, `InterruptEnable`() will fail with a `WD_NO_DEVICE_OBJECT` error [A]. To improve the interrupt handling rate on Windows and Linux, consider using WinDriver's Kernel PlugIn feature (see the **WinDriver PCI Manual**).

Example

```
VOID DLLCALLCONV interrupt_handler(PVOID pData)
{
    WD_INTERRUPT *pIntrp = (WD_INTERRUPT *)pData;
    /* Implement your interrupt handler routine here */

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

....
main()
{
    WD_CARD_REGISTER cardReg;
    WD_INTERRUPT Intrp;
    HANDLE hWD, thread_handle;

    ....
    hWD = WD_Open();
    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_INTERRUPT;
    cardReg.Card.Item[0].fNotSharable = 1;
    cardReg.Card.Item[0].I.Int.dwInterrupt = MY_IRQ;
    cardReg.Card.Item[0].I.Int.dwOptions = 0;
    ....
    WD_CardRegister(hWd, &cardReg);
    ....
    PVOID pdata = NULL;
    BZERO (Intrp);
    Intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
    Intrp.Cmd = NULL;
    Intrp.dwCmds = 0;
    Intrp.dwOptions = 0;
    printf("starting interrupt thread\n");
    pData = &Intrp;

    if (InterruptEnable(&thread_handle, hWD, &Intrp,
        interrupt_handler, pdata))
    {
        printf ("failed enabling interrupt\n")
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);
        InterruptDisable(thread_handle); /* Calls WD_IntDisable() */
    }
    WD_CardUnregister(hWD, &cardReg);
    ....
}
```

2.18. InterruptDisable()

Purpose

A convenient function for shutting down interrupt handling.

Prototype

```
DWORD InterruptDisable(HANDLE hThread);
```

Parameters

Name	Type	Input/Output
hThread	HANDLE	Input

Description

Name	Description
hThread	Interrupt thread handle, as received from `InterruptEnable`() [2.17] (`*phThread`)

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- You can view the implementation of this function in **WinDriver/src/wdapi/windrvr_int_thread.c**.
- `InterruptDisable`() uses the low-level `WD_IntDisable`() function [3.5]. When using `InterruptEnable`() [2.17] to enable interrupts, instead of calling the low-level `WD_IntEnable`() [3.2], `WD_IntWait`() [3.3] and `WD_IntCount`() [3.4] functions separately, the interrupts should be disabled using `InterruptDisable`() and not `WD_IntDisable`()

Example

```
main()
{
    ....
    if (InterruptEnable(&thread_handle, hWD, &Intrp,
        interrupt_handler, pData))
    {
        printf("failed enabling interrupt\n");
    }
    else
    {
        printf("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);
        InterruptDisable(thread_handle); /* Calls WD_IntDisable() */
    }
    ....
}
```

Chapter 3
Low-Level WD_xxx Interrupt Functions

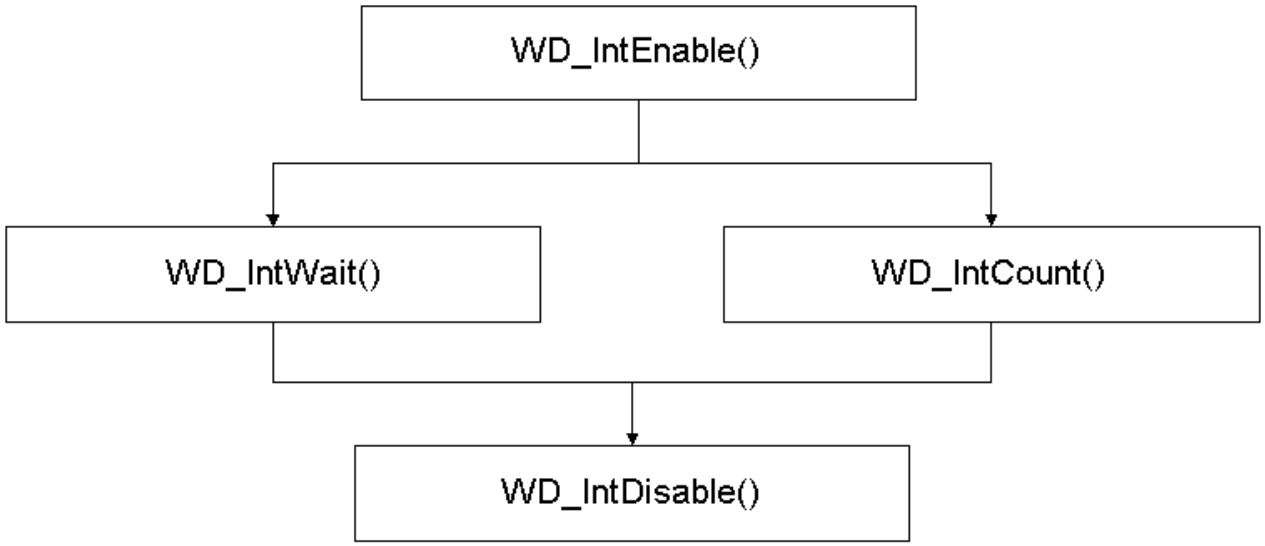
This chapter describes low-level `WD_xxx` PCI/ISA WinDriver functions.

It is recommended to use the API from WinDriver's **WDC** library, which provides convenient wrappers to the basic `WD_xxx` PCI/ISA API (see the **WinDriver PCI Manual**). If you decide not to use the WDC API, consider using the high-level `InterruptEnable`() [2.17] and `InterruptDisable`() [2.18] functions instead of directly using the low-level functions described below.

3.1. WinDriver Low-Level Interrupt Calling Sequence

The following is a typical calling sequence of the WinDriver API, used for servicing interrupts. The `InterruptEnable`() and `InterruptDisable`() functions enable interrupt handling in a more convenient manner.

Figure 3.1. Low-Level WinDriver Interrupt API Calling Sequence



3.2. WD_IntEnable()

Purpose

Registers an interrupt service routine (ISR) to be called upon interrupt.

Prototype

```
DWORD WD_IntEnable(
   HANDLE hWD,
   WD_INTERRUPT *pInterrupt);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pInterrupt	WD_INTERRUPT*	
• hInterrupt	HANDLE	Input
• dwOptions	DWORD	Input
• Cmd	WD_TRANSFER*	Input
• dwCmds	DWORD	Input
• kpCall	WD_KERNEL_PLUGIN_CALL	
* hKernelPlugIn	HANDLE	Input
* dwMessage	DWORD	N/A
* pData	PVOID	N/A
* dwResult	DWORD	N/A
• fEnableOk	DWORD	Output
• dwCounter	DWORD	N/A
• dwLost	DWORD	N/A
• fStopped	DWORD	N/A
• dwLastMessage	DWORD	N/A
• dwEnabledIntType	DWORD	Output

Description

Name	Description
HWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pInterrupt	Pointer to an interrupt information structure:

Name	Description
• hInterrupt	Interrupt handle. The handle is returned by `WD_CardRegister`() [2.6] in `pCardReg->Card.Item[i].I.Int.hInterrupt`.
• dwOptions	A bit mask flag. May be 0 for no option, or: • **`INTERRUPT_CMD_COPY`**: if set, WinDriver will copy the data received from the read commands that were used to acknowledge the interrupt in the kernel, back to the user mode. The data will be available when `WD_IntWait`() [3.3] returns.
• Cmd	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. NOTE: • Memory allocated for the transfer commands must remain available until the interrupts are disabled . • When handling level-sensitive interrupts (such as PCI interrupts) in the user mode (without a Kernel PlugIn driver), you must use this array to define the hardware-specific commands for acknowledging the interrupts in the kernel, immediately when they are received. The commands in the array can be either of the following: • A read/write transfer command that conforms to the following format: **`<dir><p>_[S]<size>`** — see the description of `pTrans->cmdTrans` in Section 2.9. • **`CMD_MASK`**: an interrupt mask command for determining the source of the interrupt: When this command is set, upon the arrival of an interrupt in the kernel WinDriver masks the value of the previous read command in the `WD_TRANSFER` commands array with the mask that is set in the relevant `Data` field union member of the mask transfer command. For example, for a `Cmd WD_TRANSFER` array, if `Cmd[i-1].cmdTrans` is `RM_BYTE`, WinDriver performs the following mask: `Cmd[i-1].Data.Byte & Cmd[i].Data.Byte`. If the mask is successful, the driver claims ownership of the interrupt and when the control is returned to the user mode, the interrupt handler routine that was passed to the interrupt enable function is invoked; otherwise, the driver rejects ownership of the interrupt, the interrupt handler routine is not invoked and the subsequent transfer commands in the array are not executed. (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.) NOTE: A `CMD_MASK` command must be preceded by a read transfer command (`RM_XXX` / `RP_XXX`).
• dwCmds	Number of transfer commands in **`Cmd`** array
• kpCall	Pointer to a Kernel PlugIn message information structure:

Name	Description
* hKernelPlugIn	Handle to Kernel PlugIn returned from `WD_KernelPlugInOpen()` [6.1]
• fEnableOk	Set by the function to `TRUE` if `WD_IntEnable()` [3.2] succeeds
• dwEnabledIntType	Updated by the function to indicate the type of interrupt enabled for the device. Can be set to any of the following values: • **`INTERRUPT_MESSAGE_X`**: Extended Message-Signaled Interrupts (MSI-X). * • **`INTERRUPT_MESSAGE`**: Message-Signaled Interrupts (MSI). * • **`INTERRUPT_LEVEL_SENSITIVE`**: Legacy level-sensitive interrupts. • **`0`**: Default interrupt type — Legacy edge-triggered interrupts. **NOTE:** * The Windows APIs do not distinguish between MSI and MSI-X; therefore, on this OS the WinDriver functions set the **`INTERRUPT_MESSAGE`** flag for both MSI and MSI-X. ** This field normally relevant only in the case of PCI devices that support more than one type of interrupt.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- For more information regarding interrupt handling please refer to the "ISA/EISA and PCI 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 [A].

Example

- ```
  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");
      }
  ```

- For another example please refer to **WinDriver/samples/pci_diag/pci_lib.c**.

3.3. WD_IntWait()

Purpose

Waits for an interrupt.

Prototype

```
DWORD WD_IntWait(
    HANDLE hWD,
    WD_INTERRUPT *pInterrupt);
```
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pInterrupt	WD_INTERRUPT*	
• hInterrupt	HANDLE	Input
• dwOptions	DWORD	N/A
• Cmd	WD_TRANSFER*	N/A
• dwCmds	DWORD	N/A
• kpCall	WD_KERNEL_PLUGIN_CALL	N/A
• fEnableOk	DWORD	N/A
• dwCounter	DWORD	Output
• dwLost	DWORD	Output
• fStopped	DWORD	Output
• dwLastMessage	DWORD	Output
• dwEnabledIntType	DWORD	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pInterrupt	Pointer to an interrupt information structure:
• hInterrupt	Interrupt handle. The handle is returned by `WD_CardRegister()` [2.6] in `pCardReg->Card.Item[i].I.Int.hInterrupt`.
• dwCounter	Number of interrupts received
• dwLost	Number of interrupts that were acknowledged in the kernel mode but not yet handled in the user mode
• fStopped	Set by the function to any of the following values: • 0 — an interrupt occurred. • **`INTERRUPT_STOPPED`** — an interrupt was disabled while waiting for interrupts. • **`INTERRUPT_INTERRUPTED`** — while waiting for an interrupt, `WD_IntWait()` [3.3] was interrupted without an actual hardware interrupt (refer also to the remark below).
• dwLastMessage	Relevant only for MSI/MSI-X interrupts on Windows 7 and higher (see information in the **WinDriver PCI Manual**): When an interrupt occurs, WinDriver's kernel-mode interrupt handler sets this field to the interrupt's message data.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

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).

Example

```
for (;;)
{
    WD_IntWait(hWD, &Intrp);
    if (Intrp.fStopped)
        break;

    ProcessInterrupt(Intrp.dwCounter);
}
```

3.4. WD_IntCount()

Purpose

Retrieves the interrupts count since the call to `WD_IntEnable`() [3.2].

Prototype

```
void WD_IntCount(
    HANDLE hWD,
    WD_INTERRUPT *pInterrupt);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pInterrupt	WD_INTERRUPT*	
• hInterrupt	HANDLE	Input
• dwOptions	DWORD	N/A
• Cmd	WD_TRANSFER*	N/A
• dwCmds	DWORD	N/A
• kpCall	WD_KERNEL_PLUGIN_CALL	N/A
• fEnableOk	DWORD	N/A
• dwCounter	DWORD	Output
• dwLost	DWORD	Output
• fStopped	DWORD	Output
• dwLastMessage	DWORD	N/A
• dwEnabledIntType	DWORD	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pInterrupt	Pointer to an interrupt information structure:
• hInterrupt	Handle of interrupt, returned by `WD_CardRegister()` [2.6] in `I.Int.hInterrupt`.
• dwCounter	Number of interrupts received
• dwLost	Number of interrupts not yet handled
• fStopped	Set by the function to `TRUE` if interrupt was disabled while waiting for interrupts

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
DWORD dwNumInterrupts;
WD_IntCount(hWD, &Intrp);
dwNumInterrupts = Intrp.dwCounter;
```

3.5. WD_IntDisable()

Purpose

Disables interrupt processing.

Prototype

```
DWORD WD_IntDisable(
    HANDLE hWD,
    WD_INTERRUPT *pInterrupt);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pInterrupt	WD_INTERRUPT	
• hInterrupt	HANDLE	Input
• dwOptions	DWORD	N/A
• Cmd	WD_TRANSFER*	N/A
• dwCmds	DWORD	N/A
• kpCall	WD_KERNEL_PLUGIN_CALL	N/A
• fEnableOk	DWORD	N/A
• dwCounter	DWORD	N/A
• dwLost	DWORD	N/A
• fStopped	DWORD	N/A
• dwLastMessage	DWORD	N/A
• dwEnabledIntType	DWORD	N/A

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pInterrupt	Pointer to an interrupt information structure:
• hInterrupt	Handle of interrupt, returned by `WD_CardRegister()` [2.6] in `I.Int.hInterrupt`.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_IntDisable(hWD, &Intrp);
```

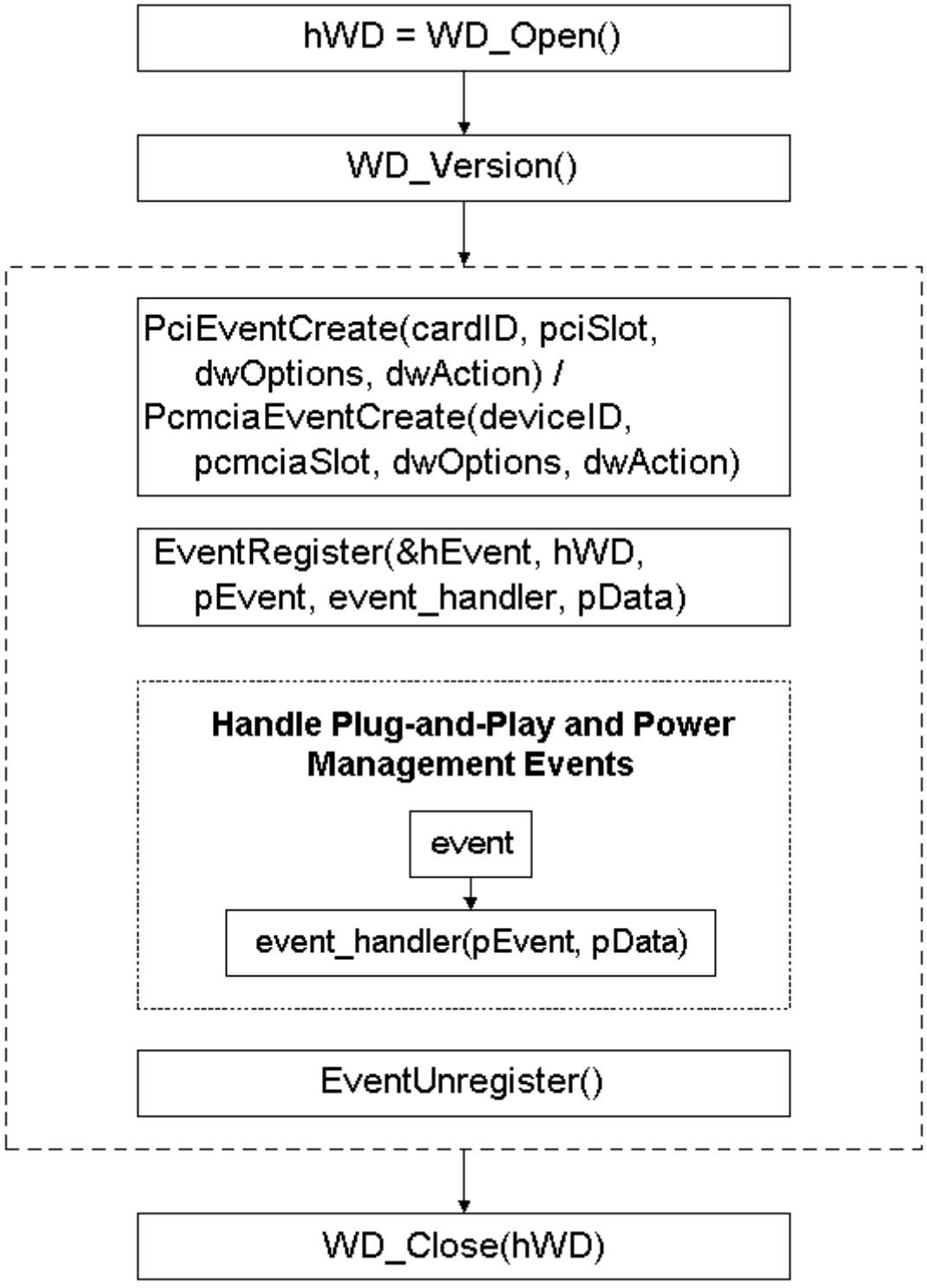
Chapter 4
Plug-and-Play and Power Management Functions

4.1. Calling_Sequence

The following is a typical calling sequence of the WinDriver API, used for handling Plug-and-Play and power management events. (Note: the use of the `PciEventCreate`() function, which appears in the sequence below, is optional).

It is recommended to use the high-level `WDC_EventRegister`() and `WDC_EventUnregister`() Plug-and-Play and power management APIs (see the **WinDriver PCI Manual**), instead of using the low-level APIs described in this chapter.

Figure 4.1. Plug-and-Play Calling Sequence



4.2. PciEventCreate()

Purpose

Convenience function for allocating and initializing a PCI Plug-and-Play and power management event structure.

Prototype

```
WD_EVENT * DLLCALLCONV PciEventCreate(
    WD_PCI_ID cardId,
    WD_PCI_SLOT pciSlot,
    DWORD dwOptions,
    DWORD dwAction);
```

Parameters

Name	Type	Input/Output
cardId	WD_PCI_ID	
• dwVendorId	DWORD	Input
• dwDeviceId	DWORD	Input
pciSlot	WD_PCI_SLOT	
• dwBus	DWORD	Input
• dwSlot	DWORD	Input
• dwFunction	DWORD	Input
dwOptions	DWORD	Input
dwAction	DWORD	Input

Description

Name	Description
cardId	PCI card information structure:
• dwVendorId	PCI vendor ID to register to. If zero, register to all PCI vendor IDs.
• dwDeviceId	PCI card ID to register to. If zero, register to all PCI device IDs.
pciSlot	PCI slot information:
• dwBus	PCI bus number to register to. If zero, register to all PCI buses.
• dwSlot	PCI slot to register to. If zero, register to all slots.
• dwFunction	PCI function to register to. If zero, register to all functions.
dwOptions	Can be either zero or: • **`WD_ACKNOWLEDGE`** — If set, the user can perform actions on the requested event before acknowledging it. The OS waits on the event until the user calls `WD_EventSend`() If `EventRegister`() [4.3] is called, `WD_EventSend`() will be called automatically after the callback function exits.
dwAction	A bit-mask indicating which events to register to: **Plug-and-Play events:** • **`WD_INSERT`** — Device was attached and configured by the operating system's Plug-and-Play Manager • **`WD_REMOVE`** — Device was unconfigured and detached by the operating system's Plug-and-Play Manager **Device power state:** • **`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

Return Value

Returns a new structure for handling PCI Plug-and-Play and power management events, or `NULL` if the allocation failed.

Remarks

It is the responsibility of the caller to free the event structure returned by the function (using `free`()) when it is no longer needed.

Example

```
PciEventCreate(cardId, pciSlot, WD_ACKNOWLEDGE, WD_INSERT | WD_REMOVE);
```

4.3. EventRegister()

Purpose

Registers your application to receive Plug-and-Play and power management event notifications, according to a predefined set of criteria.
The function receives an event handler callback, which will be invoked upon the occurrence of a registered event.

Prototype

```
DWORD EventRegister(
    HANDLE *phEvent,
    HANDLE hWD,
    WD_EVENT *pEvent,
    EVENT_HANDLER pFunc,
    void *pData);
```

Parameters

Name	Type	Input/Output
phEvent	HANDLE*	Output
hWD	HANDLE	Input
pEvent	WD_EVENT*	Input
• hEvent	DWORD	Output
• dwEventType	WD_EVENT_TYPE	Input
• dwAction	DWORD	Input
• dwEventId	DWORD	N/A
• hKernelPlugIn	DWORD	Input
• dwOptions	DWORD	N/A
dwOptions	DWORD	Input
• u	union	

Name	Type	Input/Output
* Pci	struct	
• cardId	WD_PCI_ID	
* dwVendorId	DWORD	Input
* dwDeviceId	DWORD	Input
• pciSlot	WD_PCI_SLOT	
* dwBus	DWORD	Input
* dwSlot	DWORD	Input
* dwFunction	DWORD	Input
* Usb	struct	Input
* Ipc	struct	N/A
• dwNumMatchTables	DWORD	Input
• matchTables	WDU_MATCH_TABLE[1]	Input
pFunc	typedef void (*EVENT_HANDLER)(WD_EVENT *pEvent, void *pData);	Input
pData	void	Input

Description

Name	Description
phEvent	Pointer to the handle to be used in calls to `EventUnregister`() [4.4], or `NULL` if the event registration fails.
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open`() [5.2]
pEvent	The criteria set for registering to receive event notifications.
• hEvent	Optional handle to be used by the low-level `WD_EventUnregister`() function; Zero when event registration fails.
• dwEventType	Event type. Can be either `WD_EVENT_TYPE_PCI` — for a PCI card, `WD_EVENT_TYPE_USB` — for a USB device or `WD_EVENT_TYPE_IPC` for an IPC related event (see `WD_EVENT_TYPE` enumeration values)

Name	Description
• dwAction	A bit-mask indicating which events to register to: **Plug-and-Play events:** • **WD_INSERT** — Device was attached and configured by the operating system's Plug-and-Play Manager • **WD_REMOVE** — Device was unconfigured and detached by the operating system's Plug-and-Play Manager **Device power state:** • **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
• hKernelPlugIn	Handle to Kernel PlugIn returned from `WD_KernelPlugInOpen()` [6.1] (when using the Kernel PlugIn to handle the events)
dwOptions	Can be either zero or: • **WD_ACKNOWLEDGE** — If set, the user can perform actions on the requested event before acknowledging it. The OS waits on the event until the user calls `WD_EventSend()` If `EventRegister()` [4.3] is called, `WD_EventSend()` will be called automatically after the callback function exits.
• u	Device information union:
* Pci	PCI card information:
• cardId	Card information structure:
* dwVendorId	PCI vendor ID to register to. If zero, register to all PCI vendor IDs.
* dwDeviceId	PCI card ID to register to. If zero, register to all PCI device IDs.
• pciSlot	PCI slot information:
* dwBus	PCI bus number to register to. If zero, register to all PCI buses.
* dwSlot	PCI slot to register to. If zero, register to all slots.
* dwFunction	PCI function to register to. If zero, register to all functions.
* Usb	USB device information — relevant only for USB devices
* Ipc	IPC message information - used internally by WinDriver

Name	Description
• dwNumMatchTables	Relevant only for USB devices
• matchTables	Relevant only for USB devices
pFunc	The callback function to call upon receipt of event notification.
pData	Pointer to the data to pass to the **pFunc** callback (`NULL` if there is no data to pass).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

This function wraps the low-level `WD_EventRegister`(), `WD_EventPull`(), `WD_EventSend`() and `InterruptEnable`() [2.17] functions.

Example

```
HANDLE *event_handle;
WD_EVENT event;
DWORD dwStatus;
BZERO(event);
event.dwAction = WD_INSERT | WD_REMOVE;
event.dwEventType = WD_EVENT_TYPE_PCI;
dwStatus = EventRegister(&event_handle, hWD, &event,
    event_handler_func, NULL);
if (dwStatus!=WD_STATUS_SUCCESS)
{
    printf("Failed register\n");
    return;
}
```

4.4. EventUnregister()

Purpose

Unregisters from receiving Plug-and-Play and power management event notifications.

Prototype

```
DWORD EventUnregister(HANDLE hEvent);
```

Parameters

Name	Type	Input/Output
hEvent	HANDLE	Input

Description

Name	Description
hEvent	Handle received from `EventRegister`() [4.3]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

This function wraps `WD_EventUnregister`() and `InterruptDisable`() [2.18].

Example

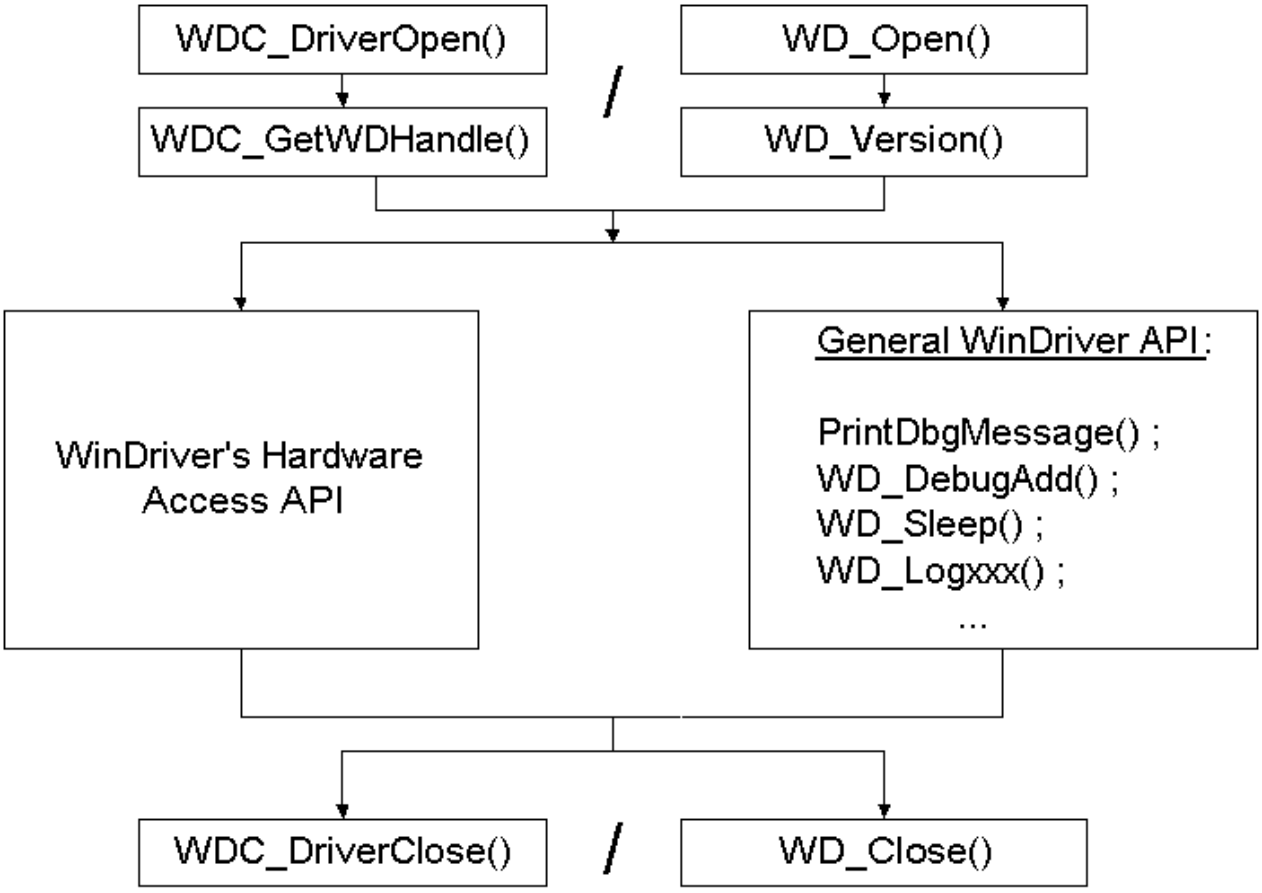
```
EventUnregister(event_handle);
```

Chapter 5
General WD_xxx Functions

5.1. Calling Sequence WinDriver — General Use

The following is a typical calling sequence for the WinDriver API.

Figure 5.1. WinDriver-API Calling Sequence



- We recommend calling the WinDriver function `WD_Version()` [5.3] after calling `WD_Open()` [5.2] and before calling any other WinDriver function. Its purpose is to return the WinDriver kernel module version number, thus providing the means to verify that your application is version compatible with the WinDriver kernel module.
- `WD_DebugAdd()` [5.6] and `WD_Sleep()` [5.8] can be called anywhere after `WD_Open()`

5.2. WD_Open()

Purpose

Opens a handle to access the WinDriver kernel module.
The handle is used by all WinDriver APIs, and therefore must be called before any other WinDriver API is called.

Prototype

```
HANDLE WD_Open(void);
```

Return Value

The handle to the WinDriver kernel module.
If device could not be opened, returns `INVALID_HANDLE_VALUE`.

Remarks

If you are a registered user, please refer to the documentation of `WD_License()` [5.9] for an example of how to register your WinDriver license.

Example

```
 HANDLE hWD;

hWD = WD_Open();
if (hWD == INVALID_HANDLE_VALUE)
{
    printf("Cannot open WinDriver device\n");
}
```

5.3. WD_Version()

Purpose

Returns the version number of the WinDriver kernel module currently running.

Prototype

```
DWORD WD_Version(
    HANDLE hWD,
    WD_VERSION *pVer);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pVer	WD_VERSION*	
• dwVer	DWORD	Output
• cVer	CHAR[128]	Output

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pVer	Pointer to a WinDriver version information structure:
• dwVer	The version number
• cVer	Version information string. The version string's size is limited to 128 characters (including the NULL terminator character).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_VERSION ver;

BZERO(ver);
WD_Version(hWD, &ver);
printf("%s\n", ver.cVer);
if (ver.dwVer < WD_VER)
{
    printf("Error - incorrect WinDriver version\n");
}
```

5.4. WD_Close()

Purpose

Closes the access to the WinDriver kernel module.

Prototype

```
void WD_Close(HANDLE hWD);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]

Return Value

None

Remarks

This function must be called when you finish using WinDriver kernel module.

Example

```
WD_Close(hWD);
```

5.5. WD_Debug()

Purpose

Sets debugging level for collecting debug messages.

Prototype

```
DWORD WD_Debug(
    HANDLE hWD,
    WD_DEBUG *pDebug);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDebug	WD_DEBUG*	Input
• dwCmd	DWORD	Input
• dwLevel	DWORD	Input
• dwSection	DWORD	Input
• dwLevelMessageBox	DWORD	Input
• dwBufferSize	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDebug	Pointer to a debug information structure:
• dwCmd	Debug command: Set filter, Clear buffer, etc. For more details please refer to `DEBUG_COMMAND` in **windrvr.h**.
• dwLevel	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to collect: Error, Warning, Info, Trace. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwSection	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the sections to collect: I/O, Memory, Interrupt, etc. Use `S_ALL` for all. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
• dwLevelMessageBox	Used for `dwCmd=DEBUG_SET_FILTER`. Sets the debugging level to print in a message box. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwBufferSize	Used for `dwCmd=DEBUG_SET_BUFFER`. The size of buffer in the kernel.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_DEBUG dbg;

BZERO(dbg);
dbg.dwCmd = DEBUG_SET_FILTER;
dbg.dwLevel = D_ERROR;
dbg.dwSection = S_ALL;
dbg.dwLevelMessageBox = D_ERROR;

WD_Debug(hWD, &dbg);
```

5.6. WD_DebugAdd()

Purpose

Sends debug messages to the debug log. Used by the driver code.

Prototype

```
DWORD WD_DebugAdd(
    HANDLE hWD,
    WD_DEBUG_ADD *pData);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pData	WD_DEBUG_ADD*	
• dwLevel	DWORD	Input
• dwSection	DWORD	Input
• pcBuffer	CHAR[256]	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pData	Pointer to an additional debug information structure:
• dwLevel	Assigns the level in the Debug Monitor, in which the data will be declared. If `dwLevel` is zero, `D_ERROR` will be declared. For more details please refer to `DEBUG_LEVEL` in **windrvr.h**.
• dwSection	Assigns the section in the Debug Monitor, in which the data will be declared. If `dwSection` is zero, `S_MISC` section will be declared. For more details please refer to `DEBUG_SECTION` in **windrvr.h**.
• pcBuffer	The string to copy into the message log.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_DEBUG_ADD add;

BZERO(add);
add.dwLevel = D_WARN;
add.dwSection = S_MISC;
sprintf(add.pcBuffer, "This message will be displayed in "
    "the Debug Monitor\n");
WD_DebugAdd(hWD, &add);
```

5.7. WD_DebugDump()

Purpose

Retrieves debug messages buffer.

Prototype

```
DWORD WD_DebugDump(
    HANDLE hWD,
    WD_DEBUG_DUMP *pDebugDump);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pDebug	WD_DEBUG_DUMP*	Input
• pcBuffer	PCHAR	Input/Output
• dwSize	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pDebugDump	Pointer to a debug dump information structure:
• pcBuffer	Buffer to receive debug messages
dwSize	Size of buffer in bytes

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
char buffer[1024];
WD_DEBUG_DUMP dump;
dump.pcBuffer=buffer;
dump.dwSize = sizeof(buffer);
WD_DebugDump(hWD, &dump);
```

5.8. WD_Sleep()

Purpose

Delays execution for a specific duration of time.

Prototype

```
DWORD WD_Sleep(
    HANDLE hWD,
    WD_SLEEP *pSleep);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pSleep	WD_SLEEP*	
• dwMicroSeconds	DWORD	Input
• dwOptions	DWORD	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open()` [5.2]
pSleep	Pointer to a sleep information structure
• dwMicroSeconds	Sleep time in microseconds — 1/1,000,000 of a second
• dwOptions	A bit-mask, which can be set to either of the following values: • **`Zero (0)`** — Delay execution by consuming CPU cycles (busy sleep); this is the default. • **`SLEEP_NON_BUSY`** — Delay execution without consuming CPU resources (non-busy sleep). Note: The accuracy of non-busy sleep is machine-dependent and cannot be guaranteed for short sleep intervals (< 1 millisecond).

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

Example usage: to access slow response hardware.

Example

```
WD_Sleep slp;

BZERO(slp);
slp.dwMicroSeconds = 200;
WD_Sleep(hWD, &slp);
```

5.9. WD_License()

Purpose

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.

Prototype

```
DWORD WD_License(
    HANDLE hWD,
    WD_LICENSE *pLicense);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pLicense	WD_LICENSE*	
• cLicense	CHAR[]	Input

Description

Name	Description
hWD	Handle to WinDriver's kernel-mode driver as received from `WD_Open`() [5.2]
pLicense	Pointer to a WinDriver license information structure:
• cLicense	A buffer to contain the license string that is to be transferred to the WinDriver kernel module.

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

When using a registered version, this function must be called before any other WinDriver API call, apart from WD_Open() [5.2], in order to register the license from the code.

Example

Example usage: Add registration routine to your application:

```
/* 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.cLicense, "12345abcde12345.CompanyName");
        dwStatus = WD_License(hWD, &lic);
        if (dwStatus != WD_STATUS_SUCCESS)
        {
             WD_Close(hWD);
             hWD = INVALID_HANDLE_VALUE;
        }
    }

    return hWD;
}
```

Chapter 6
Kernel PlugIn User-Mode Functions

This chapter describes the user-mode functions that initiate the Kernel PlugIn operations and activate its callbacks. For a description of the high-level **WDC** user-mode Kernel PlugIn APIs, which can be used instead of the low-level APIs described in this chapter, and for a description of the Kernel PlugIn structures and kernel-mode APIs, refer to the **WinDriver PCI Manual**.

6.1. WD_KernelPlugInOpen()

Purpose

Obtain a valid handle to the Kernel PlugIn.

Prototype

```
DWORD WD_KernelPlugInOpen(
    HANDLE hWD,
    WD_KERNEL_PLUGIN *pKernelPlugIn);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Output
pKernelPlugIn	WD_KERNEL_PLUGIN*	
• hKernelPlugIn	DWORD	Output
• pcDriverName	PCHAR	Input
• pcDriverPath	PCHAR	Input
• pOpenData	PVOID	Input

Description

Name	Description
hWD	Handle to WinDriver
pKernelPlugIn	Pointer to Kernel PlugIn information structure:
• hKernelPlugIn	Returns the handle to the Kernel PlugIn
• pcDriverName	Name of Kernel PlugIn to load, up to 8 characters
• pcDriverPath	This field should be set to `NULL`. WinDriver will search for the driver in the operating system's drivers/modules directory.
• pOpenData	Pointer to data that will be passed to the `KP_Open` callback in the Kernel PlugIn

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_KERNEL_PLUGIN kernelPlugIn;
BZERO(kernelPlugIn);

/* Tells WinDriver which driver to open */
kernelPlugIn.pcDriverName = "KPDriver";

HANDLE hWD = WD_Open(); /* Validate handle here */

dwStatus = WD_KernelPlugInOpen(hWD, &kernelPlugIn);
if (dwStatus)
{
    printf ("Failed opening a handle to the Kernel PlugIn. Error: 0x%x (%s)\n",
            dwStatus, Stat2Str(dwStatus));
}
else
{
   printf("Opened a handle to the Kernel PlugIn (0x%x)\n",
           kernelPlugIn.hKernelPlugIn);
}
```

6.2. WD_KernelPlugInClose()

Purpose

Closes the WinDriver Kernel PlugIn handle obtained from `WD_KernelPlugInOpen()` [6.1].

Prototype

```
DWORD WD_KernelPlugInClose(
    HANDLE hWD,
    WD_KERNEL_PLUGIN *pKernelPlugIn);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pKernelPlugIn	WD_KERNEL_PLUGIN*	Input

Description

Name	Description
hWD	Handle to WinDriver
pKernelPlugIn	Pointer to Kernel PlugIn information structure, which was previously passed to `WD_KernelPlugInOpen()` [6.1]

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Example

```
WD_KernelPlugInClose(hWD, &kernelPlugIn);
```

6.3. WD_KernelPlugInCall()

Purpose

Calls a routine in the Kernel PlugIn to be executed.

Prototype

```
DWORD WD_KernelPlugInCall(
    HANDLE hWD,
    WD_KERNEL_PLUGIN_CALL *pKernelPlugInCall);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pKernelPlugInCall	WD_KERNEL_PLUGIN_CALL*	Input
• hKernelPlugIn	DWORD	Input
• dwMessage	DWORD	Input
• pData	PVOID	Input/Output
• dwResult	DWORD	Output

Description

Name	Description
hWD	Handle to WinDriver
pKernelPlugInCall	Pointer to a Kernel PlugIn message structure:
• hKernelPlugIn	Handle to the Kernel PlugIn
• dwMessage	Message ID to pass to the `KP_Call` callback
• pData	Pointer to data to pass between the Kernel PlugIn driver and the user-mode application, via the `KP_Call` callback
• dwResult	Value set by `KP_Call` callback

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

Calling the `WD_KernelPlugInCall`() function [6.3] 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.

Example

```
WD_KERNEL_PLUGIN_CALL kpCall;

BZERO (kpCall);
/* Prepare the kpCall structure from WD_KernelPlugInOpen(): */
kpCall.hKernelPlugIn = hKernelPlugIn;
/* Set the message to pass to KP_Call. This will determine
   the action performed in the kernel: */
kpCall.dwMessage = MY_DRV_MSG;

kpCall.pData = &mydrv; /* The data to pass to the Kernel PlugIn */
dwStatus = WD_KernelPlugInCall(hWD, &kpCall);
if (dwStatus == WD_STATUS_SUCCESS)
{
    printf("Result = 0x%x\n", kpCall.dwResult);
}
else
{
    printf("WD_KernelPlugInCall() failed. Error: 0x%x (%s)\n",
            dwStatus, Stat2Str(dwStatus));
}
```

6.4. WD_IntEnable()

Purpose

Enables interrupt handling in the Kernel PlugIn.

Prototype

```
DWORD WD_IntEnable(HANDLE hWD, WD_INTERRUPT *pInterrupt);
```

Parameters

Name	Type	Input/Output
hWD	HANDLE	Input
pInterrupt	WD_INTERRUPT*	
• kpCall	WD_KERNEL_PLUGIN_CALL	
* hKernelPlugIn	HANDLE	Input
* dwMessage	DWORD	N/A
* pData	PVOID	Input
* dwResult	DWORD	N/A

Description

Name	Description
hWD	Handle to WinDriver
pInterrupt	Pointer to an interrupt information structure:
• kpCall	Kernel PlugIn message structure:
* hKernelPlugIn	Handle to the Kernel PlugIn. If zero, no Kernel PlugIn interrupt handler is installed
* pData	Pointer to data to pass to the `KP_IntEnable` callback in the Kernel PlugIn

Return Value

Returns `WD_STATUS_SUCCESS` (0) on success, or an appropriate error code otherwise [A].

Remarks

- If a valid handle to a Kernel PlugIn is passed to this function, the interrupts will be handled in the Kernel PlugIn. In such a case, the `KP_IntEnable` callback will execute as a result of the call to `WD_IntEnable`(), and upon receiving the interrupt your kernel-mode high-IRQL interrupt handler function — `KP_IntAtIrql` (legacy interrupts) or `KP_IntAtIrqlMSI` (MSI/MSI-X) — will execute. If this function returns a value greater than zero, your Deferred Procedure Call (DPC) handler — `KP_IntAtDpc` (legacy interrupts) or `KP_IntAtDpcMSI` (MSI/MSI-X) — will be called.

- For information regarding the additional `pInterrupt` fields, refer to the description of the `pInterrupt` parameter of the `WD_IntEnable`() function in Section 3.2.

Example

```
WD_INTERRUPT Intrp;
BZERO(Intrp);
Intrp.hInterrupt = hInterrupt; /* From WD_CardRegister() */
/* From WD_KernelPlugInOpen(): */
Intrp.kpCall.hKernelPlugIn = hKernelPlugIn;

WD_IntEnable(hWD, &Intrp);

if (!Intrp.fEnableOk)
    printf ("failed enabling interrupt\n");
```

Appendix A
WinDriver Status Codes

A.1. Introduction

Most of the WinDriver functions return a status code, where zero (`WD_STATUS_SUCCESS`) means success and a non-zero value means failure.
The `Stat2Str`() functions can be used to retrieve the status description string for a given status code. The status codes and their descriptive strings are listed below.

A.2. Status Codes Returned by WinDriver

Status Code	Description
WD_STATUS_SUCCESS	Success
WD_STATUS_INVALID_WD_HANDLE	Invalid WinDriver handle
WD_WINDRIVER_STATUS_ERROR	Error
WD_INVALID_HANDLE	Invalid handle
WD_INVALID_PIPE_NUMBER	Invalid pipe number
WD_READ_WRITE_CONFLICT	Conflict between read and write operations
WD_ZERO_PACKET_SIZE	Packet size is zero
WD_INSUFFICIENT_RESOURCES	Insufficient resources
WD_UNKNOWN_PIPE_TYPE	Unknown pipe type
WD_SYSTEM_INTERNAL_ERROR	Internal system error
WD_DATA_MISMATCH	Data mismatch
WD_NO_LICENSE	No valid license
WD_NOT_IMPLEMENTED	Function not implemented
WD_KERPLUG_FAILURE	Kernel PlugIn failure
WD_FAILED_ENABLING_INTERRUPT	Failed enabling interrupt
WD_INTERRUPT_NOT_ENABLED	Interrupt not enabled
WD_RESOURCE_OVERLAP	Resource overlap
WD_DEVICE_NOT_FOUND	Device not found
WD_WRONG_UNIQUE_ID	Wrong unique ID
WD_OPERATION_ALREADY_DONE	Operation already done
WD_SET_CONFIGURATION_FAILED	Set configuration operation failed

Status Code	Description
WD_CANT_OBTAIN_PDO	Cannot obtain PDO
WD_TIME_OUT_EXPIRED	Timeout expired
WD_IRP_CANCELED	IRP operation canceled
WD_FAILED_USER_MAPPING	Failed to map in user space
WD_FAILED_KERNEL_MAPPING	Failed to map in kernel space
WD_NO_RESOURCES_ON_DEVICE	No resources on the device
WD_NO_EVENTS	No events
WD_INVALID_PARAMETER	Invalid parameter
WD_INCORRECT_VERSION	Incorrect WinDriver version installed
WD_TRY_AGAIN	Try again
WD_INVALID_IOCTL	Received an invalid IOCTL
WD_OPERATION_FAILED	Operation failed
WD_INVALID_32BIT_APP	Received an invalid 32-bit IOCTL
WD_TOO_MANY_HANDLES	No room to add handle
WD_NO_DEVICE_OBJECT	Driver not installed

Appendix B
Troubleshooting and Support

Please refer to the online WinDriver support page — https://www.jungo.com/st/support/windriver/ — for additional resources for developers, including

- Technical documents
- FAQs
- Samples
- Quick start guides

Appendix C
Purchasing WinDriver

Visit the WinDriver order page on our web site — https://www.jungo.com/st/order_wd/ — to select your WinDriver product(s) and receive a quote. Then fill in the WinDriver order form — available for download from the order page — and send it to Jungo by email or fax (see details in the order form and in the online order page). If you have installed the evaluation version of WinDriver, you can also find the order form in the **WinDriver/docs** directory, or access it via **Start | WinDriver | Order Form** on Windows.

The WinDriver license string will be emailed to you immediately.
Your WinDriver package will be sent to you via courier or registered mail.

Feel free to contact us with any question you may have. For full contact information, visit our contact web page: https://www.jungo.com/st/company/contact-us/.

Appendix D
Additional Documentation

Updated Manuals

The most updated WinDriver user manuals can be found on Jungo's site at https://www.jungo.com/st/support/windriver/.

Version History

If you wish to view WinDriver version history, refer to the WinDriver release notes, available online at https://www.jungo.com/st/support/windriver/wdver/. The release notes include a list of the new features, enhancements and fixes that have been added in each WinDriver version.

Technical Documents

For additional information, refer to the WinDriver Technical Documents database: https://www.jungo.com/st/support/tech_docs_indexes/main_index.html.
This database includes detailed descriptions of WinDriver's features, utilities and APIs and their correct usage, troubleshooting of common problems, useful tips and answers to frequently asked questions.