Version

4

KRF-TECH

WINDRIVER V4

Developer's Guide

WinDriver V4 Developer's Guide For Windows 95, 98, NT3.51, NT4.0, 2000, CE, Linux, Solaris and OS/2.

For ISA/ISA PnP /PCI/USB/EISA/PCMCIA Based Devices

© KRF-Tech
POB 8493 Netanya Zip - 42504 Israel
Phone (USA) 1-877-514-0537 (WorldWide) +972-9-8859365
Fax (USA) 1-877-514-0538 (WorldWide) +972-9-8859366
Web: www.krftech.com

Contents Summery

Table of Contents

Chapter 1

WinDriver Overview

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

Introduction to WinDriver

WinDriver is a device driver development toolkit that dramatically simplifies the very difficult task of developing a device driver. The driver you develop using WinDriver will be source code compatible between all supported operating systems (WinDriver currently supports Windows 95, 98, NT3.51, NT4.00, 2000, CE, Linux, Solaris and OS/2.) It will also be binary compatible between Windows 9x, NT and 2000. Bus architecture support includes PCI /PCMCIA/ ISA /ISA PnP /EISA and USB. WinDriver provides a complete solution for creating high performance drivers, which handle interrupts and I/O at optimal rates.

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

WinDriver supports all PCI bridges, from all vendors. Enhanced support is offered for the PLX / Altera / Galileo / QuickLogic / PLDA / AMCC and V3 PCI chips. A special chapter is dedicated to developers of PCI card drivers who are using PCI chips from these vendors. The last several chapters of this manual explain how to tune your driver code to achieve optimal performance. The "Kernel PlugIn" feature of WinDriver will be thoroughly explained there. This feature allows the developer to write and debug the entire device driver in the User Mode, and later 'drop' performance critical parts of it to the Kernel Mode. This way, your driver development achieves optimal Kernel Mode performance, with User Mode ease of use.

It is recommended to periodically check out KRFTech's web site at www.krftech.com for the latest news about WinDriver and other driver development tools that KRFTech offers.

Good luck with your project!

Background

In protected operating systems (such as Windows, Linux, Solaris and OS/2), a programmer cannot access hardware directly from the application level (the "User Mode") where development work is usually done. Hardware access is allowed only from within the operating system itself (the "Kernel Mode" or "Ring 0"), by software modules called "Device Drivers". In order to access a custom hardware device from the application level, a programmer must do the following:

1. Learn the internals of the operating system he is working on (95/98/NT / CE / Linux / Solaris...)

2. Learn how to write a device driver.

3. Learn new tools for development / debugging in the Kernel Mode (DDK, ETK...).

4. Write the Kernel Mode device driver that does the basic hardware input / output.

5. Write the application in the User Mode, which accesses the hardware through the device driver written in the Kernel Mode.

6. Repeat steps 1-4 for each new operating system on which the code should run.

WinDriver Overview

Easy development - WinDriver enables Windows programmers to create PCI/ ISA /EISA /ISA PnP/PCMCIA /USB based device drivers in an extremely short time. WinDriver allows you to create your driver in the "User Mode" in the familiar environment - using MSDEV, Visual C/C++, Borland, Delphi, Visual Basic or any other Win32 compiler. WinDriver eliminates the need for you to be familiar with the operating system internals, kernel programming or with the DDK or have any device driver knowledge.

Multi Platform - The driver created with WinDriver will run on Windows 95/ 98/NT3.51 /NT4.0 /2000 /CE, Linux, Solaris and OS/2,(NT version available for x86 and Alpha processors), - i.e. write once - run on any of these platforms.

Friendly Wizards - The DriverWizard (included) is a Graphical diagnostics tool that lets you write to, and read from the hardware, before writing a single line of code. With a few clicks of the mouse, the hardware is diagnosed - memory ranges are read, registers are toggled and interrupts are checked. Once the device is operating to your satisfaction, the DriverWizard creates the skeletal driver source code, giving access functions to all of the resources on the hardware.

Kernel Mode Performance - WinDriver's API is optimized for performance. For the drivers that need kernel mode performance, WinDriver offers the "Kernel PlugIn". This powerful feature enables you

to create and debug your code in the user mode, and run the performance critical parts of your code, (such as the interrupt handler, or access to I/O mapped memory ranges), in kernel mode, thereby achieving kernel mode performance (zero performance degradation). This unique feature allows the developer to run the user mode code in the OS kernel without having to learn how the kernel works. When working on Windows CE, there is no need to use the Kernel PlugIn since the CE has no separation between user mode and kernel mode, thus enabling you to easily achieve optimal performance from the user mode code.

How fast can WinDriver go? Using the WinDriver Kernel PlugIn you can expect the same throughput of a custom Kernel Driver. You are confined only by your operating system and hardware limitations. A ballpark figure of the throughput you can reach using the Kernel PlugIn would be more than 100,000 interrupts per second.

"User Mode ease - Kernel Mode performance! "

To conclude -- using WinDriver, all a developer has to do to create an application that accesses the custom hardware is:

1. Start up the DriverWizard, and detect the hardware and its resources.

2. Automatically generate the device driver code from within the Wizard.

3. Call the generated functions from the User Mode application.

The new hardware access application now runs on all Windows platforms (including CE), on Linux, on Solaris and on OS/2 (just recompile).

WinDriver Feature List

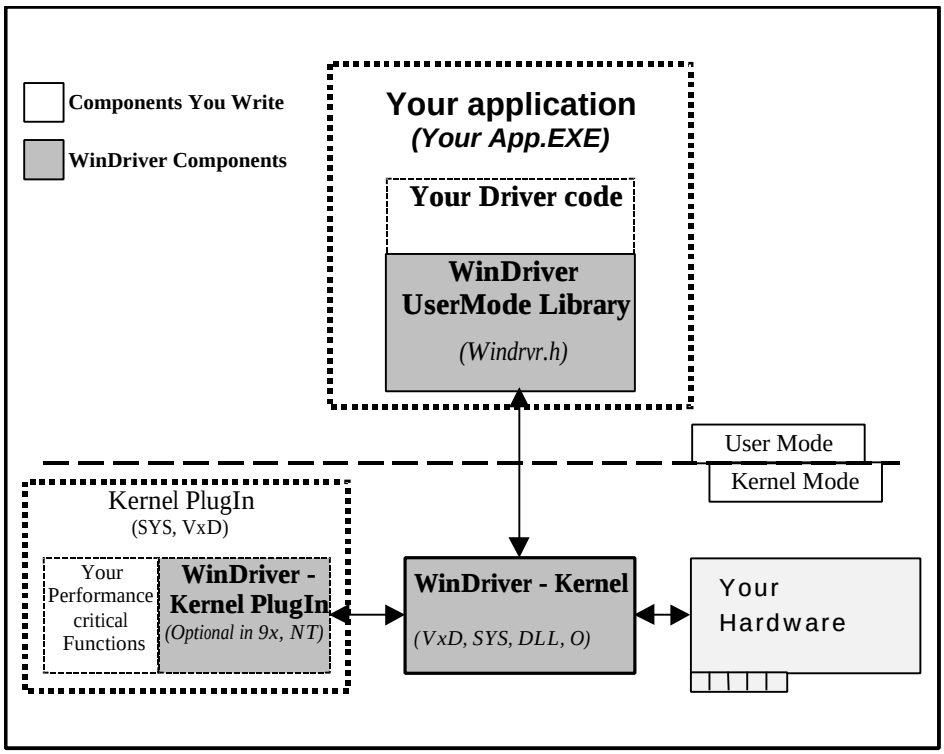
- Easy User Mode driver development.
- Kernel PlugIn for high performance drivers.
- Friendly DriverWizard allows hardware diagnostics without writing a single line of code.
- The DriverWizard automatically generates the driver code for the developer in C\C++ or Delphi (Pascal).
- Supports any PCI/ ISA/ ISA PnP/ EISA/ PCMCIA/ USB chip regardless of manufacturer.
- Enhanced –support for the PLX 9050/ 9054/ 9060/ 9080/ IOP 480, Altera, Galileo, QuickLogic, PLDA, V3 and AMCC PCI bridges, therefore hiding the PCI bridge details from the developer.
- Applications are binary compatible across Windows 9x and Windows NT/2000.
- Applications are source code compatible across Windows 9x, NT, 2000, CE, Linux, Solaris and OS/2.
- WinDriver can be used with common development environments including Visual C++, Borland C++, VB4 and Delphi.
- No DDK, ETK, DDI or any system-level programming knowledge is required.
- Detailed examples in C, Delphi and Visual Basic are included.
- Supports I/O, DMA, Interrupt handling and Access to memory mapped cards.
- Supports Multiple CPU and Multiple PCI-bus platforms.

- Includes Dynamic Driver Loader.
- Comprehensive documentation and help files.
- Six months Free technical support.
- No run time fees or royalties.

Notes:

1. In version 4.2 and below, PCMCIA is only supported in the Windows CE version.
2. VxWorks and DOS will be supported soon by WinDriver.– Please refer to the KRFTech site at http://www.krftech.com for updates.

WinDriver Architecture



For hardware access, your application calls one of the WinDriver functions from the WinDriver User Mode library (windrvr.h). The User Mode library calls the WinDriver Kernel, which accesses the hardware for you, through the native calls of the operating system.

WinDriver's design minimises performance hits on your code, even though it is running in the User Mode. However, some hardware drivers need performance, which is not achievable from the User Mode. This is where WinDriver's edge sharpens - after easily creating and debugging your code in the User Mode, you may 'drop 'the performance critical modules of your code (such as a hardware interrupt handler) to the WinDriver Kernel PlugIn without changing a single line of it. Now, the WinDriver Kernel will call this module from the Kernel Mode, thereby achieving maximal performance. This allows

you to program and debug in the User Mode, and still achieve kernel performance where needed. In Windows CE there is no separation between User Mode and Kernel Mode, therefor you may achieve optimal performance directly from the user mode, eliminating the need to use the Kernel PlugIn in this OS.

What Platforms does WinDriver Support?

WinDriver Supports Windows 95/ 98/ NT/2000 /CE, Linux, Solaris and OS/2 (NT and 2000 versions are also available for the Alpha processor). Same source code will run on All supported platforms. Same executable you write will operate on Windows 9x NT and 2000. Even if your code is meant only for one of these operating systems, using WinDriver will give you the flexibility of moving your driver to the other operating system without changing your code.

Can I Try WinDriver Before I Buy?

Yes! – Evaluation versions of WinDriver for all supported operating systems and buses are available at the KRFTech web site at http://www.krftech.com.

Limitations of the different evaluation versions

All the evaluation versions of WinDriver are fully featured. No function were limited or crippled in any way. The following is a list of the differences between the evaluation versions to the registered ones.

1. At first use of the driver, an 'unregistered' message appears.
2. Delay when reading or writing to hardware through the Wizard.
3. Delay when calling the 'close' function in WinDriver's API.
4. In the Linux, Solaris and CE versions - Driver is operational for 10 minutes after re-starting it.

Evaluation will expire within 30 days of installation.

How Do I Develop My Driver with WinDriver? (Overview)

On Windows 9x, NT and 2000

Start the DriverWizard (See the 'DriverWizard' chapter for details). Diagnose your card, and let DriverWizard generate a skeleton code for you. The code generated by DriverWizard is a diagnostic program, containing functions that read and write to any resource detected or defined (including custom defined registers), and enables and listens to your card interrupts. Modify the code generated by the DriverWizard, to suit your particular application needs.

Run and debug your driver in the User Mode.

If your code contains performance critical sections, improve their performance by turning to the "Improving performance" chapter. This chapter provides a checklist of tune-ups you can make in your code, and shows you how to take the performance critical sections and move them into the "Kernel PlugIn".

On Windows CE

Plug your hardware in to your NT machine. Install the CE ETK on the NT.

Diagnose your hardware via the DriverWizard and then let it generate your driver's skeleton code. Modify this code using Visual C++ to meet your specific needs. Test and debug your code and hardware from the CE emulation running on the NT machine.

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

If your code contains performance critical parts, you may improve their performance by turning to the "Improving performance" chapter in the WinDriver manual.

On Linux and Solaris

When purchasing the Linux or the Solaris version of WinDriver you also receive a license for the Windows version of the DriverWizard. It is recommended to start the development process on your Windows machine, using DriverWizard in the same way described above. After the wizard automatically generates your driver code, you may move the code, (as is), to your Linux\Solaris machine and alter it to perform your specific needs.
If you do not have a Windows machine, you may use the sample files included with WinDriver as skeletons for your driver and change them using the WinDriver API.

What Does the WinDriver Toolkit Include?

- The WinDriver CD
- A printed version of this manual.
- Six months of free technical support (Phone – Fax – Email).
- 45 days of free version upgrades.
- The WinDriver CE license also enables you to run your CE driver code on your NT machine via the CE emulation.
- The WinDriver Linux and Solaris license also enables you to use DriverWizard on your Windows machine to diagnose your hardware and automatically generate your driver skeletal code. You may then compile and run the code created on your Linux\Solaris machine. The code will not run on your Windows machine without WinDriver for Windows licensing.

The following modules are included in your WinDriver toolkit:

WINDRIVER MODULES

- WinDriver Version 4' - (\windriver\include) - *The general-purpose hardware access toolkit.*
- DriverWizard (accessible through Start menu \Programs \WinDriver \DriverWizard) - *A graphical debugging tool which collects debugging information on your driver as it runs. In Linux you may use the console version of this file.*
- WinDriver distribution package (\windriver\redist) - *The files needed to be included in the driver you distribute to your customers.*

- WinDriver Version 4 electronic manual - (accessible through Start menu\Programs\windriver') - *Full WinDriver manual, in pdf (Adobe Acrobat) format.*

- WinDriver Kernel PlugIn '(\windriver\kerplug) - *The files and samples needed to create a Kernel PlugIn 'for WinDriver.*

Utilities:

- PCI_SCAN.EXE (\windriver\util\pci_scan.exe) - *A utility for getting a list of the PCI cards installed and the resources allocated for each one of them.*

- PCI_DUMP.EXE (\windriver\util\pci_dump.exe) - *A utility for getting a dump of all the PCI configuration registers of the PCI cards installed.*

- PCMCIA_SCAN.EXE ((\windriver\util\pcmcia_scan.exe) - *A utility for getting a list of the PCMCIA cards installed and the resources allocated for each one of them*

- USB_DIAG.EXE ((\windriver\util\usb_diag.exe) - *A utility for getting a list of the USB devices installed, the resources allocated for each one of them, and for accessing the USB devices.*

The CE version Also Includes:

- \REDIST\... \X86EMU\WINDRVR_CE_EMU.DLL: *The DLL that communicates with the WinDriver kernel for the X86 HPC emulation mode of Windows CE*

- \REDIST\... \X86EMU\WINDRVR_CE_EMU.LIB: *The import library for linking with WinDriver applications that are compiled for the X86 HPC emulation mode of Windows CE*

WinDriver's SPECIFIC CHIP-SET SUPPORT.

These are APIs that support the major PCI bridge chip-sets, for even faster code development.

- WinDriver PLX APIs (for the 9050, 9054 and 9060/9080 PCI bridges) - \windriver\plx\9050 and ~\9054, ~\9060, ~\9080 respectively.
- WinDriver Galileo APIs (for the Galileo GT64 PCI bridges) - \windriver\galileo\gt64
- WinDriver AMCC APIs (for the AMCC S5933 PCI bridges) - \windriver\amcc
- WinDriver V3 APIs (for the V3 PCI bridges) - \windriver\v3

Each of these directories includes the following directories:

- \lib - *the special chip set API for the PLX/AMCC/V3 chip set, written using the WinDriver API.*
- \xxx_diag - *a sample diagnostics application, which was written using the special library functions available for the these chip sets. This application may be compiled and executed as-is (xxx_diag i.e. p9054_diag.c for the PLX 9054 chip).*

Samples:

Here you will find the source code for the utilities listed above, along with other samples which show how various driver tasks are performed. Find the sample which is closest to the driver you need. Use it to jump-start your driver development process.

- WinDriver samples - (\windriver\samples) - *Samples which demonstrate different common drivers.*
- WinDriver for PLX | GALILEO | AMCC | V3 'samples - *(\p9054_diag ~\p9080_diag etc.) - Source code of the diagnostics applications for the specific chipsets that WinDriver supports.*

Can I Distribute the Driver Created with WinDriver?

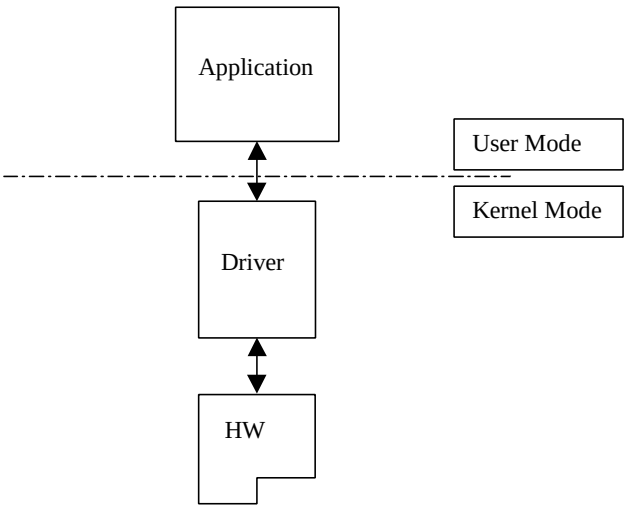
Yes. WinDriver is purchased as a development toolkit, and any device driver created using WinDriver may be distributed royalty free in as many copies as you wish. See the license agreement (\windriver\docs\license.txt) for more details.

Device Driver Overview

The following is an overview of the common types of device driver architectures:

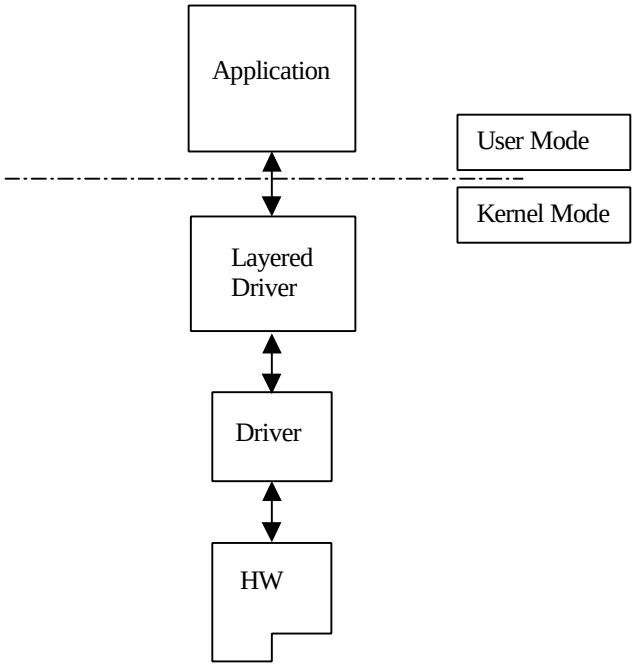
Monolithic drivers:

These are the "classic" device drivers, which are primarily used to drive custom hardware. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through IO control commands - (IOCTLs), and drives the hardware through calling the different DDK functions.



Layered drivers:

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

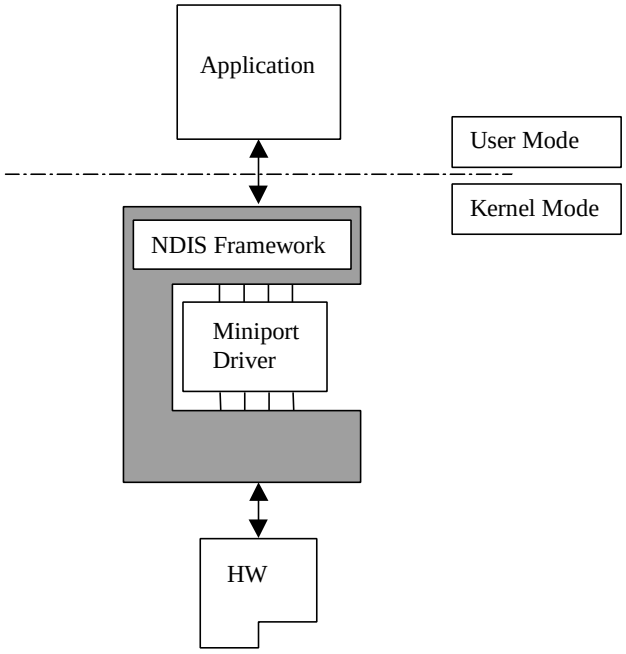


Miniport drivers:

There are classes of device drivers in which much of the code has to do with the functionality of the device, and not with the device's inner workings. In these classes of drivers, these code elements will be duplicated.

The Windows NT/2000, for instance, provides several driver classes (called "ports") which handle the common functionality of their class. It is then up to the user to add only the functionality that has to do with the inner workings of the specific hardware.

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



Matching the right tool for your driver

WinDriver is a tool designed for monolithic type drivers. WinDriver enables you to access your hardware directly from within your Win32 application, without writing a kernel mode device driver. Using WinDriver You may either access your hardware directly from your application (in user mode) or write a DLL you can call from many different applications.

WinDriver also provides a complete solution for high performance drivers. Using WinDriver's Kernel PlugIn, you will be able run your user mode code from the kernel and reach full kernel mode performance without doing any kernel programming. A driver created with WinDriver runs on Windows 95, 98, NT, 2000, CE, Linux, Solaris and OS/2. Typically, a developer without any previous driver knowledge can get a driver running in a matter of a few hours (compared to several weeks with a kernel mode driver).

For Layered or Miniport drivers, kernel programming is necessary. To Simplify this difficult task, KRFTech provides "KernelDriver" - a C++ toolkit which provides classes that encapsulate thousands of lines of kernel code, enabling you to focus on your driver's added-value functionality, instead of your OS internals.

Chapter

2

WinDriver USB Overview

This chapter explores the basic characteristics of the USB bus and introduces WinDriver USB features and architecture.

Introduction to USB

USB, Short for ***U****niversal* ***S****erial* ***B****us, is* a new industry-standard extension to the PC architecture, for attaching peripherals to the computer. The Universal Serial Bus was originally developed in 1995 by leading PC and telecommunication industry companies, such as Intel, Compaq, Microsoft and NEC. The motivation for the development of the USB emerged of several considerations. Among them are the need for an inexpensive and widespread connectivity solution for peripherals in general and for the "Computer Telephony Integration" in particular, the need for easy to use and flexible method of reconfiguring the PC and a solution for adding a large number of external peripherals.

The USB interface meets the above-mentioned needs. A single USB port can be used to connect up to 127 peripheral devices. USB also supports Plug-and-Play installation and hot swapping. USB 1.1 supports both isochronous and asynchronous data transfers and has dual-speed data transfer; 1.5Mbps (Megabit per second) for low-speed USB devices and 12Mbps for high-speed USB devices (much faster than the original serial port). Cables connecting the

device to the PC can be up to five meters (16.4 feet) long. USB also includes built-in power distribution for low power devices, and can provide limited power (maximum: 500mA of current) to devices attached on the bus. Because of these benefits, USB is enjoying broad market acceptance today.

The next USB generation (USB 2.0), which is expected to be finalized in the first quarter of 2000, will support a transfer rate of 480 Mbs (megabits per second) - 40 times faster than USB 1.1. USB 2.0 maintains full compatibility with USB 1.1; therefore same cables, connectors and software interfaces can be used.

Because its relatively low speed USB 1.1 was aimed to replace existing serial ports and parallel ports, keyboard and monitor connectors, telephone/fax/modem adapters, and to be used with keyboards, mice, monitors, printers, low-speed scanners, answering machines and removable hard drives. USB 2.0 implementers will benefit from an additional range of higher performance peripherals, such as video-conferencing cameras, next-generation scanners and printers, and fast storage devices. These devices are anticipated in the market in the second half of 2000.

Feature list

- External connection; easy to use for end user.
- Self-identifying peripherals, automatic mapping of function to driver, and configuration.
- Dynamically attachable and re-configurable peripherals.
- Suitable for device bandwidths ranging from a few Kb/s to several Mb/s.
- Supports isochronous as well as asynchronous transfer types over the same set of wires.
- Supports simultaneous operation of many devices (multiple connections).
- Supports up to 127 devices.
- Guaranteed bandwidth and low latencies; appropriate for telephony, audio, etc. (Isochronous transfer may use almost entire bus bandwidth).
- Flexibility: Supports a wide range of packet sizes and a wide range of data rates.
- Robustness: Error handling mechanism built into protocol, dynamic insertion and removal of devices identified in user observed real-time.

- Synergy with PC industry.
- Optimised for integration in peripheral and host hardware.
- Low-cost implementation, therefore suitable for development of low-cost peripherals.
- Low-cost cables and connectors.
- Uses commodity technologies.
- Built in power management and distribution.

USB Components

USB Host: The USB host computer, where the USB host controller is installed, and where the client software\device driver runs. The USB host controller is the interface between the host and the USB peripherals. The host responsible for detecting attachment and removals of USB devices, managing the control and data flow between the host and the devices, providing power to attached devices and more.

USB Hub: A USB device that enables connecting additional USB devices to a single USB port on the USB host. Hubs on the back plane of the hosts are called root hubs. Other hubs are external hubs.

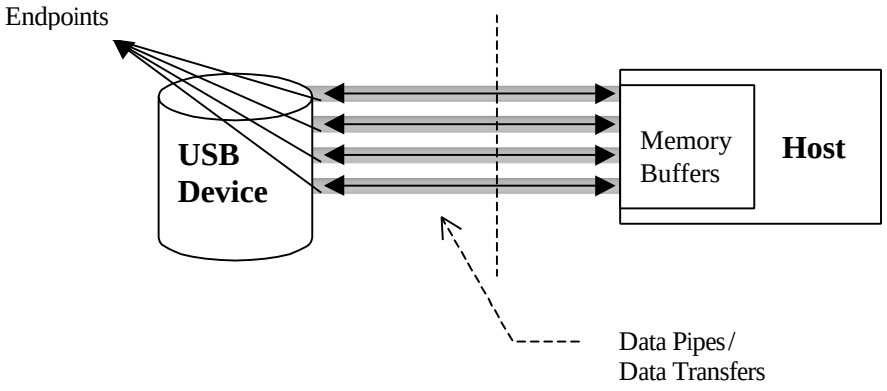
USB Function: The USB device that is able to transmit or receive data or control information over the bus, and provides a function. Compound device provides multiple functions on the USB bus.

Data Flow in USB Devices

During the operation of the USB device, Data flows between the client software and the device. The data is moved between memory buffers of the software on the host and the device, using pipes, which end in endpoints on the device side.

An endpoint is a uniquely identifiable entity on the USB device, which is the source or the terminus of the data that flows from or to the device. Each USB device, logical or physical, has a collection of independent endpoints. Endpoint attributes are their bus access frequency, their bandwidth requirement, their endpoint number, their error handling mechanism, their maximum packet size that the endpoint can transmit or receive, their transfer type and their direction (into the device \out of the device).

Pipes are logical components, representing association between an endpoint on the USB device and software on the host. The data moved to and from the device through 'a pipe. Pipe can be of two modes: stream pipe and message pipe, according to the type of data transfer used in that pipe. Pipes, sending data in interrupt, bulk or isochronous types are stream pipes, while control transfer type is supported by the message pipes. The different USB transfer types are discussed below.



USB data transfer types

The USB device (function) communicates with the host by transferring data through a pipe between a memory buffer on the host and an endpoint on the device. The USB provides different transfer types, that best suit the service required by the device and by the software. The transfer type of a specific endpoint is determined in the endpoint descriptor.

There are four different types of data transfer within the USB specification:

Control Transfer: The control transfer is mainly intended to support configuration, command and status operations between the software on the host and the device. Each USB device has at least one control pipe (default pipe), which provide access to the configuration, status and control information. The control pipe is a bi-directional pipe. The control transfer is a bursty, non-periodic communication. Control transfer has a robust error detection, recovery and retransmission mechanism and retries are made with no involvement of the driver. Control transfer is used by low speed and high-speed devices.

Isochronous Transfer: A type usually used for time dependent information, such as multimedia streams and telephony. The transfer is periodic and continuous. The isochronous pipe is uni-directional and a certain endpoint can either transmit or receive information. For a bi-directional isochronous communication there's a need to use two isochronous pipes, one in each direction. The USB guarantees the isochronous transfer access to the USB bandwidth (that is reserves the required amount of bytes of the USB frame) with bounded latency and guarantees the data transfer rate through the pipe unless there is less data transmitted. Up to 90% of the USB frame can be allocated to periodic transfers (isochronous and interrupt transfers). If, during configuration, there is no sufficient bus time available for the requester isochronous pipe, the configuration is not established. Since time is more important than correctness in these types of transfers, no retries are made in case of error in the data transfer, though the data receiver can determine error occurred on the bus. Isochronous transfer can be used only by high-speed devices.

Interrupt Transfer: Interrupt transfer is intended for devices that send and receive small amount of data, in low frequency or in an asynchronous time

frame. An interrupt transfer type guarantees a maximum service period and a retry of delivery to be attempted in the next period, in case of an error on the bus. The interrupt pipe, like the isochronous pipe is uni-directional. The bus access time period (1-255ms for high-speed devices and 10-255ms for low-speed devices) is specified by the endpoint of the interrupt pipe. Although the host and the device can count only on the time period indicated by the endpoint, the system can provide a shorter period up to 1 ms.

Bulk Transfer: Bulk transfer is a non-periodic, large packet, bursty communication. Bulk transfer typically supports devices that transfer large amount of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners. The bulk transfer allows access to bus on availability basis, guarantees the data transfer but not the latency and provides error-check mechanism with retries attempts. If part of the USB bandwidth is not being used for other transfers, system will use it for bulk transfer. Like previous stream pipes (isochronous and interrupt) the bulk pipe is also uni-directional. Bulk transfer can be used only by high-speed devices.

USB Configuration

Before the USB function (or functions in a compound device) can be operated, the device must be configured. The host does the configuring, by acquiring the configuration information from the USB device. USB devices report their attributes by descriptors. Descriptor is the defined structure and format in which the data is transferred. A complete description of the USB descriptors can be found in Chapter 9 of the USB Specification (See www.usb.org for the full specification).

It is best to view the USB descriptors as a hierarchic structure of four levels:

- The Device level
- The configuration level
- The interface level (this level may include an optional sub-level called alternate settings)
- The endpoint level.

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

Device: In the top level is the 'device descriptor', which includes general information about the USB device, that is global information for all of the device configurations. The device descriptor describes, among other things, the device class (USB devices are divided into device classes, such as HID devices, hubs, locator devices etc.), subclass, protocol code, Vendor ID, Device ID and more. Each USB device has one device descriptor.

Configuration: A USB device has one or more configuration descriptors, which describe the number of interfaces grouped in each configuration and power attributes of the configuration (such self-powered, remote Wakeup, maximum power consumption and more).. At a given time, only one configuration is loaded. An example of different configurations of same device may be an ISDN adapter, where one configuration presents it with a single interface of 128KB/s and a second configuration with two interfaces of 64KB/s.

Interface: The interface is a related set of endpoints that present a specific functionality or feature of the device. Each interface may operate independently. The interface descriptor describes the number of the interface, number of endpoints used by this interface, and the interface specific class, subclass and protocol values when the interface operates independently. An interface may also have alternate settings. The alternate settings allow the endpoints or their characteristics to be varied after the device was configured.

Endpoint: The lowest level is the endpoint descriptor that provides the host with the information regarding the data transfer type of this endpoint and the bandwidth of each endpoint (the maximum packet size of the specific endpoint). For isochronous endpoints, this value is used to reserve the bus time required for the data transfer. Other endpoints 'attributes are their bus access frequency, their endpoint number, their error handling mechanism, and their direction

Seems complicated? Not at all! WinDriver automates the USB configuration process. The included DriverWizard and USB diagnostic

application scan the USB bus, detect all USB devices and their different configurations, interfaces, settings and endpoints, and enable the developer to pick the desired configuration before starting the driver development. WinDriver also identifies the endpoint transfer type as determined in the endpoint descriptor. The driver created with WinDriver will contain all configuring information acquired at this early stage.

WinDriver USB

WinDriver USB enables developers to quickly develop high performance drivers for USB based devices, without having to learn the USB specs or the OS internals. Using WinDriver USB, developers can create USB drivers without having to use the DDK (Microsoft Driver Development Kit), and without having to be familiar with Microsoft's WDM (Win32 Driver Module).

The driver code developed with WinDriver USB is binary compatible between Windows 2000 and Windows 98 and supports Microsoft's WDM interface. The source code will be code-compatible among all other operating systems, supported by WinDriver USB. For up to date information regarding operating systems currently supported by WinDriver USB check out KRFTech's web site at www.krftech.com.

WinDriver USB encapsulates the USB specification and architecture, letting you focus on your application logic. WinDriver USB features the DriverWizard, with which you can detect your hardware, configure it and test it before writing a single line of code. The DriverWizard will lead you through the configuration procedure first, enable you to choose the desirable configuration, interface and alternate setting through a friendly graphical user interface. After detecting and configuring your USB device, you can then test it, listen to pipes, write and read packets and validate all your hardware resources function as expected. WinDriver USB is a generic tool kit, which supports all USB devices, from all vendors and with all kind of configurations.

After your hardware is diagnosed, the DriverWizard will automatically generate your device driver source code in C or in Delphi. WinDriver USB provides user-mode APIs to your hardware, which you can call from within your application. WinDriver USB API is specific for your USB device and includes

USB unique operations such as reset-pipe and reset-device. Along with the device API, WinDriver USB creates a diagnostic application, which just need to be compiled and run. You can use this application as your skeletal driver to jump-start your development cycle. If you are a VB programmer, you will find all WinDriver USB API supported for you also in VB, giving you all you need to develop your driver in VB.

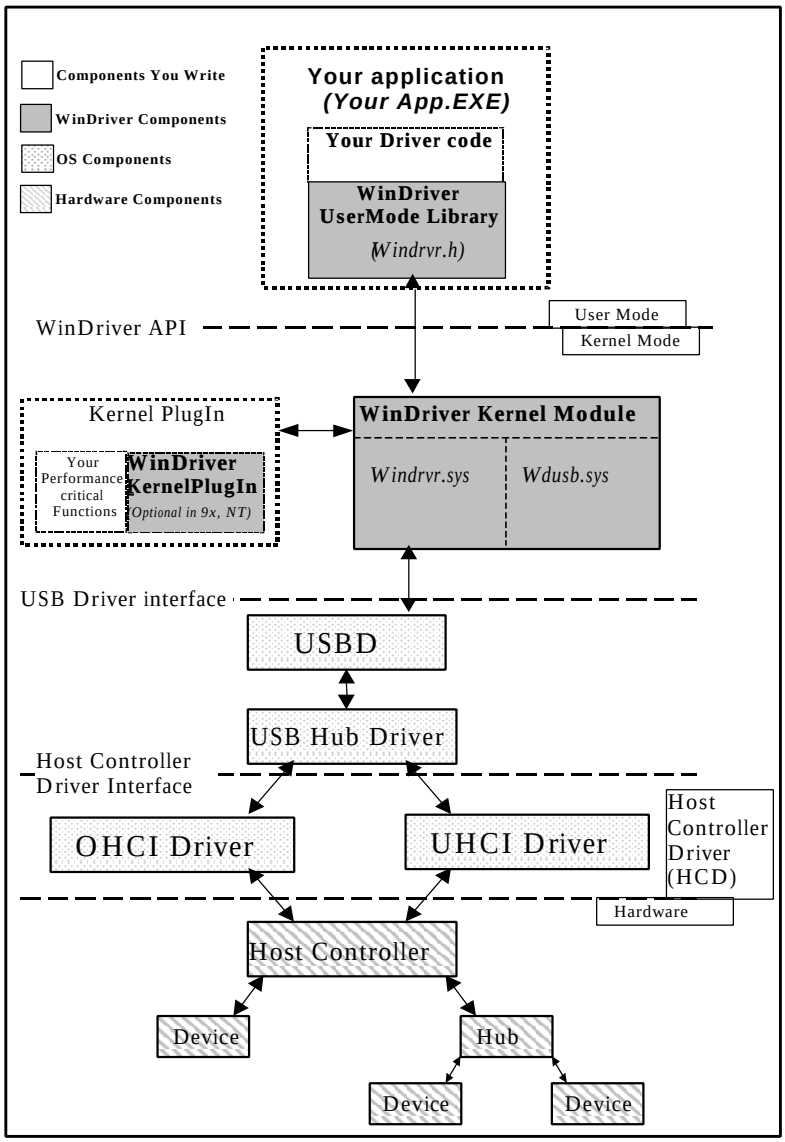
The DriverWizard also automates the creation of an .INF file where needed. The .INF file is a text file used by the Plug-&-Play mechanisms of Windows 95/98 and Windows 2000 to load the driver for the newly installed hardware or to replace an existing driver. The .INF file includes all necessary information about the device(s) and the files to be installed. .INF files are required for hardware that identifies itself, such as USB and PCI. In some cases, the .INF file of your specific device is included in the .INF files that are shipped with the operating system. In other cases, you will need to create an .INF file for your device. WinDriver automates this process for you. More information on how to create your own .INF file with the DriverWizard can be found in chapter 4 "The DriverWizard". Installation instructions of .INF file can be found in chapter 21 "Distributing your Driver".

Using WinDriver USB, all development is done in the user mode, using the familiar development and debugging tools and your favourite compiler (such as MSDEV, VC++, Cbuilder, Borland Delphi).

WinDriver USB API is designed to give you optimised performance. In the cases where native kernel mode performance is needed, use WinDriver USB's unique "KernelPlugIn" feature (included). This powerful feature enables you to write and debug your code in the user mode, and then simply 'drop' it into the Kernel PlugIn for kernel mode execution. This unique architecture enables you to achieve maximum performance with user mode ease of use.

All other WinDriver USB features can be found in the WinDriver feature list in the first chapter "WinDriver Overview".

WinDriver USB Architecture



To access your hardware, your application calls the required WinDriver USB API function from the WinDriver User Mode Library (windrvr.h). The User

Mode Library calls the WinDriver Kernel Module. The WinDriver Kernel Module is comprised of windrvr.sys and wdusb.sys. The WinDriver Kernel Module accesses your USB device resources through the native operating system calls.

There are two layers responsible to abstract the USB device to the USB device driver: The upper one is the USB Driver layer (including the USB Driver (USBD) and USB Hub Driver) and the lower one is the host controller driver layer (HCD). The division of duties between the HCD and USBD is not defined, and is operating system dependent. Both HCD and USBD are software interfaces and components of the operating system, where the HCD layer represents a lower level of abstraction.

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

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

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

WinDriver USB abstracts the configuration procedure and hardware access described above for the developer. With WinDriver USB API, developers can do all hardware-related operation without having to master the lower levels of implementing these activities.

What drivers can I write with WinDriver USB?

Almost all monolithic drivers (drivers that needs to access specific USB devices), can be written with WinDriver USB. In cases where a "standard" driver needs to be written (e.g. NDIS driver, SCSI driver, Display driver, USB to Serial port drivers, USB layered drivers, etc.)., use KernelDriver USB (also from KRFTech)

For quicker development time, prefer WinDriver USB over KernelDriver USB where possible.

Chapter

3

Installation and Setup

This chapter takes you through the WinDriver installation process, and shows you how to check that your WinDriver is properly installed.

Systems Requirements

For Windows 95 / 98

1. An x86 processor
2. Any 32bit development environment supporting C, VB or Delphi.

For Windows NT/2000

1. An x86 or Alpha NT processor.
2. Any 32bit development environment supporting C, VB or Delphi.

For Windows CE

1. Windows NT Workstation 4.0 host development platform
2. Microsoft Developer Studio 97 including:
   - Microsoft Visual C++ V5.0 or higher
   - Windows CE Platform SDK

***If you are using a commercial Windows CE handheld Computer like the HP Jornada or the Sharp Mobilon, you will need the following items in addition*:**

3. Your handheld computer.

4. Serial PC link cable for communication via Windows CE Services (This cable is normally custom manufactured and supplied by the manufacturer of the handheld computer. Do not attempt to use different cables for this purpose.)

***If you are using an X86 PC or a commercial target board like the Hitachi ODO, you will need the following items in addition*:**

1. Your target platform.

2. The Windows CE Embedded Toolkit for Visual C++ (ETK) V2.10, or Platform Builder V2.11 and above. IF you have the ETK V2.0, you should upgrade to 2.1 via the ETK 2.1 Enhancement Pack (available at a retail price of US $14.95 with major online resellers as of March 99) and upgrade your installation before installing WinDriver CE.

3. A serial null modem cable for debugging. A null modem cable can be purchased from a computer hardware store and wired by hand using a soldering iron (please see the Appendix of this manual for the pinout diagram of a null modem cable, and for information on purchasing such a cable)

4. A custom parallel port cable for downloading of the OS image and dynamic loading of WinDriver CE.

This procedure is explained in great detail in the online documentation of the Windows CE ETK and Platform Builder.

For Linux

1. An x86 processor

2. Any 32bit development environment supporting C (such as GCC).

Installing WinDriver

The WinDriver CD contains all versions of WinDriver for all the different operating systems. The CD's root directory contains the Windows 9x and NT/2000 version. This will automatically start upon entering the CD into your CD drive in your 9x or NT machine. The other versions of WinDriver are located in subdirectories i.e. \Alpha, \Linux, \Wince and so on. Following you will find installation instructions for the registered versions of WinDriver.

Installing WinDriver for Windows 9x / NT / 2000

1. Insert the WinDriver CD to your driver. (When installing WinDriver by downloading it from KRFTech's web site instead of using WinDriver CD – Double click the downloaded WinDriver file (wdxxx.exe) in your download directory, and go to step 3).

2. Wait a few seconds until the installation program automatically starts. If for some reason it does not start automatically, double click the file "Wdxxx.exe" (where "xxx" is the version number). Press the "Install WinDriver" button.

3. Read the license text carefully, and press 'YES' if you accept its terms.

Registered users:

4. Choose 'Install registered version' when prompted for which version to install

5. In the "Setup type" screen, choose one of the following:

   - Typical - To install all WinDriver modules. (Generic WinDriver toolkit + specific chip set APIs).

   - Minimal - To install only the generic WinDriver toolkit.

- Custom - To choose which modules of the WinDriver to install. You may choose which APIs will be installed.

6. You will now be prompted for an 8-digit password to continue the installation. Type in the password you received when purchasing WinDriver. Take care when entering the password. The installation will fail if the wrong password is written here. Note that the password is case sensitive.

7. After completing the set-up, It is recommended to reboot your computer.

8. Activate the DriverWizard from Start | Programs | WinDriver | DriverWizard. Enter the Register WinDriver option from the File menu and insert your license string there.

9. To activate source code you have developed in the evaluation version simply follow the instructions in \windriver\redist\register\register.txt.

Installing WinDriver CE

IF YOU ARE INSTALLING WINDRIVER CE FOR A HANDHELD COMPUTER:

1. Insert the WinDriver CD into your NT machine CD drive.

2. Exit from the auto installation and double click the "Cd_setup.exe" file from the \Wince directory inside the CD. This will copy all needed WinDriver files to your development platform (NT).

3. Copy the WinDriver CE kernel file (\windriver\redist\register\TARGET_CPU\windrvr.dll) to the \WINDOWS subdirectory of your HPC.

4. Use the Windows CE Remote Registry Editor tool or the Pocket Registry Editor on your HPC to modify your registry so that the WinDriver CE kernel is loaded appropriately. The file \windriver\samples\wince_install\PROJECT_WD.REG contains the appropriate changes to be made.

5. Restart your HPC. The WinDriver CE kernel will be automatically loaded. You will have to do a **warm RESET** rather than just Suspend/Resume. You should look for a button labelled RESET on your HPC. On the HP 3xx/6xx series, this button can be found under the reserve battery cover.

6. Compile and run the sample programs (see the section on CHECKING YOUR INSTALLATION below) to make sure that that WinDriver CE is loaded and us functioning correctly.

If You Are Using ETK / Platform Builder And Installing WinDriver CE For CE PC

It is highly recommended that you read the ETK documentation and understand the Windows CE and device driver integration procedure before you perform the following installation procedure:

1. Repeat steps 1-2 in above.

2. Open an ETK Build Command Window using the MAXALL project on your NT development platform.

3. Copy the WinDriver CE kernel file (\windriver\redist\register\TARGET_CPU\windrvr.dll) to the %_FLATRELEASEDIR% subdirectory on your development platform. This environment variable is set by the WinCE ETK and may be D:\WINCE210\RELEASE for example.

4. Append the contents of the file PROJECT_WD.REG to the file \windriver\samples\wince_install\PROJECT_WD.REG in the %_FLATRELEASEDIR% subdirectory.

5. Append the contents of the file \windriver\samples\wince_install\PROJECT_WD.BIB to the file PROJECT.BIB in the %_FLATRELEASEDIR% subdirectory. This step is only necessary if you want the WinDriver CE kernel file (WINDRVR.DLL) to be part of the WinCE image (NK.BIN) permanently. This would be the case if you were transferring the file to your target platform using a floppy disk. If you prefer to have the file WINDRVR.DLL loaded on demand via the CESH/PPSH services, you need not carry out this step until you build a permanent kernel.

6. Use the WinCE ETK tool MAKEIMG.EXE to generate a new WinCE kernel called NK.BIN. Transfer this kernel to the target platform using the PPSH/CESH service or via a floppy disk.

7. Restart your target CE platform. The WinDriver CE kernel will be automatically loaded.

8. Compile and run the sample programs (see the section on CHECKING YOUR INSTALLATION below) to make sure that that WinDriver CE is loaded and us functioning correctly.

If You Will Be Testing Your Applications on The X86 HPC Emulation on Windows NT

1. Repeat step 1-2 in the first CE installation set of instructions.

2. Compile and run one of the sample programs making sure to choose the X86EMU target to make sure that it works correctly.

Installing WinDriver for Linux

Installing WinDriver for Linux on your Linux machine

1. Insert the WinDriver CD into your Linux machine CD drive.

2. Create a directory /usr/bin/windriver

   ```
   /usr/bin> mkdir windriver
   ```

3. Make windriver your active directory

   ```
   /usr/bin> cd windriver
   ```

4. Extract the file wdxxxln.tgz (where xxx is the version number)

   ```
   /usr/bin/windriver> tar –xvzf /mnt/cdrom/LINUX/wdxxxln.tgz
   ```

5. Install WinDriver
   ```
   /usr/bin/windriver> make install
   ```

The following steps are for Registered Users only

6. Change directory to /windriver/redist/register/

   ```
   /usr/bin/windriver> cd /redist/register
   ```

7. Extract the file wdxxxreg.zip using the password you have received with the WinDriver package.

   ```
   /usr/bin/windriver/redist/register> unzip wdxxxreg.zip
   ```

8. Change directory to windriver/redist/

   ```
   /usr/bin/windriver/redist/register/> cd ..
   ```

9. Remove the evaluation module.

   ```
   /usr/bin/windriver/redist/> /sbin/rmmod windrvr
   ```

10. Clean the evaluation version module directory

    ```
    /usr/bin/windriver/redist/> make clean
    ```

11. Install registered version

    ```
    /usr/bin/windriver/redist/> make install IS_REGISTERED=1
    ```

12. To activate source code you have developed in the evaluation version simply follow the instructions in \windriver\redist\register\register.txt.

Installing the DriverWizard on your Windows machine

1. Insert the WinDriver CD into your Windows machine CD drive.

2. Follow steps 2-9 of the Windows installation instructions (above).

Installing WinDriver for Solaris

Installing WinDriver for Solaris on your Linux machine

1. Insert your CD into your Solaris machine CD drive.

2. Create a directory /usr/bin/windriver:
   /usr/bin> mkdir windriver

3. Make windriver your active directory :
   /usr/bin> cd windriver

4. Copy the file WDXXXSLS.tgz (Sparc) or WDXXXSL.tgz (Intel) from it's directory (the CD or your download directory) to the current directory:
   /usr/bin/windriver> cp file_directory/WDXXXSLS.tgz ./
   (XXX – WinDriver's version number).

5. Extract the file:
   /usr/bin/windriver> gunzip -c WDXXXSLS.tgz | tar -xvf -

6. Installing WinDriver for Solaris:
   /usr/bin/windriver>./ install_windrvr

The following steps are for Registered Users only:

1. Change directory to /windriver/redist/register/:
   /usr/bin/windriver> cd /redist/register

2. Extract the file found in wdXXXreg.zip and enter the password you have received with the WinDriver package:
   /usr/bin/windriver/redist/register> unzip wdXXXreg.zip

3. Replace the evaluation WinDriver kernel (windrvr) with the registered version you have extracted in item 6 above:
   copy platform/kernel/drv

4. To activate source code you have developed in the evaluation version simply follow the instructions in /windriver/redist/register/register.txt

Installing the DriverWizard on your Windows machine

1. Insert the WinDriver CD into your Windows machine CD drive.

2. Follow steps 2-9 of the Windows installation instructions (above).

Installing WinDriver Alpha NT (NT for the Alpha processor)

1. Insert the WinDriver CD into your NT machine CD drive.

2. Unzip the file "WdXXXaxp.zip" found on \Alpha.

3. Activate the DriverWizard from Start | Programs | WinDriver | DriverWizard. Enter the Register WinDriver option from the File menu and insert your license string there.

4. To activate source code you have developed in the evaluation version simply follow the instructions in \windriver\redist\register\register.txt

Checking Your Installation

On your Windows machine (Including Alpha processors):

1. Start the DriverWizard by choosing 'Programs | WinDriver | DriverWizard' from the start menu.

Registered users:

2. Make sure that your WinDriver license is installed (see the 'Installing WinDriver' section). If you are an evaluation version user, you do not need to install a license.

3. For PCI cards - Insert your card into the PCI bus, and check that the DriverWizard detects it.

4. For ISA cards - Insert your card into the ISA bus, Configure the DriverWizard with your card's resources and try to read / write to the card using the DriverWizard.

On your Windows CE machine:

1. Start the DriverWizard on your NT machine by choosing 'Programs | WinDriver | DriverWizard' from the start menu.

2. Make sure that your WinDriver license is installed (see the 'Installing WinDriver' section). If you are an evaluation version user, you do not need to install a license.

3. For PCI cards - Insert your card into the PCI bus, and check that the DriverWizard detects it.

4. For ISA cards - Insert your card into the ISA bus, Configure the DriverWizard with your card's resources and try to read / write to the card using the DriverWizard.

5. Activate Visual C++ for CE and load one of the WinDriver samples (e.g. \windriver\samples\speaker\speaker.dsw)

6. Select the target platform as X86em from the VisualC++ WCE Configuration Toolbar.

7. Compile and run the speaker sample. The NT speaker should be activated from within the CE emulation environment.

On your Linux machine:

1. Run the precompiled speaker sample found in \windriver\samples\speaker\LINUX\speaker

2. If the sample program works – you have installed you WinDriver for Linux properly.

Chapter 4

The DriverWizard

DriverWizard - An Overview

The DriverWizard (included in the WinDriver toolkit) is a Windows-based diagnostics tool that lets you write to and read from the hardware, before writing a single line of code. The hardware is diagnosed through a Windows interface - memory ranges are read, registers are toggled and interrupts are checked.

Once the card is operating to your satisfaction, DriverWizard creates the skeletal driver source code, creating functions accessing all your hardware resources including custom defined registers. The DriverWizard generates an API, which is specific to your hardware. This specific API is implemented by calling the WinDriver generic API. For example, WinDriver's API contains a function called WD_Transfer() for exchanging data with your hardware. The Wizard might generate a more specific function such as MyCard_ReadStatusRegister() (where 'status register' is a register you have defined on your hardware).

It is recommended to start your driver development by letting the DriverWizard generate the driver code for you. If you are developing a driver for a PLX / Altera / Galileo / AMCC / QuickLogic / PLDA / V3 based card, it

is recommended to move straight to the "Enhanced Support for specific PCI" chapter, and to start your driver development from there.

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

1. **Hardware diagnostics:** After the hardware has been built, insert the hardware into the PCI / ISA / PCMCIA bus or attach your new USB device to the USB port in your machine, and use the DriverWizard to check that the hardware is performing as expected.

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

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

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

2. A 32 bit diagnostics program, in console mode with which you can diagnose your device. This application utilises the special library functions, (described above), which were created for your device by the DriverWizard. Use this diagnostics program as the skeleton for your device driver.

3. A project workspace which you can use to automatically load all the above project information and files into your development environment. In WinDriver Linux and WinDriver Solaris the driver wizard generate the makefile for the relevant operating system.

Wizard Walkthrough

Following are the five steps in using the DriverWizard:

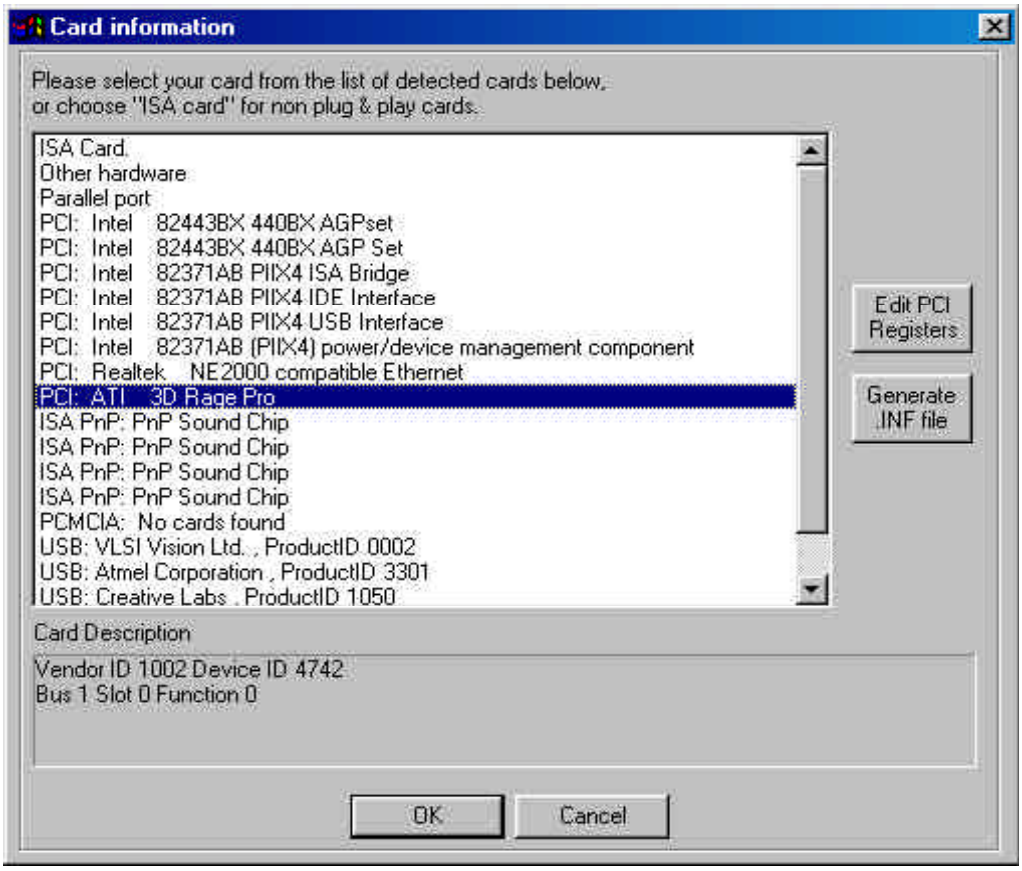
1. Insert your card in your hardware bus (PCI / ISA / PCMCIA) or attach your USB device to the USB port in your machine.

2. Run the Wizard

   - Click /Start/WinDriver/DriverWizard from the start menu or doubleclick the DriverWizard icon on your desktop.

   - The start-up dialog will appear. Click your mouse to start the DriverWizard. If you are using an evaluation copy of WinDriver, you will be notified of the time left for your evaluation period.

   - Choose your PnP device from the list of devices detected by DriverWizard or configure it manually (for non PnP cards like ISA)



- In some cases the DriverWizard will notify you of the need to generate an INF file, in order to continue your hardware diagnostics. (If after pressing the OK button no message popped up – move to the next step).

  To generate an INF file simply press the 'Generate .INF file' button in the 'Card information' screen and save the generated .INF file (the default name given to the file by the Wizard is 'my_device.inf').

  To install the .INF file follow the instruction displayed by the DriverWizard or refer to the 'Creating INF file' section in Chapter 21 'Distributing your Driver' of this manual.

Why should I create an INF file?

1. To stop the 'new hardware wizard' of the Windows operating system from popping up after boot.

2. In some cases the OS doesn't initialize the PCI configuration registers in Win98 without an INF file. In these cases you will not be able to diagnose your hardware with the DriverWizard until creating the INF file.

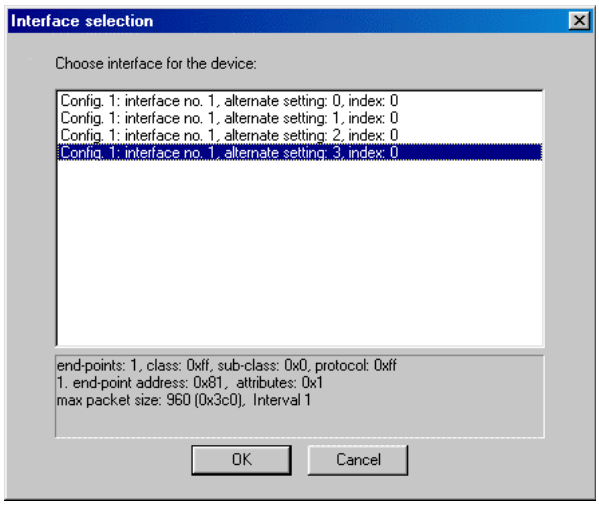
3. In some cases the OS doesn't assign physical address to USB devices without an INF file. In these cases you will not be able to diagnose your USB device with the DriverWizard until creating the INF file..

4. To load the new driver created for the card\device. Creating an INF file is required whenever developing a new driver for the hardware.

5. To replace the existing driver with a new one.

3. Configure your USB device (developers working with ISA/PCI/ISA PnP/PCMCIA cards should skip this step):

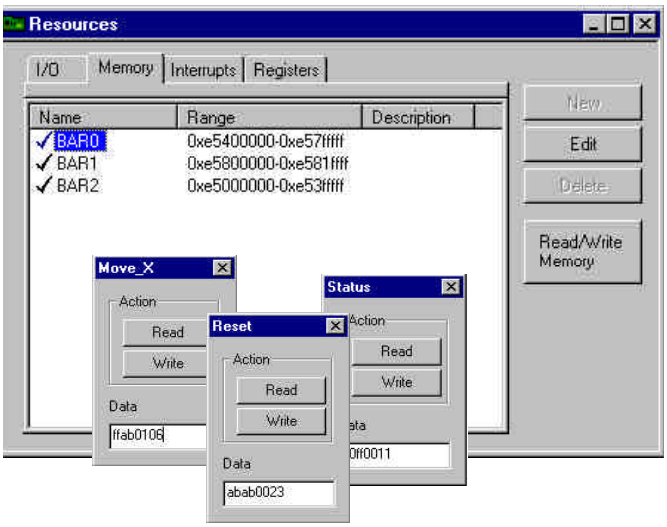
   - Choose the desired configuration\interface \settings from the list.(Note: The Wizard reads all the supported devices Interface and alternate settings and display them. For USB devices with only one interface configured, the wizard automatically selects the detected interface and the fnterface selection 'screen will not be displayed).



4. Diagnose your device:

   - Test your card's I/O, memory ranges, registers and interrupts.
   - Test the USB device's pipes.
   - All of your activity will be logged on the DriverWizard logger, so that you may later analyze your testing.
   - Make sure your card is performing as expected.

A PCI diagnostic screen:

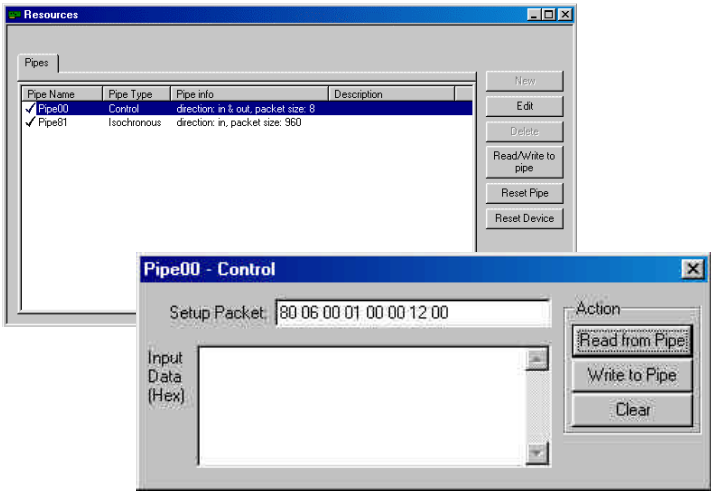


- For USB testing: The Wizard shows the pipe detected according to the selected configuration.
- In order to perform data transfers follow the below steps:
  - Select the desired pipe.
  - For control pipe (a bi-directional pipe) - press 'read/write to pipe'. New dialog will pop up where you enter a setup packet and for 'writing operation' also input data. The setup packet should be 8 bytes long (little endian) and according to the USB spesfication parameters (bmRequestType, bRequest, wValue, wIndex, wLength).
  - For input pipe (moves data from device to the host) - press 'listen to pipe' To succesfully accomplish this operation with devices other then HID, first you need to verify that the device sends data to the host. If no data is being sent, after 'listening' for a short period of time the wizard will

notify you 'Transfer failed').
To stop reading press 'stop listen to pipe'.

- For output pipe (host to device) - press 'write to pipe'. New dialog will pop up asking you to enter the data to write. The Wizard logger will contain the outcome of the opreation

A USB diagnostic screen:

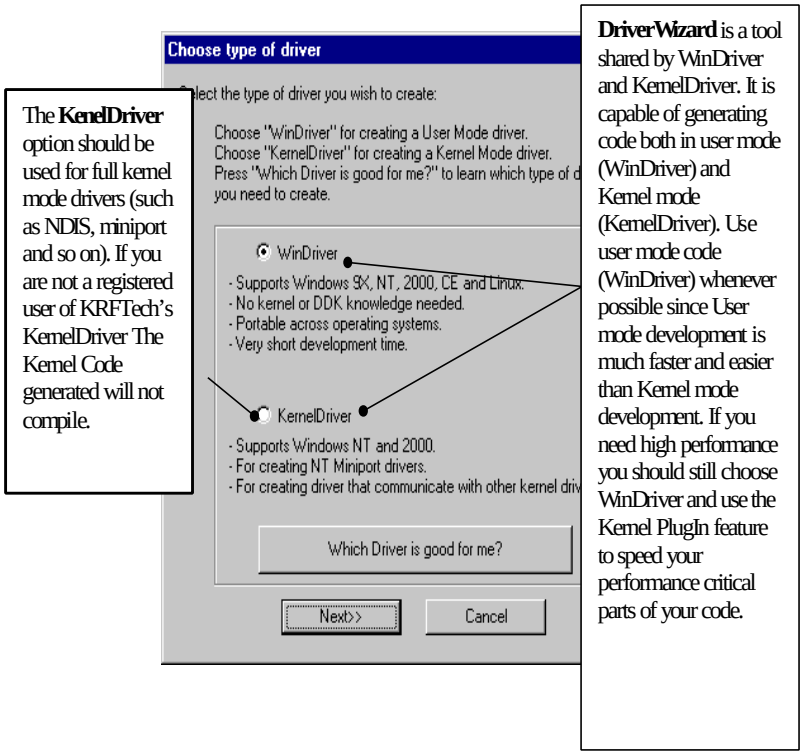


5. Generate the skeletal driver code.

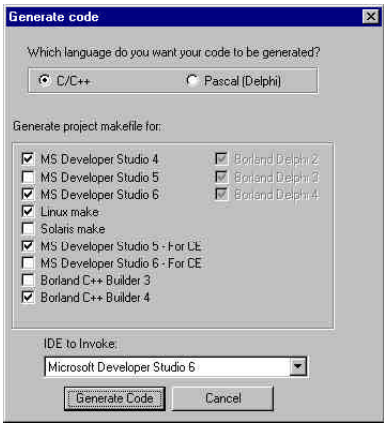
- Choose the 'Generate code' option from the Build menu.

- Select the WinDriver option on the 'Choose type of driver' screen. Selecting the KernelDriver option will generate kernel source code designed for full kernel mode drivers – See the KernelDriver documentation or the KRFTech web site for

more details (Note: this screen appears only when both WinDriver and KrnelDriver are installed on your machine.



- On the following screen, choose the language in which the code will be generated , and choose your desired development environment for the various operating systems.



- Press the 'generate code' button at the bottom of the screen.

6. Compile and run the generated code.

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

   - The source code that DriverWizard creates can be compiled with any Win32 compiler immediately, and will run on ALL Supported platforms (9x, NT, 2000, CE, Linux, Solaris and OS/2) without needing modification!

DriverWizard Notes

Sharing a Resource

When two or more drivers want to share the same resource, you must define that resource as 'shared'.

To define a resource as shared:

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

(Note: The default for new defined interrupt is 'shared'. If you wish to define it as unshared interrupt, follow steps 1-2 and select 'Unshared' from the menu in step 3).

Disabling a Resource

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

Disabling a resource:

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

DriverWizard Logger

The DriverWizard Logger is the blank window that opens up along with the device resources dialog when opening a new project.

The logger keeps track of all your input / output in the diagnostics stage, so that the developer may analyze his device's physical performance at a later time.

It is possible to save the log for future reference.

When saving the project, your log is saved as well. Each log is associated with one project.

Automatic Code Generation

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

Step One – Generating your code.
Choose 'Generate Code' from the 'Build' menu (shown in step 5 above).

DriverWizard will generate the source code for your driver, and place it along with the project file (xxx.wdp where xxx is your project name). The files are saved in a directory the DriverWizard creates for every development environment and operating system chosen in the 'Generate Code' screen.

In the source code directory you now have a new 'xxxlib.h' file which states the interface for the new functions that DriverWizard created for you, and the source of these functions 'xxxlib.c', where your device specific API is implemented. In addition, you will find the sample main() function in the file 'xxxdiag.c'.

The code generated by DriverWizard is composed of the following elements and files ('xxx' – your project name):

1. Library functions for accessing each element of your card's resources (Memory ranges, I/O ranges, registers, interrupts or the USB pipes).
   xxx_lib.c – Here you can find the implementation of your hardware specific API, (found in xxx_lib.h), using the regular WinDriver API.
   xxx_lib.h – This is the header file of the diagnostic program. Here you can find all your hardware specific API created by the DriverWizard. You should include this file in your source code to use this API.

2. A general PCI utility library

   A diagnostics program, which is a console application with which you can diagnose your card. This application utilizes the special library functions, which were created for your device by the Wizard. Use this diagnostics program as the skeleton for your device driver.
   xxx_diag.c – This is the source code of the diagnostics program the DriverWizard creates.

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

After creating your code, compile it with your favourite Win32 compiler, and see it works!

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

Step 2 - Compiling the generated code

For Windows 9X, NT, 2000 and CE (Using MSDEV)

For Windows platforms, DriverWizard generates the project files (for MSDEV 4, 5 and 6 ,C Builder and Delphi 2, 3, 4). After the code generation, the chosen IDE (Integrated development environment) will launch automatically. You may immediately compile and run the generated code.

For Linux and Solaris

The wizard creates a makefile for your project.

Compile the source code using the makefile generated by the wizard.

For Other OSs or IDEs

Create a new project in your IDE (Integrated development environment)

Include the source files created by the DriverWizard into your project.

Compile and run the project.

The project contains a working example of the custom functions that DriverWizard created for you. Use this example to create the functionality you want.

Chapter

5

Creating Your Driver

This chapter takes you through the WinDriver driver development cycle.

> **IMPORTANT NOTE:**
> If your card's PCI bridge is either a PLX, Altera, PLDA, Galileo, QuickLogic, AMCC or V3, then WinDriver's special chip-set APIs will dramatically shorten your development time. If this is the case, read the following overview, and jump straight to the chapter discussing this or refer to the electronic reference manual.

Using the DriverWizard to Build a Device Driver

1. Use the DriverWizard to diagnose your card. Read / Write to the IO / Memory ranges / registers that your card supports or the pipes of your USB device. Check that your device operates as expected. (See the 'DriverWizard' Chapter for details)

2. Use the DriverWizard to generate the skeleton code for your device in C or in Delphi. (See the 'DriverWizard' Chapter for details)

3. If you are using one of the supported chip sets (PLX / AMCC / V3 / Altera) as your PCI bridge - it is recommended that you use the p9054_diag.exe | p9050_diag.exe | p9080_diag.exe | p9060_diag.exe | p480_diag.exe |gt64_diag.exe | amccdiag.exe | pbc_diag.exe (respectively) as a skeleton for your driver code. These executables are applications that access all the registers and memory ranges through the respective bridge. Their full WinDriver source code is included. (See the respective chip-set chapter for more details on using the diagnostics applications).

4. Use any 32bit compiler (such as MSDEV, Borland C/C++ and Watcom C/C++ or Delphi 2/3/4) to build your code.

5. That's all you need to create your User Mode driver. If you discover that better performance is needed see the "Improving performance" section for details. This section will suggest some performance enhancements you can make in your User Mode driver, or instruct you on how to move parts of your code to the WinDriver Kernel PlugIn. This will eliminate any performance problem.

6. For a more detailed explanation please go back to chapter 4 of this manual.

See the 'WinDriver Function Reference' and the 'WinDriver Structure Reference' and the 'Implementation Issues' chapters for more details.

Writing the Device Driver without the Wizard

It is recommended to use the Driver Wizard to generate the skeleton of the driver you need. If you choose to write your driver directly without using the Wizard, proceed according to the below steps, or choose a sample that most closely resembles what your driver should do, and modify it.

1. Copy the file windrvr.h to your source code directory.

2. Add these lines to the source code:

   ```
   #include <windows.h>

   #include <winioctl.h>

   #include "windrvr.h"
   ```

3. Call WD_Open() At the beginning of your program to get a handle for WinDriver.

4. Call WD_Version() to make sure the WinDriver version installed is up to date.

5. For PCI cards: call WD_PciScanCards() to get a list of the PCI cards installed. Choose your card and call WD_PciGetCardInfo().

6. For ISA Plug and Play (PnP) cards: call WD_IsapnpScanCards() to get a list of the ISA PnP cards installed. Choose your card and call WD_IsapnpGetCardInfo().

7. For ISA (non PnP) cards: fill in your card information (IO, memory & interrupts) in the WD_CARD structure.

8. For PCMCIA Cards: call WD_PcmciaScanCards to get a list of the PCMCIA cards installed. Choose your card and call WD_PcmciaGetCardInfo().

> **Note:** *WD_ PcmciaGetCardInfo() inserts a ITEM_BUS item as the first element of the WD_ITEMS array of the WD_CARD structure that it returns. This item must be present for PCMCIA card configuration to work correctly. If you are filling up the WD_CARD structure yourself without the help of WD_ PcmciaGetCardInfo(), then you must set up this item yourself and it must be the first entry in the WD_ITEMS array*

9. For USB devices – Call WD_UsbScanDevice() to get your device unique ID.

10. For USB devices – an optional step is to call WD_UsbGetConfiguration to learn about your device configurations and interfaces.

11. Call WD_CardRegister(). For USB devices call WD_UsbDeviceRegister() instead, to open a handle to your device with the desired configuration.

12. Now you can use WD_Transfer() to perform IO and memory transfers or opreate your USB device by calling WD_UsbTransfer()

13. For ISA / ISA PnP/ PCI / PCMCIA cards: If the card uses interrupts call WD_IntEnable(). Now you can wait for interrupts using WD_IntWait().

14. To finish call WD_CardUnregister() or WD_USBDeviceUnregister() for your USB device, and at the end call WD_Close().

Win CE - Testing Your Driver on Your CE Emulation under Windows NT.

WinDriver is currently the only tool that enables you to test your driver code with your hardware on your NT machine – under the CE emulation

environment. This can dramatically shorten your development time by eliminating the need to work via a serial cable each time you want to see how your driver code operates your hardware.

If your NT host development workstation already has the target hardware plugged in, you can use the X86 HPC software emulator to test your driver. You need to generate the code as usual using the DriverWizard, or from scratch as described earlier in this chapter. When compiling the code, select the target platform as X86em from the VisualC++ WCE Configuration Toolbar. You will need to link the import library \windriver\redist\register\x86emu\windrvr_ce_emu.lib with your application program objects.

Using the Help Files

You may use the help files supplied to you with the WinDriver toolkit. Use these files by pressing 'Start' on your task bar, and choosing 'Programs \ WinDriver \ WinDriver Help' from there.

Chapter 6

Debugging

Debugging your hardware access application code should be approached in the following manner:

User Mode Debugging

Since WinDriver is accessed from User Mode, it is recommended you first debug your code using your standard debugging software.

- Use Set Debug On ’and Set Debug Off ’to toggle WinDriver runtime debugging. This will check the validity of the addresses sent to the register commands in run-time, and report errors.

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

- When developing for windows CE - If you are using the WinDbg debugger from Microsoft to connect to your target platform using a serial (COM1) port, you can use the DEBUGMSG macro inside your user-mode driver code to send *printf* style debugging output to the debugger window. Refer to the following files or directories for more information. (The ETK

documentation also includes detailed documentation on using WinDbg for user mode or driver debugging)

- \WINCE210\PUBLIC\COMMON\DDK\INC\DBGPRINT.H
- \WINCE210\PUBLIC\COMMON\OAK\DEMOS\DBGSAMP1

DebugMonitor

DebugMonitor is a powerful graphical and console mode tool for monitoring all activities handled by the WinDriver Kernel (windrvr.sys/ windrvr.vxd / windrvr.dll / windrvr.o / wdusb.sys). Using this tool you can monitor how each command sent to the kernel is executed.

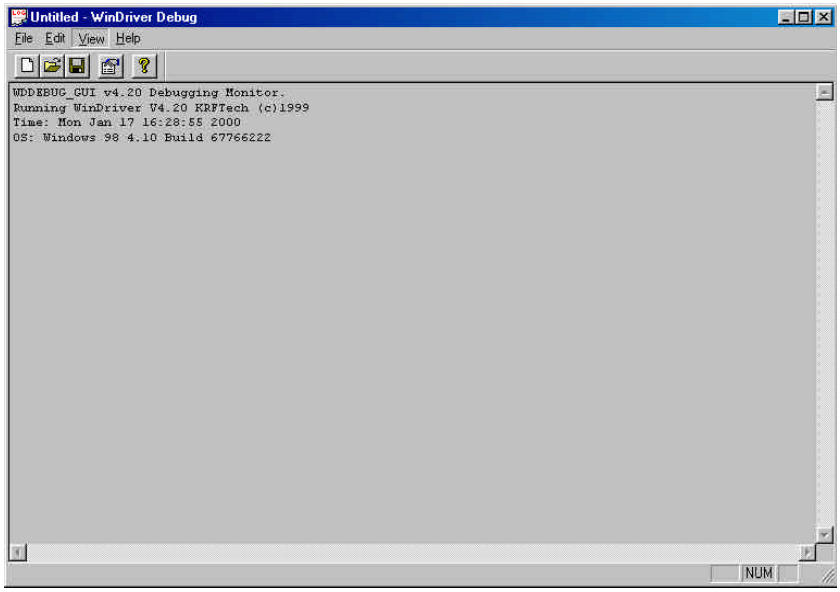
Using DebugMonitor

DebugMonitor has two modes – Graphic and Console mode. The following is an explanation on how to operate DebugMonitor in both modes.

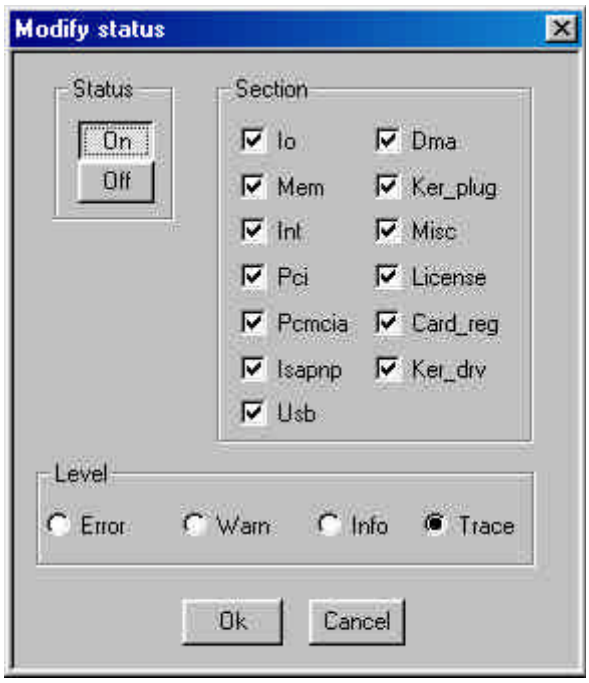
DebugMonitor – Graphical mode:

Applicable for Windows 9x, NT, 2000. You may also use DebugMonitor to debug your CE driver code running on CE emulation on Windows NT. For Linux, Solaris and CE targets use the console mode DebugMonitor.

1. Start DebugMonitor from the Start | Programs | WinDriver | DebugMonitor menu.



2. Activate and set the trace level you are interested in from the View | Debug Options menu or using the change status button.



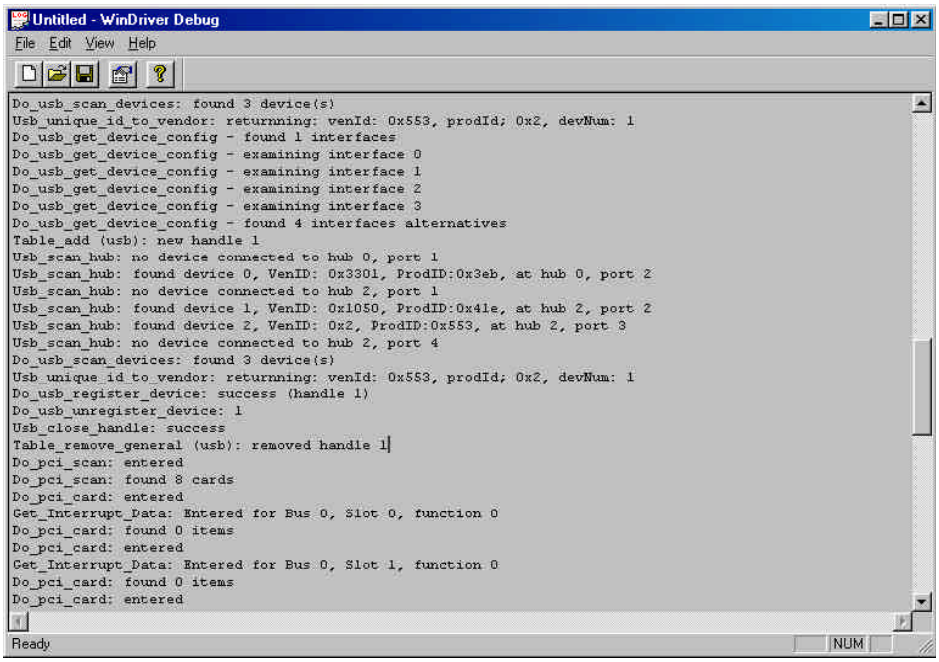
Status - Set trace on or off.

Section - Choose what part of the WinDriver API you are interested to monitor. If you are developing a PCI card and experiencing problems with your interrupt handler you should check the Int box and the PCI box. Checking more options than necessary could amount to overflow of information making it harder for you to locate your problem. USB developers should choose the USB box.
The Ker_drv option is for KernelDriver users, monitoring communication between their custom Kernel mode drivers (developed using KernelDriver) and the WinDriver kernel.

Level - Choose the level of messages you are interested to see for the resources defined. Error is the lowest level of trace, resulting with minimum output to the

screen. Trace is the highest level of tracing displaying every operation the WinDriver Kernel performs.

3. Once you have defined what you want to trace and on what level just press OK to close the "Modify status" window, activate your program, (Step by step or in one run), and watch the monitor screen for error or any unexpected messages.



DebugMonitor – Console Mode

This tool is available in all operating systems supported including Linux. To use it simply type "DebugMonitor" from the \WinDriver\util\ directory with the appropriate switches. For a list of switches available with the DebugMonitor in console mode just type "WDDebug" and a help screen will appear, describing all the different options for this command.

To see activity logged with the DebugMonitor simply type "WDDebug dump".

Chapter

7

WinDriver Function Reference

Use this chapter as a 'quick reference' to the WinDriver functions. This reference may also be found in 'WinDriver Help'.

The definition of the structures used in the following functions may be found in the 'WinDriver Structure Reference' section.

WD_Open()

Open WinDriver device & return a handle to the device. WD_Open must be called before any other WinDriver functions can be used.

Prototype

```
HANDLE WD_Open();
```

Return Value

INVALID_HANDLE_VALUE if device could not be opened, otherwise returns the handle.

Example

```
HANDLE hWD;

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

WD_Close()

Closes the WinDriver device. Must be called when finished using the driver.

Prototype

void WD_Close(HANDLE hWD);

Parameters

hWD - handle of driver from WD_Open().

Example

WD_Close (hWD);

WD_Version()

Returns the version of WinDriver currently running.

Prototype

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

Parameters (WD_VERSION elements)

dwVer - returns WinDriver's version.
cVer - returns a string of the driver's version.

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");
}
```

WD_PciScanCards()

Scan the PCI bus for cards installed.

Prototype

void WD_PciScanCards(HANDLE hWD, WD_PCI_SCAN_CARDS *pPciScan);

Parameters (WD_PCI_SCAN_CARDS elements)

searchId.dwVendorId - PCI Vendor ID to detect. If 0 then detect cards from all vendors.

searchId.dwDeviceId - PCI Device ID to detect. If 0 then detect all devices.

dwCards - returns the number of cards detected.

cardSlot[] - list of the PCI slots (dwBus, dwSlot and dwFunction) where matching cards were detected.

cardId[] - list of the corresponding PCI IDs (dwVendorId and dwDeviceId) where matching cards were detected.

Example

```
WD_PCI_SCAN_CARDS pciScan;
DWORD cards_found;
WD_PCI_SLOT pciSlot;

BZERO(pciScan);
pciScan.searchId.dwVendorId = 0x12bc;
pciScan.searchId.dwDeviceId = 0x1;
WD_PciScanCards (hWD, &pciScan);
if (pciScan.dwCards>0) // Found at least one card
{
   pciSlot = pciScan.cardSlot[0];
}
else
{
   printf ("No matching PCI cards found\n");
}
```

WD_PciGetCardInfo()

Get PCI card information: interrupts, IO & memory.

Prototype

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

Parameters (WD_PCI_CARD_INFO elements)

pciSlot - the slot of the card needed, from WD_PciScanCards().
Card - returns the card information.

Example

```
WD_PCI_CARD_INFO pciCardInfo;
WD_CARD Card;

BZERO(pciCardInfo);
pciCardInfo.pciSlot = pciSlot;
WD_PciGetCardInfo (hWD, &pciCardInfo);
if (pciCardInfo.Card.dwItems!=0)
{
   Card = pciCardInfo.Card;
}
else
{
   printf ("Failed fetching PCI card information\n");
}
```

WD_PciConfigDump()

Read / Write the PCI configuration registers.

Prototype

void WD_PciConfigDump(HANDLE hWD, WD_PCI_CONFIG_DUMP *pConfig);

Parameters (WD_PCI_CONFIG_DUMP elements)

pciSlot - PCI bus, slot and function number
pBuffer - buffer for read/write
dwOffset - offset in PCI configuration space to read/write from
dwBytes - bytes to read/write from/to buffer, returns the number of bytes read/wrote
fIsRead - if TRUE then read PCI config, FALSE write PCI config

dwResult - returns

PCI_ACCESS_OK if read/write ok
PCI_ACCESS_ERROR if error
PCI_BAD_BUS if bus doesn't exist
PCI_BAD_SLOT if slot and function don't exist

Example

```
WD_PCI_CONFIG_DUMP pciConfig;
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;

WD_PciConfigDump( hWD, &pciConfig);
if (pciConfig.dwResult!=PCI_ACCESS_OK)
{
   printf ("No PCI card in Bus 0 Slot 3\n");
}
else
{
   printf ("Card in Bus 0 Slot 3 has VendorID %x"
       "DeviceID %x\n", aBuffer[0], aBuffer[1]);
}
```

WD_PcmciaScanCards()

Scan the PCMCIA bus for PCMCIA cards installed.

Prototype

BOOL WD_PcmciaScanCards(HANDLE hWD, WD_PCMCIA_SCAN_CARDS *pBuf)

Parameters (WD_PCMCIA_SCAN_CARDS elements)

SearchId. cManufacturer – PCMCIA Card manufacturer name string
SearchId.cProductName - PCMCIA Card product name string
DwCards – returns the number of cards detected
CardSlot[] – list of the PCMCIA slots (uSocket, uFunction) where matching cards were detected.
CardId[] – list of the corresponding PCMIA Ids (cVersion, cManufacturer, cProductName, CheckSum) where matching cards were detected

Example

```
WD_PCMCIA_SCAN_CARDS pcmciaScan;
DWORD cards_found;
WD_PCMCIA_CARD pcmciaCard;

BZERO(pcmciaScan);
// Kingston DATAFLASH ATA Flash Card
strcpy (pcmciaScan.searchId.cManufacturer, "Kingston Technology");
strcpy (pcmciaScan.searchId.cProductName, "DataFlash");

WD_PcmciaScanCards (hWD, &pcmciaScan);
if (pcmciaScan.dwCards>0) // Found at least one card
{
   pcmciaCard = pcmciaScan.Card[0];
}
else
{
   printf ("No matching PCMCIA cards found\n");
}
```

WD_PcmciaGetCardInfo()

Get PCMCIA card information: interrupts, IO & memory.

Prototype

BOOL WD_PcmciaGetCardInfo(HANDLE hWD, WD_PCMCIA_CARD_INFO pPcmciaCard);

Parameters (WD_PCMCIA_CARD_INFO elements)

pcmciaSlot - the slot/function information of the card needed, from WD_PcmciaScanCards().
Card - returns the card information.

Example

```
WD_PCMCIA_CARD_INFO pcmciaCardInfo;
WD_CARD Card;

BZERO(pcmciaCardInfo);

// get this from WD_PcmciaScanCards()
pcmciaCardInfo.pcmciaSlot = pcmciaSlot;

WD_PcmciaGetCardInfo (hWD, &pcmciaCardInfo);
if (pcmciaCardInfo.Card.dwItems!=0)
{
   Card = pcmciaCardInfo.Card;
}
else
{
   printf ("Failed fetching PCMCIA card information\n");
}
```

WD_PcmciaConfigDump()

Read / Write the PCMCIA configuration registers.

Prototype

void WD_PcmciaConfigDump(HANDLE hWD, WD_PCMCIA_CONFIG_DUMP *pConfig);

Parameters (WD_PCMCIA_CONFIG_DUMP elements)

PcmiaSlot - Slot descriptor of PCMCIA card
PBuffer - buffer for read/write
DwOffset - offset in pcmcia configuration space to read/write from
DwBytes - bytes to read/write from/to buffer returns the number of bytes read/wrote
FisRead - if 1 then read pci config, 0 write pci config
DwResult - PCMCIA_ACCESS_RESULT

WD_IsapnpScanCards()

Scan the ISA bus for ISA Plug and Play cards installed.

Prototype

void WD_IsapnpScanCards(HANDLE hWD, WD_ISAPNP_SCAN_CARDS *pIsapnpScan);

Parameters (WD_ISAPNP_SCAN_CARDS elements)

searchId.cVendor - ISA PnP Vendor ID. This identifies the vendor and card type. If cVendor[0] is \0 'then this will search for all Vendor IDs.

searchId.dwSerial - ISA PnP serial device number. If zero, then search for all serial numbers.

dwCards - returns the number of cards detected.

Card[] - list of the cards detected.

Example

```
WD_ISAPNP_SCAN_CARDS isapnpScan;
DWORD cards_found;
WD_ISAPNP_CARD isapnpCard;

BZERO(isapnpScan);
// CTL009e - Sound Blaster ISA PnP card
strcpy (isapnpScan.searchId.cVendorId, "CTL009e");
isapnpScan.searchId.dwSerial = 0;
WD_IsapnpScanCards (hWD, &isapnpScan);
if (isapnpScan.dwCards>0) // Found at least one card
{
   isapnpCard = isapnpScan.Card[0];
}
else
{
   printf ("No matching ISA PnP cards found\n");
}
```

WD_IsapnpGetCardInfo()

Get ISA Plug and Play card information: interrupts, IO & memory.

Prototype

```
BOOL WD_IsapnpGetCardInfo(HANDLE hWD, WD_ISAPNP_CARD_INFO *pIsapnpCard);
```

Parameters (WD_ISAPNP_CARD_INFO elements)

CardId - the card ID needed, from WD_IsapnpScanCards().
dwLogicalDevice - if ISA card device is multi-function, then this is the number of the logical device to use, otherwise set it to zero.
cLogicalDeviceId - returns ascii code of logical device ID found.
dwCompatibleDevices - returns the number of compatible device IDs in CompatibleDevice array.
CompatibleDevice[] - returns an array of compatible device IDs
cIdent - returns the ascii device identification string
Card - returns the card information.

Example

```
WD_ISAPNP_CARD_INFO isapnpCardInfo;
WD_CARD Card;

BZERO(isapnpCardInfo);
// from WD_IsapnpScanCard():
isapnpCardInfo.CardId = isapnpCard;
isapnpCardInfo.dwLogicalDevice = 0;
WD_IsapnpGetCardInfo (hWD, &isapnpCardInfo);
if (isapnpCardInfo.Card.dwItems!=0)
{
   Card = isapnpCardInfo.Card;
}
else
{
   printf ("Failed fetching ISA PnP card information\n");
}
```

WD_IsapnpConfigDump()

Read / Write the ISA PnP configuration registers.

Prototype

```
void WD_IsapnpConfigDump( HANDLE hWD, WD_ISAPNP_CONFIG_DUMP *pConfig);
```

Parameters (WD_ISAPNP_CONFIG_DUMP elements)

```
CardId - the card ID needed, from WD_IsapnpScanCards().
dwOffset - offset in ISA PnP configuration space to read/write from
fIsRead - if TRUE then read config, FALSE write config
bData - the data to read or write.
dwResult - returns
        ISAPNP_ACCESS_OK if read/write ok
        ISAPNP_ACCESS_ERROR if error
        ISAPNP_BAD_ID if card does not exist
```

Example

```
WD_ISAPNP_CONFIG_DUMP isapnpConfig;

BZERO(isapnpConfig);
// from WD_IsapnpScanCard():
isapnpConfig.CardId = isapnpCard;
isapnpConfig.dwOffset = 0;
isapnpConfig.fIsRead = TRUE;

WD_IsapnpConfigDump( hWD, &isapnpConfig);
if (isapnpConfig.dwResult!=ISAPNP_ACCESS_OK)
{
   printf ("No ISA PnP card specified slot\n");
}
else
{
   printf ("ISA PnP config in offset 0 = %x"
      isapnpConfig.bData);
}
```

WD_CardRegister()

Register card - install interrupts & map card memory. For USB device see - WD_UsbDeviceRegister()

Must be called in order to use interrupts & perform IO & memory transfers to card.

Prototype

void WD_CardRegister(HANDLE hWD, WD_CARD_REGISTER *pCardReg);

Parameters (WD_CARD_REGISTER elements)

Card - information of card to register (interrupts, IO & memory)
Card.dwItems - number of items in Card.Item array.
Card.Item[] - items of card. each item can be an IO range, memory range or an interrupt.
Card.Item[i].item - can be ITEM_INTERRUPT, ITEM_MEMORY or ITEM_IO
Card.Item[i].fNotSharable - normally should be TRUE, in order that two applications will not attempt to access the same hardware at the same time.

for IO range item:

Card.Item[i].I.IO.dwAddr - first address of IO range.
Card.Item[i].I.IO.dwBytes - length of range in bytes.

for memory range item:

Card.Item[i].I.Mem.dwPhysicalAddr - first address of physical memory range.
Card.Item[i].I.Mem.dwBytes - length of range in bytes.
Card.Item[i].I.Mem.dwTransAddr - returns the base address to use for memory transfers with WD_Transfer().
Card.Item[i].I.Mem.dwUserDirectAddr - returns the base address to use for memory transfers directly by user.

for interrupt item:

Card.Item[i].I.Int.dwInterrupt - interrupt IRQ to install.
Card.Item[i].I.Int.dwOptions - normally 0. for level sensitive interrupts use INTERRUPT_LEVEL_SENSITIVE.
Card.Item[i].I.Int.hInterrupt - returns interrupt handle to use with WD_IntEnable().
fCheckLockOnly - to register card should be FALSE. in order just to check if card can be registered (i.e.: not used by someone else) should be TRUE.
hCard - returns the handle of the card, or 0 if card cannot be registered. if this is just check lock then hCard will return 1 if card can be registered, or 0 if not.

Example

```
WD_CARD Card;
WD_CARD_REGISTER cardReg;

// the info for Card comes from WD_PciGetCardInfo()
// for PCI cards. for ISA cards the information has to be
// set by the user (IO/memory address & interrupt number).
BZERO(cardReg);
cardReg.Card = Card;
cardReg.fCheckLockOnly = FALSE;
WD_CardRegister (hWD, &cardReg);
if (cardReg.hCard==0)
   printf ("could not lock device - already in use");
```

WD_CardUnregister()

Un-register a card, and free its resources. For USB devices see - WD_UsbDeviceUnregister()

Prototype

void WD_CardUnregister(HANDLE hWD, WD_CARD_REGISTER *pCardReg);

Parameters (WD_CARD_REGISTER elements)

hCard - handle of card to un-register.

Example

WD_CardUnregister (hWD, &cardReg);

WD_Transfer()

Execute a read/write instruction to IO port or memory. For USB devices see - WD_UsbTransfer()

Prototype

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

Parameters (WD_TRANSFER elements)

cmdTrans - command of operation: <dir><p>_<string><size>:
    dir - R for read, W for write
    p   - P for port, M for memory
    string   - S for string, none for single transfer
    size   - BYTE, WORD or DWORD
dwPort - Port address for IO, or User address for memory transfer. User address for a memory mapped card is returned by WD_CardRegister(), in the Card structure.

For single transfer:

Data.Byte for Byte read/write.
Data.Word for Word read/write.
Data.Dword for DWord read/write.

For string transfer:

dwBytes - number of bytes to transfer.
fAutoinc - is TRUE if IO or memory address should increment for transfer. if FALSE, all data is transferred to the same port address.
dwOptions - should be 0.
Data.pBuffer - the buffer with the data to transfer to/from.

Example

```
WD_TRANSFER Trns;
BYTE read_data;

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

WD_MultiTransfer()

Perform multiple IO & memory transfers.

Prototype

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

Parameters

pTransArray - array of transfer commands, same as in WD_Transfer()
dwNumTransfers - number of commands in array

Example

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

BZERO(Trns);
Trns[0].cmdTrans = WP_WORD; // Write Port Word
Trns[0].dwPort = 0x1e0;
Trns[0].Data.Word = 0x1023;

Trns[1].cmdTrans = WP_WORD;
Trns[1].dwPort = 0x1e0;
Trns[1].Data.Word = 0x1022;

Trns[2].cmdTrans = WP_SBYTE; // Write Port String Byte
Trns[2].dwPort = 0x1f0;
Trns[2].dwBytes = strlen(cData);
Trns[2].fAutoinc = FALSE;
Trns[2].dwOptions = 0;
Trns[2].Data.pBuffer = cData;

Trns[3].cmdTrans = RP_DWORD; // Read Port DWord
Trns[3].dwPort = 0x1e4;

WD_MultiTransfer(hWD, Trns, 4);
dwResult = Trans[3].Data.Dword;
```

WD_IntEnable()

Enable interrupt processing.

Note: The easiest way to handle interrupts with WinDriver is by defining the Interrupt in the Wizard, and letting the Wizard generate the code for you. (In Plug-n-Play cards, the Wizard will auto-detect the interrupts for you).

Prototype

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

Parameters (WD_INTERRUPT elements)

hInterrupt - handle of interrupt to enable. The handle is returned by WD_CardRegister(), in the Card structure.
Cmd - array of transfer commands to perform on hardware interrupt. These commands are needed for level sensitive interrupts, to lower the interrupt level. Otherwise, after WinDriver finishes dealing with the interrupt, another interrupt will immediately occur. If no commands are needed, this should be NULL. The commands are the same as in WD_Transfer().
dwCmds - number of transfer commands in Cmd array.
dwOptions - should be 0. If transfer commands are used for the interrupt installed, set the value to INTERRUPT_CMD_COPY to copy back the transfer to user-mode from the WinDriver kernel.
kpCall - kernel plugin call
fEnableOk - returns TRUE if enable succeeded.

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 = TRUE;
cardReg.Card.Item[0].I.Int.dwInterrupt = 10; // IRQ 10
// INTERRUPT_LEVEL_SENSITIVE - set to level sensitive
// interrupts, otherwise should be 0.
// ISA cards usually are edge sensitive, and PCI cards
// usually are 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 - already in use");
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");
}
```

WD_IntDisable()

Disable interrupt processing

Prototype

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

Parameters (WD_INTERRUPT elements)

hInterrupt - handle of interrupt to disable.

Example

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

WD_IntWait()

Wait for an interrupt.

Prototype

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

Parameters (WD_INTERRUPT elements)

hInterrupt - handle of interrupt to wait for.
fStopped - returns TRUE if interrupt was disabled while waiting.
dwCounter - returns the number of interrupts processed.
dwLost - returns the number of interrupts not yet dealt with.
Cmd - if commands are set on interrupt should point to commands array, otherwise should be NULL.

Example

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

ProcessInterrupt (Intrp.dwCounter);
}
```

WD_IntCount()

Count the number of interrupts from the time WD_IntEnabled was called.

Prototype

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

Parameters (WD_INTERRUPT elements)

hInterrupt - handle of interrupt to count.
dwCounter - returns the number of interrupts processed.
dwLost - returns the number of interrupts not yet dealt with.
Cmd - if commands are set on interrupt should point to commands array, otherwise should be NULL.

Example

```
DWORD dwNumInterrupts;

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

WD_DMALock()

Lock a linear memory region, and return a list of the corresponding physical addresses.

Prototype

void WD_DMALock(HANDLE hWD, WD_DMA *pDma);

Parameters (WD_DMA elements)

pUserAddr - user base address of region needed to be locked for DMA transfer.
dwBytes - number of bytes to lock.
dwOptions - normally 0.

- Set to DMA_KERNEL_BUFFER_ALLOC so WinDriver will allocate a contiguous buffer. When this option is set, the user address of the buffer will be returned in pUserAddr. Use this option if your device does not support scatter/gather transfers.
- Set to DMA_LARGE_BUFFER for locking down regions larger than 1MB (See Implementing DMA 'for more details).

Page[] - returns an array listing the physical addresses of the locked memory ranges. Program the card's DMA to transfer data to these addresses.
Page[i].pPhysicalAddr - physical address of page i.
Page[i].dwBytes - length in bytes of page i.
dwPages - returns the number of pages in Page array.
hDma - returns the handle for DMA buffer.

Example

```
WD_DMA Dma;
PVOID pBuffer = malloc (20000);

BZERO(Dma);
Dma.dwBytes = 20000;
Dma.pUserAddr = pBuffer;
Dma.dwOptions = 0;
WD_DMALock (hWD, &Dma);
// on return Dma.Page has the list of physical addresses
if (Dma.hDma==0)
   printf ("Could not lock down buffer\n");
```

WD_DMAUnlock()

Unlock a DMA buffer.

Prototype

void WD_DMAUnlock(HANDLE hWD, WD_DMA *pDma);

Parameters (WD_DMA elements)

hDma - handle for DMA buffer to unlock.

Example

WD_DMAUnlock (hWD, &Dma);

WD_Sleep()

Delay execution for a specific amount of time. This function is used when accessing slow hardware.

Prototype

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

Parameters (WD_Sleep elements)

dwMicroSeconds - time, in microseconds, to sleep.
dwOptions - should be zero.

Example

```
WD_SLEEP sleep;

BZERO (sleep);
sleep.dwMicroSeconds = 1000; // Sleep for 1 millisecond
sleep.dwOptions = 0;
WD_Sleep (hWD, &sleep);
```

WD_UsbScanDevice()

Scan the USB tree for devices installed.

Prototype :

```
void WD_UsbScanDevice( HANDLE hWD, WD_USB_SCAN_DEVICES *pScan);
```

Parameters (WD_USB_SCAN_DEVICES elements):

searchId.dwVendorId - USB Vendor ID to detect. If 0 then detect devices from all vendors.
searchId.dwProductId - USB Product ID to detect. If 0 then detect all products from the selected vendor.
DwDevices - returns the number of devices detected.
uniqueId[] - list of the USB unique IDs where matching devices were detected.
deviceGeneralInfo[] - list of the devices general info (Vendor ID, Product ID, device address, number of configurations ...)

Example:

```
WD_USB_SCAN_DEVICES scan;
DWORD uniqueId;

BZERO(scan);
scan.searchId.dwVendorId = 0x553;
scan.searchId.dwProductId = 0x2;
WD_UsbScanDevice(hWD, &scan);
if (scan.dwDevices>0) // Found at least one card
{
   uniqueId = scan.uniqueId[0];
}
else
{
   printf ("No matching USB devices found\n");
}
```

WD_UsbGetConfiguration()

Get a USB device information

Prototype:

void WD_UsbGetConfiguration(HANDLE hWD, WD_USB_CONFIGURATION *pConfig);

Parameters (WD_USB_CONFIGURATION elements):

uniqueId - the unique ID of the device as received from WD_UsbScanDevice()
dwConfigurationIndex - the index of the configuration to get (zero based). The number of configurations are received from WD_UsbScanDevice() in the deviceGeneralInfo.
configuration - configuration general data (value, attributes ...)
dwInterfaceAlternatives - how many interfaces (and alternate interfaces) there are on the device.
Interface[] - list of intrface descriptions (number of endpoints, class, sub class, protocol ...)

Example:

```
WD_USB_CONFIGURATION config;

BZERO(config);
config.uniqueId = 2;
config.dwConfigurationIndex = 0;
WD_UsbGetConfiguration(hWD, &config);
printf("found %d interfaces\n", config.dwInterfaceAlternatives);
```

WD_UsbDeviceRegister()

Register the selected interface of the device. (This tells the hardware which interface to work with)

Must be called in order to perform data transfers on the pipes.

Prototype:

```
void WD_UsbDeviceRegister( HANDLE hWD, WD_USB_DEVICE_REGISTER *pDevice);
```

Parameters (WD_USB_DEVICE_REGISTER elements):

uniqueId - the device unique ID as received from WD_UsbScanDevice()
dwConfigurationIndex - the index of the configuration to register (zero based). The number of configurations are received from WD_UsbScanDevice() in the deviceGeneralInfo.
dwInterfaceNum - interface number to register as received from WD_UsbGetConfiguration()
dwInterfaceAlternate - interface alternate number to register as received from WD_UsbGetConfiguration()
hDevice - the returned handle of the device
Device - the returned device description (number of pipes and their description)
dwOptions; - should be zero
cName[]; - name of card
cDescription[]; - description

Example:

```
WD_USB_DEVICE_REGISTER device;

BZERO(device);
device.uniqueId = 2;
device.dwConfigurationIndex = 0;
device.dwInterfaceNum = 1;
device.dwInterfaceAlternate = 1;
WD_DeviceRegister(hWD, &device);

if (!device.hDevice)
   printf("error - could not register device\n");
else
   printf("deivce has %d pipes\n", device.Device.dwPipes);
```

WD_UsbDeviceUnregister()

Un-register the device.

Prototype:

void WD_UsbDeviceUnregister(HANDLE hWD, WD_USB_DEVICE_REGISTER *pDevice);

Parameters (WD_USB_DEVICE_REGISTER elements):

hDevice - the handle of the device to un-register

Example:

WD_UsbDeviceUnregister(hWD, &device)

WD_UsbTransfer()

Perform Read/Write data transfers drom/to the device using it's pipes.

Prototype:

void WD_UsbTransfer(HANDLE hWD, WD_USB_TRANSFER *pTrans);

Parameters (WD_USB_TRANSFER elements):

hDevice - handle of USB device as recieved from WD_UsbDeviceRegister()
dwPipe - pipe number on the device
fRead - perform read or write
dwOptions - can be USB_TRANSFER_HALT to halt the pervious transfer on the same pipe.
pBuffer - pointer to buffer to read/write
dwBytes - size of the buffer
dwTimeout - timeout for the transfer in milli-seconds. 0==>no timeout.
dwBytesTransfered - returns the number of bytes actually read/written
SetupPacket[8] - 8 bytes setup packet for control pipe transfer
fOK - return TRUE if the transfer was successful

Example:

```
WD_USB_TRANSFER trans;

BZERO(trans);
trans.hDevice = hDevice;
trans.dwPipe = 0x81;
trans.fRead = TRUE;
trans.pBuffer = malloc(100);
trans.dwBytes = 100;
WD_UsbTransfer(hWD, &trans);

if (!fOK)
   printf("error on transfer\n");
else
   printf("transfered %d bytes from %d\n", trans.dwBytesTransfered, trans.dwBytes);
```

WD_UsbResetPipe()

Reset the pipe to its default state (resets the firmware's pipe's state machine to first state).

Prototype:

```
void WD_UsbResetPipe( HANDLE hWD, WD_USB_RESET_PIPE *pReset);
```

Parameters (WD_USB_RESET_PIPE elements):

```
hDevice  - handle of the USB deivce
dwPipe   - the pipe number to reset
```

Example:

```
WD_USB_RESET_PIPE reset;

BZERO(reset);
reset.hDevice = hDevice;
reset.dePipe = 0x81;
WD_UsbResetPipe(hWD, &reset);
```

WD_UsbResetDevice()

Reset the USB device to its default state.

Prototype:

void WD_UsbResetPipe(HANDLE hWD, DWORD hDevice);

Parameters:

hDevice - handle of the USB deivce

Example:

WD_UsbResetDevice(hWD, hDevice)

Chapter 8

WinDriver Structure Reference

WD_DMA

Contains information about a DMA buffer.

Used by WD_DMALock() and WD_DMAUnlock().

Members:

TYPE	NAME	DESCRIPTION
DWORD	hDma	Handle of DMA buffer
PVOID	pUserAddr	Beginning of buffer
DWORD	dwBytes	Size of buffer
DWORD	dwOptions	Allocation options: Bit masked flag - set to '0' for no option, or: DMA_KERNEL_BUFFER_ALLOC DMA_KBUF_BELOW_16M DMA_LARGE_BUFFER
DWORD	dwPages	Number of pages in buffer
WD_DMA_PAGE	Page [WD_DMA_PAGES]	Array of the pages in the buffer

WD_DMA_PAGE

Members:

TYPE	NAME	DESCRIPTION
PVOID	pPhysicalAddr	physical address of page
DWORD	dwBytes	size of page

WD_TRANSFER

This structure defines a single transfer operation to be performed by WinDriver.

Used by WD_Transfer(), WD_MultiTransfer(), WD_IntEnable().

Members:

TYPE	NAME	DESCRIPTION
DWORD	cmdTrans	Transfer command WD_TRANSFER_CMD
DWORD	dwPort	io port for transfer or user memory address
DWORD	dwBytes	Number of bytes for string transfer
DWORD	fAutoinc	transfer from one port/address or use incremental range of addresses
DWORD	dwOptions	must be 0
Union	Data	the data for transfer
UCHAR	Data.Byte	Use for byte transfer
USHORT	Data.Word	Use for word transfer
DWORD	Data.Dword	Use for dword transfer
PVOID	Data.pBuffer	Use for string transfer

WD_INTERRUPT

Used to describe an interrupt

Used by WD_IntEnable(), WD_IntDisable(), WD_IntWait(), WD_IntCount().

Members:

TYPE	NAME	DESCRIPTION
DWORD	hInterrupt	handle of interrupt
DWORD	dwOptions	interrupt options: Bit masked flag. May be '0' for no option, or: INTERRUPT_LEVEL_SENSITIVE (for level sensitive interrupts) or INTERRUPT_CMD_COPY (choose this for when you need the WinDriver kernel to copy the actions of the read command it has done to acknowledge the interrupt, back to the user mode).
WD_TRANSFER	*Cmd	Pointer to commands to perform on interrupt
DWORD	dwCmds	number of commands
WD_KERNEL_PLUGIN_CALL	kpCall	kernel plugin call
DWORD	fEnableOk	'1' if WD_IntEnable() succeeded
DWORD	dwCounter	number of interrupts received
DWORD	dwLost	number of interrupts not yet dealt with
DWORD	fStopped	was interrupt disabled during wait

WD_VERSION

Describes version of WinDriver in use

Used by WD_Version().

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwVer	version
CHAR	cVer[100]	string of version

WD_CARD_REGISTER

Holds a handle to a registered card.

Used by WD_CardRegister(), WD_CardUnregister().

Members:

TYPE	NAME	DESCRIPTION
WD_CARD	Card	card to register
DWORD	fCheckLockOnly	only check if card is lockable, return hCard=1 if OK
DWORD	hCard	handle of card

WD_CARD

Describes the card's resources.

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwItems	Number of items in card
WD_ITEMS	Item [WD_CARD_ITEMS]	Array of items[0...dwItems-1]

WD_ITEMS

Defines each item (resource) in a card.

Members:

TYPE	NAME	DESCRIPTION
DWORD	item	ITEM_TYPE
DWORD	fNotSharable	If TRUE, item may not be shared.
union	I	Item specific information
struct	I.Mem	ITEM_MEMORY
DWORD	I.Mem.dwPhysicalAddr	Physical address on card
DWORD	I.Mem.dwBytes	Address range
DWORD	I.Mem. dwTransAddr	Returns the address to pass on to transfer commands
DWORD	I.Mem. dwUserDirectAddr	Returns the address for direct user read/write
DWORD	dwCpuPhysicalAddr	returns the CPU physical address of card
struct	I.IO	ITEM IO
DWORD	I.IO.dwAddr	Beginning of IO address
DWORD	I.IO.dwBytes	IO range
struct	I.Int	ITEM INTERRUPT
DWORD	I.Int. dwInterrupt	Number of interrupt to install
DWORD	I.Int.dwOptions	interrupt options: INTERRUPT_LEVEL_SENSITIVE
DWORD	I.Int.hInterrupt	Returns the handle of the interrupt installed

WD_SLEEP

Defines a sleep command.

Used by WD_Sleep().

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwMicroSeconds	Sleep time in micro seconds - 1/1,000,000 of a second.
DWORD	dwOptions	should be zero

WD_PCI_SLOT

Defines a physical location of a PCI card.

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwBus	PCI physical bus number of card
DWORD	dwSlot	PCI physical slot number of card
DWORD	dwFunction	PCI function on card

WD_PCI_ID

Defines the identity of a PCI card.

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwVendorId	The PCI Vendor ID of the card.
DWORD	dwDeviceId	The PCI Device ID of the card.

WD_PCI_SCAN_CARDS

Receives information on cards detected on the PCI bus.

Used by WD_PciScanCards().

Members:

TYPE	NAME	DESCRIPTION
WD_PCI_ID	searchId	If searchId.dwVendorId==0, scan all vendor IDs. If searchId.dwDeviceId==0, scan all device IDs.
DWORD	dwCards	Number of cards found
WD_PCI_ID	cardId [WD_PCI_CARDS]	VendorID & DeviceID of cards found
WD_PCI_SLOT	cardSlot [WD_PCI_CARDS]	PCI slot info of cards found

WD_PCI_CARD_INFO

Describes a PCI card's resources detected.

Used by WD_PciGetCardInfo().

Members:

TYPE	NAME	DESCRIPTION
WD_PCI_SLOT	pciSlot	PCI slot
WD_CARD	Card	get card parameters for PCI slot

WD_PCI_CONFIG_DUMP

Defines a read / write command to the PCI configuration registers of a PCI card.

Used by WD_PciConfigDump().

Members:

TYPE	NAME	DESCRIPTION
WD_PCI_SLOT	pciSlot	PCI bus, slot and function number
PVOID	pBuffer	buffer for read/write
DWORD	dwOffset	offset in PCI configuration space to read/write from
DWORD	dwBytes	bytes to read/write from/to buffer returns the number of bytes read/written
DWORD	fIsRead	FALSE - write PCI config TRUE - read PCI config
DWORD	dwResult	0PCI_ACCESS_OK - read/write ok 1PCI_ACCESS_ERRROR - error 2PCI_BAD_BUS - bus does not exist (read only) 3PCI_BAD_SLOT - slot or function does not exist (read only)

WD_ISAPNP_CARD_ID

Identifies a specific ISA Plug and Play card on the ISA bus.

Members:

TYPE	NAME	DESCRIPTION
CHAR	cVendor [8]	Vendor ID
DWORD	dwSerial	Serial number of card

WD_ISAPNP_CARD

Information on an ISA Plug and Play card.

Members:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_CARD_ID	cardId	Vendor ID and serial number of card found
DWORD	dwLogicalDeices	Number of logical devices on the card
BYTE	bPnPVersionMajor	ISA PnP version major
BYTE	bPnPVersionMinor	ISA PnP version minor
BYTE	bVendorVersionMajor	Vendor version major
BYTE	bVendorVersionMinor	Vendor version minor
WD_ISAPNP_ANSI	cIdent	Device identifier

WD_ISAPNP_SCAN_CARDS

Used to receive information on cards detected on the ISA PnP bus.

Used by WD_IsapnpScanCards().

Members:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_CARD_ID	searchId	If searchId.cVendorId[0]==0, scan all vendor IDs. If searchId.dwSerial==0, scan all serial numbers.
DWORD	dwCards	Number of cards found
WD_ISAPNP_CARD	Card [WD_ISAPNP_CARDS]	cards found

WD_ISAPNP_CARD_INFO

Describes an ISA PnP card device's resources detected.

Used by WD_IsapnpGetCardInfo().

Members:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_CARD_ID	cardId	Vendor ID and serial number of card to get information on
DWORD	dwLogicalDeice	Number of logical device to get information on
WD_ISAPNP_COMP_ID	clogicalDeviceId[8]	ascii of logical device id found
DWORD	dwCompatibleDevices	Number of compatible devices found
WD_ISAPNP_COMP_ID	CompatibleDevice[WD_ISAPNP_COMPATIBLE_IDS]	Compatible device IDs
WD_ISAPNP_ANSI	cIdent	Identity of device
WD_CARD	Card	The card resource information

WD_ISAPNP_CONFIG_DUMP

Defines a read / write command to the ISA PnP configuration registers of an ISA PnP card.

Used by WD_IsapnpConfigDump ().

Members:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_CARD_ID	cardId	VendorID and serial number of card
DWORD	dwOffset	offset in ISA PnP configuration space to read/write from
DWORD	fIsRead	if 1 then read ISA PnP config, 0 write ISA PnP config
BYTE	bData	result data of byte read/write
DWORD	dwResult	ISAPNP_ACCESS_RESULT

WD_PCMCIA_SLOT

Defines a physical location of a PCMCIA card.

Members:

TYPE	NAME	DESCRIPTION
BYTE	uSocket	Specifies the socket number (first socket is 0)
BYTE	uFunction	Specifies the function number (first function is 0)

WD_PCMCIA_ID

Defines the identity of a PCMCIA card.

Members:

TYPE	NAME	DESCRIPTION
CHAR	cVersion[4]	The Card's PCMCIA version
CHAR	cManufacturer[16]	Manufacturer name
CHAR	cProductName[12]	Product name
USHORT	cCheckSum	Card's checksum value

WD_PCMCIA_SCAN_CARDS

Receives information on cards detected on the PCMCIA bus.

Used by WD_PcmciaScanCards().

Members:

TYPE	NAME	DESCRIPTION
WD_PCMCIA_ID	searchId	if strlen(searchId.cManufacturer) ==0, scan all Manufacturers. if strlen(searchId.cProductName) ==0, scan all product names.
DWORD	dwCards	Number of cards found
WD_PCMCIA_ID	cardId [WD_PCMCIA_CARDS]	Manufacturer Name, Product Name, Version and CRC Info of card found
WD_PCMCIA_SLOT	cardSlot [WD_PCMCIA_CARDS]	PCMCIA slot/function info of cards found

WD_PCMCIA_CARD_INFO

Describes a PCMCIA card's resources detected.

Used by WD_PcmciaGetCardInfo().

Members:

TYPE	NAME	DESCRIPTION
WD_PCMCIA_SLOT	pcmciaSlot	PCMCIA slot information
WD_CARD	Card	get card parameters for PCMCIA slot

WD_ PCMCIA_CONFIG_DUMP

Defines a read / write command to the PCMCIA configuration registers of a PCMCIA card.

Used by WD_PcmciaConfigDump ().

Members:

TYPE	NAME	DESCRIPTION
WD_PCMCIA_SLOT	pcmiaSlot	Slot descriptor of PCMCIA card
PVOID	pBuffer	buffer for read/write
DWORD	dwOffset	offset in pcmcia PnP configuration space to read/write from
DWORD	dwBytes	bytes to read/write from/to buffer returns the number of bytes read/wrote
DWORD	fIsRead	if 1 then read pci config, 0 write pci config
DWORD	dwResult	PCMCIA_ACCESS_RESULT

WD_USB_ID

Defines the identity of the USB device

Members:

TYPE	NAME	DESCRIPTION
DWORD	DwVendorId	vendor ID of the USB device
DWORD	DwProductId	product ID of the USB device

WD_USB_PIPE_INFO

Information about a pipe

Members

TYPE	NAME	DESCRIPTION
DWORD	dwNumber	the number of the pipe (Pipe 0 is the default pipe)
DWORD	dwMaximumPacketSize	the maximum packet size of internals transfers on the pipe
DWORD	type	Control, Isochronous, Bulk or Interrupt
DWORD	direction	In=1, out=2 or in&out=3
DWORD	dwInterval	Intervals of data transfer in ms (relevant to Interrupt pipes)

WD_USB_CONFIG_DESC

Describe a configuration

Members

TYPE	NAME	DESCRIPTION
DWORD	dwNumInterfaces	the configuration number
DWORD	dwValue	the device value
DWORD	dwAttributes	the device attributes
DWORD	MaxPower	the device MaxPower

WD_USB_INTERFACE_DESC

Describe a interface

Members:

TYPE	NAME	DESCRIPTION
DWORD	DwNumber	the interface number
DWORD	DwAlternateSetting	the interface alternate value
DWORD	DwNumEndpoints	the number of endpoints in the interface
DWORD	DwClass	the interface class
DWORD	DwSubClass	the interface sub class
DWORD	DwProtocol	the interface protocol
DWORD	DwIndex	the index of the interface

WD_USB_ENDPOINT_DESC

Describes an endpoint

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwEndpointAddress	endpoint address
DWORD	dwAttributes	endpoints Attributes
DWORD	dwMaxPacketSize	maximum packet size
DWORD	dwInterval	interval in mili-seconds

WD_USB_INTERFACE

Holds interface data

Members:

TYPE	NAME	DESCRIPTION
WD_USB_INTERFACE_DESC	Interface	the interface description
WD_USB_ENDPOINT_DESC	Endpoints[]	list of the interface endpoints

WD_USB_CONFIGURATION

Holds configuration data

Members:

TYPE	NAME	DESCRIPTION
DWORD	uniqueId	the unique ID of the device
DWORD	dwConfigurationIndex	the Configuration Index
WD_USB_CONFIG_DESC	configuration	the configuration description
DWORD	dwInterfaceAlternatives	number of interfaces and their alternates
WD_USB_INTERFACE	Interface[]	list of the configuration interfaces

WD_USB_HUB_GENERAL_INFO

Holds hub information (if the selected device is hub)

Members:

TYPE	NAME	DESCRIPTION
DWORD	fBusPowered	is bus powered or self powered
DWORD	dwPorts	number of ports on this hub
DWORD	dwCharacteristics	hub Charateristics
DWORD	dwPowerOnToPowerGood	port power on till power good in ms
DWORD	dwHubControlCurrent	max current in mA

WD_USB_DEVICE_GENERAL_INFO

Device general information

Members:

TYPE	NAME	DESCRIPTION
WD_USB_ID	deviceId	the device vendor ID and product ID
DWORD	dwHubNum	the number of the hub the device attached to
DWORD	dwPortNum	the number of port on the hub that the device is attached to
DWORD	fHub	is the device itself a hub
DWORD	fFullSpeed	full speed or low speed
DWORD	dwConfigurationsNum	how many configurations does this device have
DWORD	deviceAddress	the device's phisical address
WD_USB_HUB_GENERAL_INFO	hubInfo	contains information on the device if the device is a hub

WD_USB_DEVICE_INFO

Holds device pipes information

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwPipes	number of pipes
WD_USB_PIPE_INFO	Pipe[]	list of pipes information

WD_USB_SCAN_DEVICES

Defines a scan command

Members:

TYPE	NAME	DESCRIPTION
WD_USB_ID	searchId	if dwVendorId==0 - scan all vendor IDs, if dwProductId==0 - scan all product IDs
DWORD	dwDevices	how many devices were found
DWORD	uniqueId[]	a list of the unique IDs to identify the devices
WD_USB_DEVICE_GENERAL_INFO	deviceGeneralInfo[]	list of general information about the found devices

WD_USB_TRANSFER

Defines transfer command

Members:

TYPE	NAME	DESCRIPTION
DWORD	hDevice	handle of USB device to read from or write to
DWORD	dwPipe	pipe number on device
DWORD	fRead	read or write
DWORD	dwOptions	can be USB_TRANSFER_HALT to halt the pervious transfer on the same pipe.
PVOID	pBuffer	pointer to buffer to read/write
DWORD	dwBytes	the size of the buffer
DWORD	dwTimeout	timeout for the transfer in milli-seconds. 0==>no timeout.
DWORD	dwBytesTransfered	returns the number of bytes actually read/written
BYTE	SetupPacket[8]	setup packet for control pipe transfer
DWORD	fOK	return TRUE if the transfer was successful

WD_USB_DEVICE_REGISTER

Define device registration command

Members:

TYPE	NAME	DESCRIPTION
DWORD	uniqueId	the device unique ID
DWORD	dwConfigurationIndex	the index of the configuration to register
DWORD	dwInterfaceNum	interface to register
DWORD	dwInterfaceAlternate	alternate number of the interface to register
DWORD	hDevice	handle of device
WD_USB_DEVICE_INFO	Device	description of the device
DWORD	dwOptions	should be zero
CHAR	cName[32	name of card
CHAR	cDescription[100	description

WD_USB_RESET_PIPE

defines reset pipe command

Members:

TYPE	NAME	DESCRIPTION
DWORD	hDevice	handle of device
DWORD	dwPipe	number of pipe to reset

Chapter

9

WinDriver Enhanced Support for Specific PCI Chip Sets.

This chapter is relevant for you if you are using one of the PCI chip-sets for which WinDriver offers Enhanced support. This currently includes PLX 9050, 9054, 9060, 9080, Galileo gt64, V3 PBC and AMCC 5933. WinDriver supports all other PCI chip-sets are via DriverWizard and the regular WinDriver API.

Overview

In addition to the regular WinDriver API, described in former chapters, WinDriver also offers a custom API for specific PCI chip-sets – Currently including PLX, Galileo, V3, Altera, PLDA, QuickLogic and AMCC chip-sets.

The following is an overview of the development process when using WinDriver specific PCI API.

1. Run the custom diagnostics program to diagnose your card.

2. Locate your specific card diagnostic program. *See \WinDriver\chip_vendor\chip_name\xxxdiag\xxxdiag.c*

3. Use this source code as a skeleton for your device driver.

4. Modify the code to suit your application. Use your PCI chip specific function reference to add your own code. More help and details can be found on the WinDriver electronic reference manual.

5. If the User Mode driver you have created in the above steps contains some parts which in the performance must be enhanced (an interrupt handler for example), see the 'WinDriver Kernel PlugIn' chapter. There you will learn how to move parts of your source code to WinDriver's Kernel PlugIn, thereby eliminate any calling overhead, and achieving maximal performance.

What is The PCI Diagnostics program?

The diagnostics program is a ready-to-run sample diagnostics application for specific PCI chip-sets. The diagnostics program accesses the hardware via WinDriver's specific PCI API (xxxLIB.C). It is written as a console mode application, and not as a GUI application, to simplify the understanding of the source code of the diagnostics program. This will help you learn how to properly use the your specific API.

This application can be used as a skeleton for your device driver. If your driver is not a console mode application, just remove the printf() calls from the code (you may replace them with MessageBox() if you wish).

You may also find that the xxx_DIAG.C is both an example of using your specific API as well as a useful diagnostics utility.

Using Your PCI chip-set Diagnostics program

Introduction

The custom diagnostics program (xxx_DIAG.EXE) accesses the hardware using WinDriver. Therefore WinDriver must be installed before being able to

run xxx_DIAG. If WinDriver is installed correctly, at boot time a message will appear on screen displaying the WinDriver version installed.

Once WinDriver is running, you may run xxx_DIAG. Click Start | Programs | WinDriver | Samples | Chip_name Diagnostics.

The application will first try to locate the card, with the default VendorID and DeviceID assigned by your PCI chip vendor (for example – PLX 9054 - VendorID = 0x10b5, DeviceID = 0x9054). If such a card is found you will get a message "your PCI card found" ("PLX 9054 card found"). If you have programmed your EEPROM to load a different VendorID/DeviceID, then at the main menu you will have to choose your card (option 'Locate/Choose your board' in main menu).

Main Menu Options:

Scan PCI bus:
Displays all the cards present on the PCI bus and their resources. (IO ranges, Memory ranges, Interrupts, VendorID/DeviceID). This information may be used to choose the card you need to access.

Locate/Choose ***your*** board:
Chooses the active card that the diagnostics application will use. You are asked to enter the VendorID/DeviceID of the card you want to access. In case there are several cards with the same VendorID/DeviceID, you will be asked to choose one of them.

PCI configuration registers:
This option is available only after choosing an active card. A list of the PCI configuration registers and their READ value are displayed. These are general registers, common to all PCI cards. In order to WRITE to a register, enter its number, and then the value to write to it.

Your_PCI local registers:
This option is available only after choosing an active card. A list of your PCI registers and their READ value are displayed. In order to WRITE to a register, enter the register number, and then enter the value to write to it.

Access memory ranges on the board:

This option is available only after choosing an active card. Use this option carefully. Accessing memory ranges, accesses the local bus on your card -- If you access an invalid local address, or if you have any problem with your card (such as a problem with the IRDY signal), the CPU may hang.

- To access a local region, first toggle active mode between BYTE/WORD/DWORD, to fit the hardware you are accessing.
- To READ from a local address, choose 'Read from board'. You will be asked for local address to read from.
- To WRITE from a local address, choose 'Write from board'. You will be asked for local address to write to, and the data to write.

Both in board READ and WRITE, the address you give will also be used to set the base address register. For More detail see the electronic reference manual.

Enable / Disable interrupts:

This option will appear only if the card was set to open with interrupts. Choosing this item toggles the interrupt status (Enable / Disable). When interrupts are disabled, interrupts that the card generates are not intercepted by the application. If interrupts will be generated by the hardware while the interrupts are disabled by the application, the computer may 'hang'.

Access EEPROM device (Where available):

This option provides basic read/write access to the serial configuration EEPROM.
is available only after choosing an active card. This option assumes that the configuration EEPROM has initialised the Configuration Register, Aperture zero and one space to valid local

- To read an EEPROM location, choose Read a byte from serial EEPROM'. You will be asked for the address of the location to read from.

- To write an EEPROM location, choose 'Write a byte to serial EEPROM'. You will be asked for the address and the data to write.

Pulse Local Reset (where available):

This option provides a way to reset the local processor, from the host.

- To RESET the local host processor, choose 'Enter reset duration in milliseconds'. You will be asked for the time in milliseconds. Note: Resolution of delay time is based on PC timer tick, or approximately 55 milliseconds.

Creating your driver without using the PCI diagnostic code as a skeleton.

- Add xxxLIB.C to your project or your make file.
- Include "xxxlib.h" in your driver source code.

NOTE: In your \windriver\chip_vedor\chip_name\xxx_diag folder, you will find the source code for xxx_DIAG.EXE. Double click the 'mdp' file (which contains the project environment used to compile this code) in this directory to start your MSDEV with the proper settings for a project. You may use this as a skeleton for your code.

- Call Pxxx_Open() at beginning of your code to get a 'handle' to your card.
- After locating your card, you may read / write to memory, enable / disable interrupts, access your EEPROM and more, using the following functions *(Some of these functions are not available to all PCI chip-sets or have different grammar. It is highly recommended to review your specific PCI API on the electronic reference manual)*:

  xxx_IsAddrSpaceActive()

  xxx_GetRevision()

  xxx_ReadReg ()

  xxx_WriteReg ()

  xxx_ReadSpaceBlock()

  xxx_WriteSpaceBlock()

  xxx_ReadSpaceByte()

xxx_ReadSpaceWord()

xxx_ReadSpaceDWord()

xxx_WriteSpaceByte()

xxx_WriteSpaceWord()

xxx_WriteSpaceDWord()

xxx_ReadBlock()

xxx_WriteBlock()

xxx_ReadByte()

xxx_ReadWord()

xxx_ReadDWord()

xxx_WriteByte()

xxx_WriteWord()

xxx_WriteDWord()

xxx_IntIsEnabled()

xxx_IntEnable()

xxx_IntDisable()

xxx_DMAOpen()

xxx_DMAClose()

xxx_DMAStart()

xxx_DMAIsDone()

xxx_EEPROMRead()

xxx_EEPROMWrite()

xxx_ReadPCIReg()

xxx_WritePCIReg()

- Call xxx_Close() before end of code.

NOTES:

1. Using one of the sample drivers included with WinDriver as a skeleton for your code may shorten the development process. (see Sample Code 'topic)

2. APIs may slightly vary between PCI chips. See the electronic manual (Start Menu | Programs | WinDriver | WinDriver Manual) for details.

Sample Code

Sample uses of WinDriver for all PCI chip sets are supplied with the WinDriver toolkit.

You may find the WinDriver samples under \windriver\samples, and the WinDriver for PLX/Galileo/V3/AMCC samples under \windriver\chip_vendor. Each directory contains files.txt, which describes the various samples included.

Each sample is located in its own directory. For your convenience, we have supplied an 'mdp 'file alongside each 'c 'file, so that users of Microsoft's Developers Studio may double-click the mdp file and have the whole environment ready for compilation. (Users of different win32 compilers need to include the *.c files in their standalone console project, and include the xxx_lib.c in their project)

You may use the source of the diagnostic program described earlier to learn your PCI's specific API usage.

WinDriver's specific PCI chip-set API Function Reference

Use this section as a 'quick reference' to the WinDriver's specific PCI API functions. A more detailed reference (per chip) may be found in the WinDriver Help files.

Advanced users may find more functionality in the WinDriver's API.

All of the functions outlined in 'Function Reference' are implemented in the \windriver\chip_vendor\chip_name\lib\xxx_lib.c file.

For more detailed information on specific PCI chip set APIs see the electronic reference manual.

The definition of the structures used in the following functions may be found in the 'structure reference'

xxx_CountCards ()

Returns the number of cards on the PCI bus that have the given VendorID and DeviceID. This value can then be used when calling xxx_Open, to choose which board to open. Normally, only one board is in the bus and this function will return 1.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

Returns the number of matching PCI cards found.

Example

```
nCards = P9054_CountCards( 0x10b5, 0x9054 );
```

xxx_Open()

Used to open a handle to your card. If several cards with identical PCI chips are installed, the specific card to open may be specified by using 'xxx_CountCards' before using 'xxx_Open', and calling 'open' with a specific card number (See 'prototype' below)

If open is successful, function returns TRUE, and a handle to the card.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

TRUE if OK.

Example

```
if (!P9054_Open( &hPlx, 0x10b5, 0x9054,  0, P9054_OPEN_USE_INT ))
{
   printf( "Error opening device\n");
}
```

xxx_Close()

Closes WinDriver device. Must be called after finished using the driver.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

none

Example

```
P9054_Close(hPLX);
```

xxx_IsAddrSpaceActive()

Checks if the specified address space is enabled. The enabled address spaces

are determined by the EEPROM, which at boot time sets the memory ranges requests.

Use this function after calling xxx_Open() to make sure that the address space(s) that

your driver is going to use are enabled.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

TRUE if address space is enabled

Example

```
if ( !P9054_IsAddrSpaceActive(hPlx, P9054_ADDR_SPACE2) )
{
        printf ("Address space2 is not active!\n");
}
```

xxx_GetRevision()

Returns your PCI chip-set silicon revision.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return value

Returns the silicon revision.

xxx_ReadReg ()

xxx_WriteReg ()

Reads or writes to or from a specified register on the board.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

data read from register (for P9054_ReadReg() only).

xxx_ReadSpaceByte()

xxx_ReadSpaceWord()

xxx_ReadSpaceDWord()

xxx_WriteSpaceByte()

xxx_WriteSpaceWord()

xxx_WriteSpaceDWord()

Reads or writes a byte / word / dword from address space on board.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

For READ - Data read from board.
For WRITE - none.

xxx_ReadSpaceBlock()

xxx_WriteSpaceBlock()

Reads or writes a block to or from an address space on the board.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

none

xxx_ReadByte()

xxx_ReadWord()

xxx_ReadDWord()

xxx_WriteByte()

xxx_WriteWord()

xxx_WriteDWord()

Reads or writes a byte / word / dword from memory on board.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

For READ - Data read from board.
For WRITE - None.

xxx_ReadBlock()

xxx_WriteBlock()

Reads or writes a block of memory to or from the board.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

none

xxx_IntIsEnabled()

Checks whether interrupts are enabled or not.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

TRUE if interrupts are already enabled (e.g. if P9054_IntEnable() was called).

xxx_IntEnable()

Enable interrupt processing.

IMPORTANT NOTE: All PCI chip-sets use level sensitive interrupts, therefore you must edit the implementation of this function (found in your \windriver\chip_vendor\chip_name\lib\xxx_lib.c) to fit your specific hardware. The comments in the function will instruct you where your changes must be inserted. See more about this in the in the section regarding PCI interrupts implementation.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

TRUE if successful.

xxx_IntDisable()

Disable interrupt processing.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

none

xxx_DMAOpen()

Initialises the WD_DMA structure (see windrvr.h) and allocates a contiguous buffer

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

return value

Returns TRUE if DMA buffer allocation succeeds

xxx_DMAClose()

Frees the DMA handle, and frees the allocated contiguous buffer.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

xxx_DMAStart()

Start DMA to/from card.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return value

Returns TRUE if DMA transfer succeeds.

xxx_DMAIsDone()

Used to test if DMA is done. (Use when V3PBC_DMAStart was called with fBlocking == FALSE)

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return value

Returns TRUE if DMA transfer is completed.

xxx_PulseLocalReset()

Sends a reset signal to the card, for a period of 'wDelay' milliseconds.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

xxx_EEPROMRead()

xxx_EEPROMWrite()

Reads or writes one to the EEPROM. – Syntax and functionality may vary between different chip-sets. See the electronic reference manuals for your chip-set's exact syntax and usage.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return value

Returns TRUE if EEPROM write succeeds.

xxx_ReadPCIReg ()

xxx_WritePCIReg()

Read from or write to the PCI configuration registers.

Prototype and Parameters – See electronic reference manual for your PCI chip specific details.

Return Value

data read from configuration register (for xxx_ReadPCIReg only).

Structure Reference for WinDriver's specific PCI APIs

Structure reference for the PLX, Galileo, Altera, V3, PLDA, Galileo and AMCC specific APIs can be found in the electronic version of this manual.

Chapter
10

WinDriver Implementation Issues

This chapter contains instructions for performing operations that DriverWizard cannot automate.

If you are using a PCI chip set from PLX, Galileo, AMCC, or V3 – You do not have to read this chapter. WinDriver includes custom APIs built especially for these PCI chip-set vendors. These APIs save you the need to learn both the PCI internals and the chip-set's data sheets. Using these specific APIs – a DMA function is as simple as calling a function (i.e. P9054_DMAOpen(), P9054_DMAStart() and so on...).

Performing DMA.

If you are not using a PCI chip which is supported by WinDriver (currently - PLX, Galileo, V3 or AMCC) – these few pages will guide you through the steps of performing all kinds of DMA via WinDriver's API.

There are basically two methods to perform DMA - Contiguous Buffer DMA and Scatter/Gather DMA. Scatter/Gather DMA is much more efficient than contiguous DMA. This feature allows the PCI device to copy memory blocks from different addresses. This means that the transfer can be done directly to/from the user's buffer - that is contiguous in Virtual memory, but fragmented in the Physical memory. If your PCI device does not support

Scatter/Gather, you will need to allocate a Physically contiguous memory block, perform the DMA transfer to there, and then copy the data to your own buffer.

The programming of the DMA is specific for different PCI devices. Normally, you need to program your PCI device with the Local address (on your PCI device), the Host address (the Physical memory address on your PC), the transfer count (size of block to transfer), and then set the register that initiates the transfer.

Scatter/Gather DMA

Following is an outline of a DMA transfer routine for PCI devices that support Scatter/Gather DMA. More detailed examples can be found at:

1. \windriver\plx\9054\lib\p9054_lib.c
2. \windriver\plx\9080\ lib\p9080_lib.c
3. \windriver\galileo\gt64\ lib\gt64_lib.c

Note for Linux developers: Due to Linux's own limitations WinDriver does not yet support Scatter Gather DMA on this OS. This feature will be added to the Linux version of WinDriver as soon as the Linux kernel includes support for Scatter Gather DMA operations.

Sample DMA implementation:

```
BOOL DMA_routine(void *startAddress, DWORD transferCount,
  BOOL fDirection)
{
  WD_DMA dma;
  int i;

  BZERO (dma);
  dma.pUserAddr = startAddress;
  dma.dwBytes = transferCount;
  dma.dwOptions = 0;

  // lock region in memory
  WD_DMALock(hWD,&dma);
  if (dma.hDma==0)
```

```
    return FALSE;

  for(i=0;i!=dma.dwPages;i++)
  {
    // Program the registers for each page of the transfer
    My_DMA_Program_Page(dma.Page[i].pPhysicalAddr,
      dma.Page[i].dwBytes, fDir);
  }
  // write to the register that initiates the DMA transfer
  My_DMA_Initiate();
  // read register that tells when the DMA is done
  while(!My_DMA_Done());

  WD_DMAUnlock(hWD,&dma);
  return TRUE;
}
```

You should implement:

My_DMA_Program_Page() - Set the registers on your device that are part of the chained list of transfer addresses.

My_DMA_Initiate() - Set the start bit on your PCI device to initiate the DMA

My_DMA_Done() - Read the 'Transfer Ended' bit on your PCI device

Scatter/Gather DMA for buffers larger than 1MB

The WD_DMA structure holds a list of 256 pages. The x86 CPU uses 4K page size, so 256 pages can hold 256*4K = 1MB. Since the first and last page might not start (or end) on a 4096 byte boundary, 256 pages can hold 1MB - 8K.

If you need to lock down a buffer larger than 1MB, that needs more than 256 pages, you will need the DMA_LARGE_BUFFER option.

```
BOOL DMA_Large_routine(void *startAddress, DWORD transferCount, BOOL fDirection)
{
  DWORD dwPagesNeeded = transferCount / 4096 + 2;
  WD_DMA *dma = calloc(
    sizeof(WD_DMA)+sizeof(WD_DMA_PAGE)*dwPagesNeeded,1);
```

```
  dma->pUserAddr = startAddress;
  dma->dwBytes = transferCount;
  dma->dwOptions = DMA_LARGE_BUFFER;
  dma->dwPages = dwPagesNeeded;

  // lock region in memory
  WD_DMALock(hWD,&dma);

  // the rest is the same as in the DMA_routine()
  ...

  // free the WD_DMA structure allocated
  free (dma);
}
```

Contiguous Buffer DMA

More detailed examples can be found at:

1. windriver\v3\lib\pbclib.c
2. windriver\amcc\lib\amcclib.c

A read sequence (from the card to the mother-board's memory):

```
{
  WD_DMA dma;

  BZERO (dma);
  // allocate the DMA buffer (100000 bytes)
  dma.pUserAddr = NULL;
  dma.dwBytes = 10000;
  dma.dwOptions = DMA_KERNEL_BUFFER_ALLOC;
  WD_DMALock(hWD, &dma);
  if (dma.hDma==0)
    return FALSE;

  // transfer data from the card to the buffer
  My_Program_DMA_Transfer(dma.Page[0].pPhysicalAddr,
    dwLocalAddr);
  // Wait for transfer to end
  while(!My_Dma_Done());

  // now the data is the buffer, and can be used
```

```
  UseDataReadFromCard(dma.pUserAddr);

  // release the buffer
  WD_DMAUnlock(hWD,&dma);
}
```

A Write Sequence (from the mother-board's memory to the card):

```
{
  WD_DMA dma;

  BZERO (dma);
  // allocate the DMA buffer (100000 bytes)
  dma.pUserAddr = NULL;
  dma.dwBytes = 10000;
  dma.dwOptions = DMA_KERNEL_BUFFER_ALLOC;
  WD_DMALock(hWD, &dma);
  if (dma.hDma==0)
    return FALSE;

  // prepare data into buffer
  PrepareDataInBuffer(dma.pUserAddr);

  // transfer data from the buffer to the card
  My_Program_DMA_Transfer(dma.Page[0].pPhysicalAddr,
    dwLocalAddr);
  // Wait for transfer to end
  while(!My_Dma_Done());

  // release the buffer
  WD_DMAUnlock(hWD,&dma);
}
```

Handling Interrupts

Interrupts can easily be handled via DriverWizard. It is recommended that you use the DriverWizard to generate the interrupt code for you, by defining (or auto-detecting) your hardware's interrupt, and generating code. Use this section to understand the code DriverWizard generates for you or to write your own Interrupt handler.

General - Handling an interrupt

1. A thread that will handle incoming interrupts needs to be created.
2. The interrupt handler thread will run an infinite loop that will wait for an interrupt to occur.
3. When interrupt occurs, the driver's interrupt handler code is called.
4. When interrupt handler code returns, the wait loop continues.

The WD_IntWait() function, puts the thread to sleep until an interrupt occurs. There is no CPU consumption while waiting on an interrupt. Once an interrupt occurs, it is first handled by the WinDriver kernel, then the WD_IntWait() wakes up the interrupt handler thread and returns.

Since your interrupt thread runs in user-mode, you may call any Windows API function, including File handling and GDI functions.

Simple interrupt handler routine, for edge-triggered interrupts (normally ISA/EISA cards):

```
// interrupt structure
WD_INTERRUPT Intrp;

DWORD WINAPI wait_interrupt (PVOID pData)
{
  printf ("Waiting for interrupt");
  for (;;)
  {
```

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

      break; // WD_IntDisable called by parent
// call your interrupt routine here
      printf ("Got interrupt %d\n", Intrp.dwCounter);
}
  return 0;
}

void Install_interrupt()
{
  BZERO(Intrp);
  // put interrupt handle returned by WD_CardRegister
  Intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
  // no kernel transfer commands to do upon interrupt
  Intrp.Cmd = NULL;
  Intrp.dwCmds = 0;
  // no special interrupt options
  Intrp.dwOptions = 0;
  WD_IntEnable(hWD, &Intrp);
  if (!Intrp.fEnableOk)
  {
    printf ("Failed enabling interrupt\n");
    return;
  }

  printf ("starting interrupt thread\n");
  thread_handle = CreateThread (0, 0x1000,
      wait_interrupt, NULL, 0, &thread_id);

  // call your driver code here

  WD_IntDisable (hWD, &Intrp);
  WaitForSingleObject(thread_handle, INFINITE);
}
```

ISA/EISA and PCI interrupts

Generally, ISA/EISA interrupts are edge triggered, as opposed to PCI interrupts that are level sensitive. This difference has many implications on writing the interrupt handler routine.

Edge triggered interrupts are generated once, when the physical interrupt signal goes from low to high. Therefore, exactly one interrupt is generated. This makes the Windows OS to call the WinDriver kernel interrupt handler, that

released the thread waiting on the WD_IntWait() function. There is no special action that needs to take place in order to acknowledge this interrupt.

Level sensitive interrupts are generated as long as the physical interrupt signal is high. If the interrupt signal is not lowered by the end of the interrupt handling by the kernel, The Windows OS will call the WinDriver kernel interrupt handler again - This will cause the PC to hang!

To prevent this situation from happening, the interrupt must be acknowledged by the WinDriver kernel interrupt handler. Explanation on acknowledging level-sensitive interrupts may be found in Chapter 8 - 'Computer hangs on interrupt'.

Transfer commands at kernel-level (acknowledging the interrupt)

Usually, interrupt handlers for PCI cards (level sensitive interrupt handlers) need to perform transfer commands at the kernel to lower the interrupt level (acknowledge the interrupt).

To pass transfer commands to be performed in the WinDriver kernel interrupt handler, before WD_IntWait() returns, you must prepare an array of commands (WD_TRANSFER structure), and pass it to the WD_IntEnable() function.

```
WD_TRANSFER trans[2];
BZERO(trans);
trans[0].cmdTrans = RP_DWORD; // Read Port Dword
// address of IO port to write to
trans[0].dwPort = dwAddr;
trans[1].cmdTrans = WP_DWORD; // Write Port Dword
// address of IO port to write to
trans[1].dwPort = dwAddr;
// the data to write to the IO port
trans[1].Data.Dword = 0;
Intrp.dwCmds = 2;
Intrp.Cmd = trans;
Intrp.dwOptions =
   INTERRUPT_LEVEL_SENSITIVE | INTERRUPT_COPY_CMD;
WD_IntEnable(hWD, &Intrp);
```

This sample performs a DWORD read command from the IO address dwAddr, then it writes to the same IO port a value of '0'.

The INTERRUPT_COPY_CMD option is used to retrieve the value read by the first transfer command, before the write command is issued. This is useful when you need to read the value of a register, and then write to it to lower the interrupt level. If you will try to read this register after WD_IntWait() returns, it will already be '0' because the write transfer command was issued at kernel level.

```
DWORD WINAPI wait_interrupt (PVOID pData)
{
  printf ("Waiting for interrupt");
  for (;;)
  {
    WD_TRANSFER trans[2];
    Intrp.dwCmds = 2;
    Intrp.Cmd = trans;
    WD_IntWait (hWD, &Intrp);
    if (Intrp.fStopped)

      break; // WD_IntDisable called by parent
                    // call your interrupt routine here
      printf (
        "Got interrupt %d. Value of register read %x\n",
        Intrp.dwCounter, trans[0].Data.Dword);
}
  return 0;
}
```

Interrupts in Windows CE

Windows CE uses a logical interrupt scheme rather than the physical interrupt number. It maintains an internal kernel table that maps the physical IRQ number to the logical IRQ number. Device drivers are generally expected to get the logical interrupt number after having ascertained the physical interrupt. This is handled internally by WinDriver so programmers using WinDriver need not worry about this issue. However, the X86 CEPC builds provided with the ETK do not provide interrupt mappings for certain reserved interrupts including the following:

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

An attempt to initialise and use any of these interrupts will fail. In case you wish to use any of these interrupts - (e.g.: you do not want to use the PPSH and you want to reclaim the parallel port for the some other purpose) - you should modify the file CFWPC.C that is found in the directory %_TARGETPLATROOT%\KERNEL\HAL to include code as shown below that sets up a value for the interrupt 7 in the interrupt mapping table. You will then need to rebuild the Windows CE image NK.BIN and download the new executable onto your target platform.

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

For non-X86 machines like the handheld PCs from HP and Sharp, the developer should use the logical interrupt id which can be got from the platform specific header file NKINTR.H

A complete discussion of this procedure is outside the scope of this manual. Please refer to the ETK documentation for more details.

PCMCIA interrupts in Windows CE

Windows CE handles PCMCIA interrupts differently than PCI and ISA/EISA interrupts. WinDriver handles the setting up of the PCMCIA interrupt internally by calling the Card Services API so this process is transparent to the developer. The WD_PcmciaGetCardInfo() function automatically sets up the interrupt items and registers the interrupt.

Chapter 11

Improving performance

Improving the performance of your device driver - overview

Once your User Mode driver has been written and debugged, you might find that certain modules in your code do not operate fast enough (for example - an interrupt handler or accessing IO mapped regions). If this is the case, try to improve the performance by one of the two ways suggested in this chapter.

1. Improve the performance of your User Mode driver.

2. Move the performance critical parts of your code in to the WinDriver's "Kernel PlugIn".

Note that the Kernel PlugIn is not implemented under Windows CE since in this OS there is no separation between kernel mode and user mode, thus top performance can be achieved without using the Kernel PlugIn. Use the checklist below to determine how the performance should be improved in your driver.

Performance improvement checklist

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

1. Create your driver in the User Mode as explained in the previous chapters of this manual.

2. Compile and debug your driver in the User Mode.

3. When working in the User Mode, performance may take a hit. Check if you have performance problems. If you do not have any performance problems, you have finished your driver development.

If you do have performance problems:

4. Identify which part of the code the performance problem is at. Classify and solve the problem according to the table below:

	Problem	Solution
#1	ISA Card - Accessing an IO mapped range on the card.	• Try to convert multiple calls to WD_Transfer() to one call to WD_MultiTransfer() *(See the 'Improving performance - Accessing IO mapped regions' section later in this chapter).* • If this does not solve the problem, handle the IO at Kernel Mode, by writing a kernel PlugIn. *(See the Kernel PlugIn 'chapter for details)*
#2	PCI Card - Accessing an IO mapped range on the card.	• First, try to change the card from IO mapped to memory mapped by changing bit 0 of the address space PCI configuration register to 0, and then try the solutions for problem #3. You will probably need to re-program the EEPROM to initialise BAR0/1/2/3/4/5 registers with different values. • If this is not possible, try the solutions suggested for problem #1. • If this does not solve the problem, handle the IO at Kernel Mode, by writing a kernel PlugIn. *(See the Kernel PlugIn 'chapter for details)*

	Problem	Solution
#3	Accessing a memory mapped range on the card.	• Try to access memory without using WD_Transfer(), by using direct access to memory mapped regions *(See the 'Improving Performance - using direct access to memory mapped regions 'section later in this chapter).* • If this does not solve the problem, then there is a hardware design problem. You will not be able to increase performance by using any software design method, or by writing a Kernel PlugIn, or even by writing a full kernel driver.
#4	Interrupt latency. (Missing interrupts, Receiving interrupts too late)	You need to handle the interrupts at Kernel Mode, by writing a kernel PlugIn. *(See the Kernel PlugIn 'chapter for details)*
#5	USB devices: Slow transfer rate	To increase the transfer rate try to increase the packet size by choosing a different device configuration. If there is need to do many small transfers, the Kernel-Plugin can be used.

Improving the performance of your User Mode driver

As a general rule, transfers to memory mapped regions are faster than transfers to IO mapped regions. The reason is that WinDriver enables the user to

directly access the memory mapped regions, without calling the WD_Transfer() function.

Using Direct access to memory mapped regions

After registering a memory mapped region, via WD_CardRegister(), two results are returned: dwTransAddr and dwUserDirectAddr. dwTransAddr should be used as a base address when calling WD_Transfer() to read or write to the memory region. A more efficient way to perform memory transfers would be to use dwUserDirectAddr directly as a pointer, and access with it the memory mapped range. This method enables you to read/write data to your memory-mapped region without any function calls overhead (i.e. Zero performance degradation).

Accessing IO mapped regions

They only way to transfer data on IO mapped regions is by calling WD_Transfer() function. If a large buffer needs to be transferred, the String (Block) Transfer commands can be used. For example: RP_SBYTE - Read Port String Byte command will transfer a buffer of Bytes to the IO port. In such a case the function calling overhead is negligible compared to the block transfer time.

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

An example for such a case could be: A block of 1MB of data needs to be transferred Word by Word, where in each word that is transferred, first the LOW byte is transferred to IO port 0x300, then the HIGH byte is transferred to IO port 0x301.

Normally this would mean calling WD_Transfer() 1 million times - Byte 0 to port 0x300, Byte 1 to port 0x301, Byte 2 to port 0x300 Byte 4 to port 0x301 etc (WP_BYTE - Write Port Byte).

A quick way to save 50% of the function call overhead would be to call WD_Transfer() with a WP_SBYTE (Write Port String Byte), with two bytes at a time. First call would transfer Byte0 and Byte1 to ports 0x300 and 0x301,

Second call would transfer Byte2 and Byte3 to ports 0x300 and 0x301 etc. This way, WD_Transfer() will only be called 500,000 times to transfer the block.

The third method would be by preparing an array of 1000 WD_TRANSFER commands. Each command in the array will have a WP_SBYTE command that transfers two bytes at a time. Then you call WD_MultiTransfer() with a pointer to the array of WD_TRANSFER commands. In one call to WD_MultiTransfer() - 2000 bytes of data will be transferred. To transfer the 1MB of data you will need only 500 calls to WD_Transfer(). This is 0.5% of the original calls to WD_Transfer(). The trade off in this case is the memory that is used to set-up the 1000 WD_TRANSFER commands.

Chapter
12

WinDriver Kernel PlugIn Overview

This chapter will provide you with a brief description of the "Kernel PlugIn" Feature of WinDriver.

Background

Creating your driver in the User Mode has some distinct advantages:

1. No kernel knowledge necessary.
2. No need to learn the MS DDK or Kernel debuggers.
3. Easy development / debugging using standard development / debugging tools.

However, this architecture imposes some function call overhead from the kernel to the User Mode. In most cases, this is not a problem. In other cases, this performance hit may cause the driver not to function as needed.

When performance is a problem, WinDriver's Kernel PlugIn feature allows you to move only the performance critical section of your code to the kernel, while

keeping most of the code intact as it is. The advantages of writing a Kernel PlugIn driver over a Kernel Mode driver are:

1. All of the driver code is written and debugged in the User Mode.

2. The code segments that are moved to the Kernel Mode remain essentially the same, therefore no kernel debugging is needed.

3. The parts of your code that will run in the kernel through the Kernel PlugIn are platform independent, and therefore will run on all platforms supported by WinDriver. A standard Kernel Mode driver will run only on the platform it was written for.

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

Do I need to write a Kernel PlugIn?

Not every performance problem requires you to write a Kernel PlugIn. Some performance problems may be overcome in the User Mode driver, by better utilisation of the features WinDriver provides.

To determine how to solve your performance issues, please see the chapter titled "Improving Performance". This chapter contains a table which will help you determine how to solve your performance problem. In some cases a quick User Mode solution is provided, and in other cases, it determines that a Kernel PlugIn driver must be written.

If you need to write a Kernel PlugIn, continue reading the following chapter.

What kind of performance can I expect?

Since you can write your own interrupt handler in the kernel with WinDriver Kernel PlugIn, you can expect to handle more than 10,000 interrupts without missing any of them.

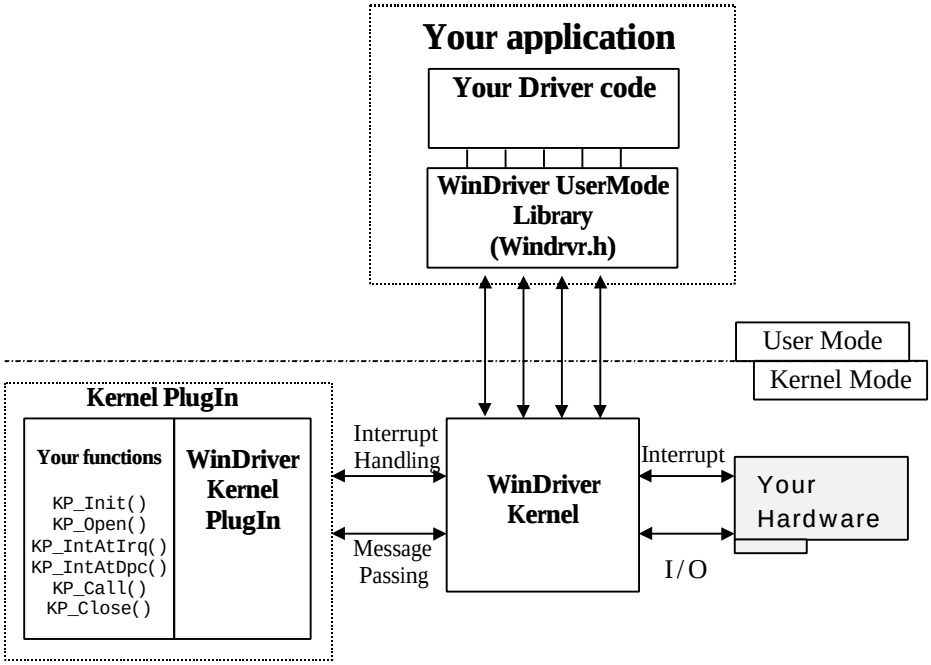
Overview of the development process

Using the WinDriver Kernel PlugIn, the developer first develops and debugs the driver in the User Mode with the standard WinDriver tools. After identifying the performance critical parts of the code (such as the interrupt handler, access to I/O mapped memory ranges, or slow data transfer rate through the USB pipes, etc.), the developer can 'drop' these parts of code into the WinDriver's Kernel PlugIn (which runs in the Kernel Mode), thereby eliminating the calling overhead. This unique feature allows the developer to start with quick and easy development in the User Mode, and progress to performance oriented code only where needed. This unique architecture saves time, and allows for absolutely zero performance degradation.

Chapter

13

WinDriver Kernel PlugIn Architecture



A driver written in User Mode, uses WinDriver's functions ("WD_xxx" functions) for the device access. If a certain function in the User Mode needs to achieve kernel performance (the interrupt handler for example), that function is moved to the WinDriver Kernel PlugIn. The code will still work "As Is", since WinDriver exposes its WD_xxx interface to the Kernel PlugIn as well.

There are two types of interaction between the WinDriver Kernel and the WinDriver Kernel PlugIn which will be discussed in detail in the next chapter. They are:

1. **Interrupt handling**: When WinDriver receives an interrupt, it will activate the interrupt handler in the User Mode driver by default. However, if the interrupt was set to be handled by the WinDriver Kernel PlugIn, then once the WinDriver receives the interrupt, it will be processed by your interrupt function in the Kernel. This is the same code that you wrote and debugged in the User Mode interrupt handler before.

2. **Message passing**: To execute functions in the Kernel Mode (such as I/O processing functions), the user mode driver simply passes a "message" to the WinDriver Kernel PlugIn. This message is mapped to a specific function, which is then executed in the kernel. This function contains the same code as it did when it was written and debugged in the User Mode.

At the end of your Kernel PlugIn development cycle, you will have the following elements to your driver:

1. Your User Mode driver - written with the WD_xxx functions.

2. The WinDriver Kernel - Windrvr.sys or Windrvr.vxd and wdusb.sys for USB drivers

3. Your Kernel PlugIn - <Your Driver Name>.sys or <Your Driver Name>.vxd - this is the driver that contains the functionality which you have chosen to bring down to the Kernel level.

Typical event sequence when using a Kernel PlugIn

The following is a typical event sequence, which covers all of the functions that you can implement in your Kernel PlugIn (KP):

Event / Callback	**Remarks**
Event: Windows loads your Kernel PlugIn driver	At boot time or by dynamic loading, or as instructed by the registry.
KP Call-back: Your KP_Init() Kernel PlugIn function is called	KP_Init() informs WinDriver what the name of your KP_Open() routine is. WinDriver will call this routine when the application wishes to open your driver (when it calls WD_KernelPlugInOpen())
Event: Your app (User Mode driver) calls WD_KernelPlugInOpen()	
KP Call-back: Your KP_Open() routine is called.	In your KP_Open() function, you inform WinDriver of the name of all of the call-back functions that you have implemented in your KP driver, and initiate the KP driver if needed.

Event: Your app calls WD_KernelPlugInCall()	Your app calls WD_KernelPlugInCall() to run code in the Kernel Mode (in the KP driver). The app passes a message to the KP driver. The KP driver will select which function to execute according to the message sent.
KP Call-back: Your KP_Call() routine is called	Executes code according to the message passed to it from the User Mode.
Event: Your hardware creates an interrupt	
KP Call-back: Your KP_IntAtIrql() routine is called. (If the KP interrupts are enabled)	KP_IntAtIrql() runs at high priority, and therefore should only do the basic interrupt handling (such as lowering the HW interrupt signal). If more interrupt processing is needed, it is deferred to the KP_IntAtDpc() function. If your KP_IntAtIrql() function returns a value greater than 0, your KP_IntAtDpc() function is called.
Event: KP_IntAtIrql() function returns a value greater than 0	Needs interrupt code to be processed as a Deferred procedure call in the Kernel.

KP Call-back: KP_IntAtDpc() is called.	Processes the rest of the interrupt code, but at a lower priority than KP_IntAtIrql.
Event: KP_IntAtDpc() returns a value greater than 0.	Needs interrupt code to be processed in the User Mode as well.
KP Call-back: WD_IntWait() returns.	Execution resumes at your User Mode interrupt handler.

Chapter

14

Kernel PlugIn – How it works

This chapter takes you through the development cycle of a Kernel PlugIn. It assumes that you have already written and debugged your entire driver code in the User Mode, and have encountered a performance problem.

Minimal requirements for creating a Kernel PlugIn

To compile the Kernel Mode driver you need the VC compiler (cl.exe, rc.exe and link.exe and nmake.exe).

To create a Win95 driver (VXD) you do not need the Win95 DDK.

To create a WinNT driver (SYS) you need the WinNT DDK, for the following file: ntoskrnl.lib.

NT DDK can be downloaded (Free) at
http://www.microsoft.com/hwdev/ddk/ddk40.htm.

Directory structure for the WinDriver Kernel PlugIn

\windriver\kerplug

\windriver\kerplug\lib - includes the files needed to link your Kernel PlugIn

\windriver\kerplug\kptest - contains a sample minimal Kernel PlugIn driver. KPTest_com.h contains common definitions such as messages, between the Kernel PlugIn and the User-mode.

\windriver\kerplug\kptest\usermode - User-mode part of the driver

\windriver\kerplug\kptest\kermode - Kernel PlugIn driver

Kernel PlugIn implementation

1. BEFORE YOU BEGIN

The following functions are call-back functions which you will implement in your Kernel PlugIn driver, and which will be called when their 'calling' event occurs. For example, KP_Init() is the call-back function which is called when the driver is loaded. Any code that you want to execute upon loading should be in this function.

In KP_Init(), the name of your driver is given. From then on, all of the call-backs which you implement in the kernel will contain your driver's name. For example, if your driver's name is MyDriver, then your 'Open' call-back will be called MyDriver_Open(). It is the convention of this reference guide to mark these functions as KP_ functions - i.e. the 'Open' function will be written here as KP_Open(), where the KP replaces your driver's name.

In your kernel driver you should implement the following function:

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

where KP_INIT is the following structure:

```
typedef struct {
   DWORD          dwVerWD;   // Version of WinDriver library WD_KP.LIB
   CHAR cDriverName[9];   // The device driver name, up to 8 chars.
   KP_FUNC_OPEN funcOpen; // The KP_Open function
} KP_INIT;
```

This function is called once, when the driver is loaded. The kpInit structure should be filled out with the KP_Open function and the name of your kernel plug in. (see example in KPTest.c). Note that the name that you choose for your KP driver (by setting it in the kpInit structure), should be the same name as the driver you are creating. For example, if you are creating a driver called ABC.VXD or ABC.SYS, then you should pass the name ABC in the kpInit structure.

From the KPTest Sample:

```
BOOL __cdecl KP_Init(KP_INIT *kpInit)
{
   // check if the version of WD_KP.LIB is the same version
   //as WINDRVR.H and WD_KP.H
   if (kpInit->dwVerWD!=WD_VER)
   {
      // you need to re-compile your kernel plugin with
      // the compatible version of WD_KP.LIB, WINDRVR.H
      // and WD_KP.H!
      return FALSE;
   }

   kpInit->funcOpen = KPTest_Open;
   strcpy (kpInit->cDriverName, "KPTest");

   return TRUE;
}
```

3. WRITE YOUR KP_OPEN() FUNCTION

In your Kernel PlugIn file, implement the KP_Open() function, where KP is the name of your KP driver (copied to kpInit->cDriverName in the KP_Init() function.

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

This call-back is called when the User Mode application calls the WD_KernelPlugInOpen() function.

In the KP_Open() function, define the call-backs that you wish to implement in the Kernel PlugIn.

Following is a list of the call-backs which can be implemented:

Call-back name	Functionality
KP_Close()	Called when the User Mode application calls the WD_KernelPlugInClose() function.
KP_Call()	Called when the User Mode application calls the WD_KernelPlugInCall() function. This function is a message handler for your utility functions.
KP_IntEnable()	Called when the User Mode application calls the WD_IntEnable() function. This function should contain any initialization needed for your Kernel PlugIn interrupt handling.
KP_IntDisable()	Called when the User Mode application calls the WD_IntDisable() function. This function should free any memory which was allocated in the KP_IntEnable() callback.

KP_IntAtIrql()	Called when WinDriver receives an interrupt. This is the function that will handle your interrupt in the Kernel Mode.
KP_IntAtDpc()	Called if the KP_IntAtIrql() callback has requested deffered handling of the interrupt (by returning with a value of TRUE).

These handlers will later be called when the user-mode program opens a KP driver (WD_KernelPlugInOpen(), WD_KernelPlugInClose()), sends a message (WD_KernelPlugInCall()), or installs an interrupt where hKernelPlugIn passed to WD_IntEnable() is of a Kernel PlugIn driver opened with WD_KernelPlugInOpen().

From the KPTest Sample:

```
BOOL __cdecl KPTest_Open(KP_OPEN_CALL *kpOpenCall, PVOID
   pOpenData, PVOID *ppDrvContext)
{
   kpOpenCall->funcClose = KPTest_Close;
   kpOpenCall->funcCall = KPTest_Call;
   kpOpenCall->funcIntEnable = KPTest_IntEnable;
   kpOpenCall->funcIntDisable = KPTest_IntDisable;
   kpOpenCall->funcIntAtIrql = KPTest_IntAtIrql;
   kpOpenCall->funcIntAtDpc = KPTest_IntAtDpc;

   *ppDrvContext = NULL; // you can allocate memory here

   return TRUE;
}
```

4. WRITE THE REST OF THE KERNEL PLUGIN CALL-BACKS

Add your specific code inside the call backs routines.

KPTest – A sample Kernel PlugIn Driver

The KPTest directory (\windriver\kerplug\KPTest) contains a sample minimal Kernel PlugIn driver which you can compile and execute. Use this sample as a skeleton for your Kernel PlugIn driver.

This sample builds KPTest.VXD and KPTest.SYS, and KPTest.EXE. The sample demonstrates communication between your application (KPTest.EXE) and your Kernel PlugIn (KPTest.VXD or KPTest.SYS).

The KPTest sample in this directory, is a Kernel PlugIn which implements a "Get Version" function, to demonstrate passing data (messages) to/from the Kernel PlugIn. It also implements an interrupt handler in the kernel. This Kernel PlugIn is called by the User Mode driver called KPTest.EXE.

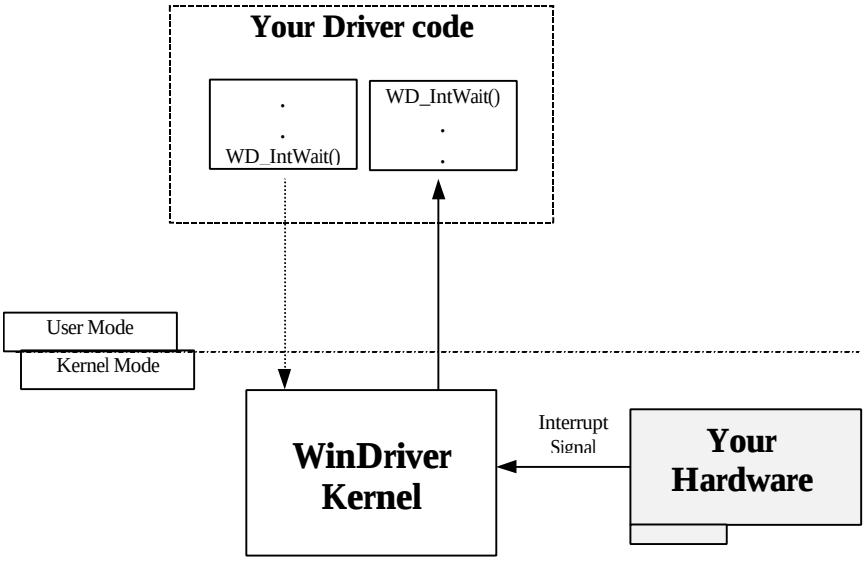
> **To check that you are ready to build a Kernel PlugIn driver, it is recommended to build and run this project first, before continuing to write your own Kernel PlugIn.**

Interrupt handling in the Kernel PlugIn

Interrupts will be handled by the Kernel PlugIn, if a Kernel PlugIn handle was passed to WD_IntEnable() by the User Mode application when it enabled the interrupt. When WinDriver receives a hardware interrupt, it calls the KP_IntAtIrql() (if Kernel PlugIn interrupts are enabled). In the KPTest sample, the interrupt handler running in the Kernel PlugIn counts 5 interrupts, and notifies the user-mode only of one out of each 5 incoming interrupts. This means that WD_IntWait() (in the user-mode) will return only on one out of 5 incoming interrupts.

Interrupt handling in the User Mode (Without Kernel PlugIn)

If the Kernel PlugIn interrupt handle is NOT enabled, then each incoming interrupt will cause WD_IntWait() to return. See drawing below:



Interrupt handling in the Kernel (With the Kernel PlugIn)

To instruct the interrupts to be handled by the Kernel PlugIn, the Kernel PlugIn handle must be given as a parameter to the WD_IntEnable() function. This enables the Kernel PlugIn interrupt handler.

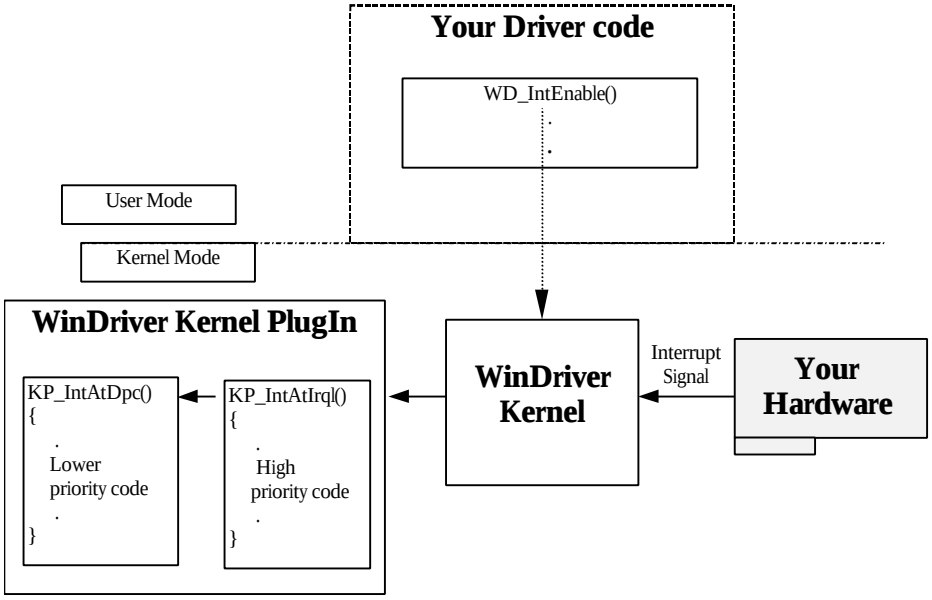
If the Kernel PlugIn interrupt handler is enabled, then KP_IntAtIrql() will be called on each incoming interrupt. The code in the KP_IntAtIrql() function is executed at IRQL. While this code is running, the system is halted (i.e. there will be no context switch and no lower priority interrupts will be handled). The code in the KP_IntAtIrql() function is limited to the following restrictions:

1. You may only access non pageable memory.

2. You may only call the following functions: WD_Transfer(), specific DDK functions which are allowed to be called from an IRQL.

3. You may not call malloc(), free(), or any WD_xxx command (other than WD_Transfer()).

Therefore, the code in KP_IntAtIrql() should be kept to a minimum, while the rest of the code that you want to run in the interrupt handler should be written in the KP _IntAtDpc(), which is called after IRQL finishes. The code in KP_IntAtDpc() is not limited by the above restrictions.



Message passing

The WinDriver architecture enables calling a Kernel Mode function from the User Mode by passing a message through the WD_KernelPlugInCall() function. The messages are defined by the developer in KP_comm.h. Upon receiving the message, WinDriver Kernel PlugIn executes the KP_Call function which maps a function to this message.

In the KPTest sample, the GetVersion function is a simple function which returns an arbitrary integer and string (which simulates your KPTest's version). This function will be called by the Kernel PlugIn, whenever the Kernel PlugIn receives a 'GetVersion' message from the KPTest.EXE.

The KPTest.EXE sends the message using the WD_KernelPlugInCall() function.

Chapter
15

Writing a Kernel PlugIn – Step by step instructions

The Kernel PlugIn directory (\windriver\kerplug) contains a sample Kernel PlugIn driver called KP_Test. The sample demonstrates communication between your application (KPTest.EXE) and your Kernel PlugIn (KPTest.VXD or KPTest.SYS).

The easiest way to write a Kernel PlugIn driver is to use this example as the skeleton code for your driver.

The following is a step by step guide to creating your kernel driver. The KPTest sample code will be used to demonstrate the different stages:

Determining whether a Kernel PlugIn is needed

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

2. Determine whether a Kernel PlugIn should be written by consulting the "Improving Performance" chapter.

Preparing the User Mode source code:

3. Isolate the function or functions that you need to move into the Kernel PlugIn.

4. Remove any platform specific code from the function. Use only the WinDriver functions which may be used from the kernel as well (See details later on in this chapter).

5. Compile and debug your driver in the User Mode again, to see that your code still works after these changes are made.

Creating a new Kernel PlugIn Project (Modifying the KP_Test sample for your needs)

6. Make a copy the KPTest directory. For example, to create a new project called MyDrv, copy \windriver\kerplug\KPTest to \windriver\kerplug\MyDrv.

7. Change all instances of KPTest in the all files in your new directory to MyDrv. (You may use the 'find in files' option in MSDEV for this).

8. Change all KPTest in file names to MyDrv.

Creating a handle to the WinDriver Kernel PlugIn in your User Mode driver

10. In your original User Mode source code, call WD_KernelPlugInOpen() at the beginning of your code, and WD_KernelPlugInClose() before terminating.

Interrupt Handling in the Kernel PlugIn

11. When calling WD_IntEnable(), give the handle to the Kernel PlugIn that you received from opening the Kernel PlugIn

12. Move the source code in the User Mode interrupt handler to the Kernel PlugIn, by moving some of it to KP_IntAtIrql() and some of it to KP_IntAtDpc() (See previous 'Interrupt handling in the kernel' section).

IO handling in the Kernel PlugIn

13. Move your IO handling code from the User Mode to KP_Call(). To call this code in the kernel from the User Mode, use WD_KernelPlugInCall(), with the Kernel PlugIn handle, and a message for each of the different functionalities you need. For each functionality, create a different message. Define these messages in the file KPTest_Com.H, which is a common header file, between the kernel-mode and the user-mode. This file should have the message definitions (IDs) and data structures used to communicate between the kernel-mode and user-mode.

Compiling your Kernel PlugIn Driver

14. Run compile.bat to compile and link your KP driver. In this example, run \windriver\kerplug\MyDrv\kermode\compile.bat. This will create two files: MyDrv.SYS and MyDrv.VXD

15. Copy the relevant file (MyDrv.SYS or MyDrv.VXD) to your drivers directory (For Win95, copy it to the c:\windows\system\vmm32 directory. For WinNT copy it to the c:\winNT\system32\drivers directory).

16. For Windows NT ONLY: Run 'WDReg -name mydrv -file mydrv install' (see instructions later) to register your driver, so that WinNT will load your driver.

The Kernel PlugIn driver is dynamically loadable, therefore you will not need to re-boot in order to run your driver.

Chapter
16

Kernel PlugIn Function reference

User Mode functions

The following functions are the User Mode functions which initiate the Kernel PlugIn's operation, and activate its call-backs.

WD_KernelPlugInOpen()

This function is used to obtain a valid handle for the Kernel PlugIn.

Prototype

void WD_KernelPlugInOpen(HANDLE hWD, WD_KERNEL_PLUGIN *pKernelPlugIn);

Parameters (WD_KERNEL_PLUGIN elements)

hKernelPlugIn - returns the handle of the Kernel PlugIn.
pcDriverName - name of Kernel PlugIn to load, up to 8 chars
pcDriverPath - file name of Kernel PlugIn to load. If NULL the driver will be searched in the
windows system directory.
with the name in pcDriverName.
pOpenData - pointer to data that will be passed to KP_Open() callback in the Kernel PlugIn.

Return Value

none

Example

```
// Handle to the KernelPlugIn
WD_KERNEL_PLUGIN kernelPlugIn;

BZERO (kernelPlugIn);
// Tells WinDriver which driver to open
kernelPlugIn.pcDriverName = "KPTEST";
// Opens KPTEST.SYS or KPTEST.VXD
WD_KernelPlugInOpen(hWD, &kernelPlugIn);

if (!kernelPlugIn.hKernelPlugIn)
{
   printf ("There was an error loading driver "
      "%s\n", kernelPlugIn.pcDriverName);
   return ;
}
printf("Kernel PlugIn opened\n");
```

WD_KernelPlugInClose()

Closes the WinDriver Kernel PlugIn handle obtained from WD_KernelPlugInOpen().

Prototype

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

Parameters (WD_KERNEL_PLUGIN elements)

hKernelPlugIn - handle of the Kernel PlugIn to close.

Return Value

none

Example

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

WD_KernelPlugInCall()

Calls a routine in the Kernel PlugIn to be executed.

Calling the WD_KernelPlugInCall() function in the User Mode, calls your KP_Call() callback function in the Kernel Mode. Your KP_Call() function in the Kernel PlugIn will decide what routine to execute according to the message passed to it in the WD_KERNEL_PLUGIN_CALL structure.

Prototype

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

Parameters (WD_KERNEL_PLUGIN_CALL elements)

```
hKernelPlugIn - handle of the Kernel PlugIn.
dwMessage - message ID to pass to KP_Call() callback.
pData - pointer to data to pass to KP_Call() callback.
dwResult - value set by KP_Call() callback.
```

Return Value

none

Example

```
WD_KERNEL_PLUGIN_CALL kpCall;

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

kpCall.pData = &mydrvVer;  // The data to pass to the call.
WD_KernelPlugInCall(hWD, &kpCall);
```

WD_IntEnable()

If the handle passed to this function is of a Kernel PlugIn, then that Kernel PlugIn will handle the interrupts.

In this case, upon receiving the interrupt, your Kernel Mode KP_IntAtIrql() function will execute. If this function returns a value greater than 0, then your deferred procedure call, KP_IntAtDpc() , will be called.

Prototype

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

Parameters (WD_INTERRUPT elements)

kpCall - information on Kernel PlugIn to install as interrupt handler.
kpCall.hKernelPlugIn - handle of Kernel PlugIn. if zero, then no Kernel PlugIn interrupt handler is installed.
kpCall.dwMessage - message ID to pass to KP_IntEnable() callback.
kpCall.pData - pointer to data to pass to KP_IntEnable() callback.
kpCall.dwResult - value set by KP_IntEnable() callback.

For information about all other parameters of WD_IntEnable(), see the documentation of WD_IntEnable() in WinDriver Function Referance chapter.

Return Value

none

Example

```
WD_INTERRUPT Intrp;

BZERO(Intrp);
Intrp.hInterrupt = hInterrupt; // from WD_CardRegister()
Intrp.Cmd = NULL;
Intrp.dwCmds = 0;
Intrp.dwOptions = 0;
// from WD_KernelPlugInOpen()
Intrp.kpCall.hKernelPlugIn = hKernelPlugIn;
WD_IntEnable(hWD, &Intrp);
if (!Intrp.fEnableOk)
```

```
    printf ("failed enabling interrupt\n");
```

Kernel functions

The following functions are call-back functions which you will implement in your Kernel PlugIn driver, and which will be called when their 'calling' event occurs. For example, KP_Init() is the call-back function which is called when the driver is loaded. Any code that you want to execute upon loading should be in this function.

In KP_Init(), the name of your driver is given, and its call-backs. From then on, all of the call-backs which you implement in the kernel will contain your driver's name. For example, if your driver's name is MyDriver, then your KP_Open call-back may be called MyDriver_Open(). It is the convention of this reference guide to mark these functions as KP_ functions - i.e. the 'Open' function will be written here as KP_Open(), where the KP replaces your driver's name.

KP_Init()

You must define the KP_Init() function in your code in order to link the Kernel PlugIn driver to the WinDriver.

KP_Init() is called when the driver is loaded. Any code that you want to execute upon loading should be in this function.

Prototype

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

Parameters

kpInit - structure to fill in the address of the KP_Open() callback function.

Return Value

TRUE if successful. If FALSE, then the Kernel PlugIn driver will be unloaded.

Example

```
BOOL __cdecl KP_Init(KP_INIT *kpInit)
{
   // check if the version of WD_KP.LIB is the same
   // version as WINDRVR.H and WD_KP.H
   if (kpInit->dwVerWD!=WD_VER)
   {
      // you need to re-compile your kernel plugin
      // with the compatible version of WD_KP.LIB,
      // WINDRVR.H and WD_KP.H!
      return FALSE;
   }

   kpInit->funcOpen = KP_Open;
   strcpy (kpInit->cDriverName, "KPTEST"); // until 8 chars

   return TRUE;
}
```

KP_Open()

Called when WD_KernelPlugInOpen() is called from the User Mode. The pDrvContext returned will be passed to rest of the functions

Prototype

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

Parameters

kpOpenCall - structure to fill in the addresses of the KP_xxxx() callback functions.
hWD - handle of WinDriver that WD_KernelPlugInOpen() was called with.
pOpenData - pointer to data, passed from user-mode.
ppDrvContext - pointer to driver context data that KP_Close(), KP_Call() and KP_IntEnable() functions will be called with. Use this to keep driver specific information.

Return Value

TRUE if successful. If FALSE, then WD_KernelPlugInOpen() call from user-mode will fail.

Example

```
BOOL __cdecl KP_Open(KP_OPEN_CALL *kpOpenCall, HANDLE hWD,
   PVOID pOpenData, PVOID *ppDrvContext)
{
   kpOpenCall->funcClose = KP_Close;
   kpOpenCall->funcCall = KP_Call;
   kpOpenCall->funcIntEnable = KP_IntEnable;
   kpOpenCall->funcIntDisable = KP_IntDisable;
   kpOpenCall->funcIntAtIrql = KP_IntAtIrql;
   kpOpenCall->funcIntAtDpc = KP_IntAtDpc;

   *ppDrvContext = NULL; // you can allocate here memory

   return TRUE;
}
```

KP_Close()

Called when WD_KernelPlugInClose() is called from the User Mode.

Prototype

```
void __cdecl KP_Close(PVOID pDrvContext);
```

Parameters

pDrvContext - driver context data that was set by KP_Open().

Return Value

none

Example

```
void __cdecl KP_Close(PVOID pDrvContext)
{
   // you can free here the memory allocated pDrvContext
}
```

KP_Call()

Called when the User Mode application calls the WD_KernelPlugInCall() function. This function is a message handler for your utility functions.

Prototype

```
void __cdecl KP_Call(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall);
```

Parameters

pDrvContext - driver context data that was set by KP_Open().
kpCall - structure with information from WD_KernelPlugInCall().
kpCall.dwMessage - message ID passed from WD_KernelPlugInCall().
kpCall.pData - pointer to data passed from WD_KernelPlugInCall().
kpCall.dwResult - value to return to WD_KernelPlugInCall().

Return Value

none

Example

```
void __cdecl KP_Call(PVOID pDrvContext,
   WD_KERNEL_PLUGIN_CALL *kpCall)
{
   kpCall->dwResult = MY_DRV_OK;

   switch ( kpCall->dwMessage )
   {
   // in this sample we implement a GetVersion message
   case MY_DRV_MSG_VERSION:
      {
         MY_DRV_VERSION *ver = (MY_DRV_VERSION *)
            kpCall->pData;
         ver->dwVer = 100;
         strcpy(ver->cVer,"My Driver V1.00");
         kpCall->dwResult = MY_DRV_OK;
      }
      break;
   // you can implement other messages here
   default:
      kpCall->dwResult = MY_DRV_NO_IMPL_MESSAGE;
   }
}
```

KP_IntEnable()

Called when WD_IntEnable() is called from the User Mode, with a Kernel PlugIn handler specified the pIntContext will be passed to the rest of the functions handling interrupts.

This function should contain any initialization needed for your Kernel PlugIn interrupt handling.

Prototype

```
BOOL __cdecl KP_IntEnable(PVOID pDrvContext, WD_KERNEL_PLUGIN_CALL *kpCall,
      PVOID *ppIntContext);
```

Parameters

```
pDrvContext - driver context data that was set by KP_Open().
kpCall - structure with information from WD_IntEnable().
kpCall.dwMessage - message ID passed from WD_IntEnable().
kpCall.pData - pointer to data passed from WD_IntEnable().
kpCall.dwResult - value to return to WD_IntEnable().
ppIntContext - pointer to interrupt context data that KP_IntDisable(), KP_IntAtIrql() and
      KP_IntAtDpc() functions will be called with. Use this to keep interrupt specific
      information.
```

Return Value

Returns TRUE if enable is succesful.

Example

```
BOOL __cdecl KP_IntEnable(PVOID pDrvContext,
   WD_KERNEL_PLUGIN_CALL *kpCall, PVOID *ppIntContext)
{
   // you can allocate memory specific for each interrupt
   // in *ppIntContext
   *ppIntContext = NULL;

   return TRUE;
}
```

KP_IntDisable()

Called when the User Mode application calls the WD_IntDisable() function. This function should free any memory which was allocated in the KP_IntEnable().

Prototype

```
void __cdecl KP_IntDisable(PVOID pIntContext);
```

Parameters

pIntContext - interrupt context data that was set by KP_Enable().

Return Value

none

Example

```
void __cdecl KP_IntDisable(PVOID pIntContext)
{
   // you can free the interrupt specific
   // memory in pIntContext here
}
```

KP_IntAtIrql()

This is the function which will run at IRQL if the Kernel PlugIn handle is passed when enabling interrupts.

Code running at IRQL will only be interrupted by higher priority interrupts.

Code running at IRQL is limited by the following restrictions:

1. You may only access non-pageable memory.

2. You may only call the following functions: WD_Transfer(), specific DDK functions which are allowed to be called from an IRQL.

3. You may not call malloc(), free(), or any WD_xxx command (other than WD_Transfer()).

The code performed at IRQL should be minimal (e.g. only the code which acknowledges the interrupt), since it is operating at a high priority. The rest of your code should be written at KP_AtDpc(), in which the above restrictions do not apply.

Prototype

```
BOOL __cdecl KP_IntAtIrql(PVOID pIntContext, BOOL *pfIsMyInterrupt);
```

Parameters

pIntContext - interrupt context data that was set by KP_IntEnable().
pfIsMyInterrupt - set this to TRUE, if the interrupt belongs to this driver, or FALSE if not. If you are not sure, it is safest to return FALSE.

Return Value

Returns TRUE if needs DPC function to execute.

Example

```
static DWORD G_dwInterruptCount = 0;
BOOL __cdecl KP_IntAtIrql(PVOID pIntContext,
   BOOL *pfIsMyInterrupt)
{
   // you should check your hardware here to see if
   // the interrupt belongs to you.
   // if in doubt, return FALSE (this is the safest)
   *pfIsMyInterrupt = TRUE;

   // in this example we will schedule a DPC once
   // in every 5 interrupts
   G_dwInterruptCount ++;
   if ((G_dwInterruptCount % 5) == 0 )
      return TRUE;

   return FALSE;
}
```

KP_IntAtDpc()

This is the Deferred Procedure Call which is executed only if the KP_IntAtIrql() function returned true.

Most of your interrupt handler should be written at DPC.

If KP_IntAtDpc() returns with a value of 1 or more, WD_IntWait() returns. I.e., if you do not want the User Mode interrupt handler to execute, the KP_IntAtDpc() function should return 0.

If KP_IntAtDpc() returns with a value which is larger than 1, this means that some interrupts have been 'lost' (i.e. were not processed by the User Mode). In this case, dwLost will contain the number of interrupts that were lost.

Prototype

```
DWORD __cdecl KP_IntAtDpc(PVOID pIntContext, DWORD dwCount);
```

Parameters

pIntContext - interrupt context data that was set by KP_Enable().
dwCount - the number of times KP_IntAtIrql() returned TRUE. If dwCount is 1, then only KP_IntAtIrql() only requested once a DPC. If the value is greater, then KP_IntAtIrql() has already requested a DPC a few times, but the interval was too short, therefore KP_IntAtDpc() was not called for each one of them.

Return Value

Returns the number of times to notify user-mode (i.e. return from WD_IntWait()).

Example

```
DWORD __cdecl KP_IntAtDpc(PVOID pIntContext, DWORD dwCount)
{
   // return WD_IntWait as many times as
   // KP_IntAtIrql scheduled KP_IntAtDpc()
   return dwCount;
}
```

Chapter 17

Kernel PlugIn structure reference

User Mode structures

WD_KERNEL_PLUGIN

Defines a Kernel PlugIn open command.

Used by WD_KernelPlugInOpen() and WD_KernelPlugInClose().

Members:

TYPE	NAME	DESCRIPTION
DWORD	hKernelPlugIn	handle to Kernel PlugIn
PCHAR	pcDriverName	Name of Kernel PlugIn driver. Should be no longer than 8 letters. Should not include the VXD or SYS extension.
PCHAR	pcDriverPath	The directory and file name in which to look for the KP driver. If NULL, then the driver will be searched for in the default windows system directory, under the name supplied in pcDriverName, with VXD added for Windows-95, or SYS added for Windows-NT.
PVOID	pOpenData	data to pass to KP_Open() callback in the Kernel PlugIn.

WD_INTERRUPT

Used to describe an interrupt.

Used by the following functions: WD_IntEnable(), WD_IntDisable(), WD_IntWait(), WD_IntCount().

Members:

TYPE	NAME	DESCRIPTION
WD_KERNEL_PLUGIN_CALL	kpCall	The kpCall structure contains the handle to the Kernel PlugIn, and other information which should be passed to the Kernel mode interrupt handler when installing it. If the handle is zero, then interrupt is installed without a Kernel PlugIn interrupt handler.
For information about all other members of WD_INTERRUPT, see the documentation of this structure in the 'WinDriver structure reference' chapter.		

WD_KERNEL_PLUGIN_CALL

Contains information about the Kernel PlugIn, which will be used when calling a utility Kernel PlugIn function or installing an interrupt.

Used by WD_KernelPlugInCall() and WD_IntEnable().

Members:

TYPE	NAME	DESCRIPTION
DWORD	hKernelPlugIn	handle to Kernel PlugIn.
DWORD	dwMessage	message ID to pass to Kernel PlugIn callback.
PVOID	pData	pointer to data to pass to Kernel PlugIn callback.
DWORD	dwResult	value set by Kernel PlugIn callback, to return back to User Mode.

Kernel Mode structures

KP_INIT

The KP_INIT structure is used by your KP_Init() function in the Kernel PlugIn. Its primary use is for notifying WinDriver what the name of the driver will be, and which Kernel Mode function to call when the application calls WD_KernelPlugInOpen().

Members:

TYPE	NAME	DESCRIPTION
DWORD	dwVerWD	Version of WinDriver library WD_KP.LIB.
CHAR	cDriverName[9]	The device driver name, upto 8 chars.
KP_FUNC_OPEN	funcOpen	The KP_Open() Kernel Mode function which WinDriver should call when the application calls WD_KernelPlugInOpen().

KP_OPEN_CALL

This is the structure through which the Kernel PlugIn defines the names of the call-backs which it implements. It is used in the KP_Open() Kernel PlugIn function.

A kernel PlugIn may implement 6 different call-back functions:

funcClose - Called when application is done with this instance of the driver.

FuncCall - Called when application calls the WD_KernelPlugInCall() function. This function is the 'general purpose' function. In it, implement any functions which should run in the Kernel Mode, (besides the Interrupt handler which is a special case). The funcCall will determine which function to execute according to the message passed to it.

funcIntEnable - Called when application calls the WD_KernelPlugInIntEnable(). This call-back function should initiate any activity which needs to be done when enabling an interrupt.

funcIntDisable - The cleanup function which is called when the application calls WD_KernelPlugInIntDisable().

funcIntAtIrql - This is the Kernel Mode interrupt handler. This call-back function is called when the WinDriver processes the interrupt which is assigned to this Kernel PlugIn. If this function returns a value greater than 0, then funcIntAtDpc is called as a Deferred procedure call.

funcIntAtDpc - Most of your interrupt handler code should be written in this call-back. It is called as a Deferred procedure call, if the funcIntAtIrql returns a value greater than 0.

Members:

TYPE	NAME	DESCRIPTION
KP_FUNC_CLOSE	funcClose	Name of your KP_Close() function in the kernel.
KP_FUNC_CALL	funcCall	Name of your KP_Call() function in the kernel.
KP_FUNC_INT_ENABLE	funcIntEnable	Name of your KP_IntEnable() function in the kernel.
KP_FUNC_INT_DISABLE	funcIntDisable	Name of your KP_IntDisable() function in the kernel.
KP_FUNC_INT_AT_IRQL	funcIntAtIrql	Name of your KP_IntAtIrql() function in the kernel.
KP_FUNC_INT_AT_DPC	FuncIntAtDpc	Name of your KP_IntAtDpc() function in the kernel.

Chapter
18

Developing in Visual Basic and Delphi

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

Using DriverWizard

DriverWizard can be used to diagnose your hardware and verify it is working properly before you start writing code. The DriverWizard's automatic source code generation generates code in C and Delphi only. Automatic generation of Visual Basic will be supported in later versions of WinDriver.

To create your driver code in C or in Delphi, refer to the 'DriverWizard' chapter.

Samples

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

1. \windriver\delphi\samples
2. \windriver\vb\samples

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

Kernel PlugIn

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

Creating your Driver

Development method in Visual Basic is the same as for developing under C except for the automatic code generation feature of DriverWizard.

Your work process should be as follows:

- Use DriverWizard to easily diagnose your hardware and verify it is working properly.
- Write your driver code in user mode using the WinDriver API. See details and explanations in the "Writing the device driver without the wizard" section in chapter 5.
- The files to be included when developing in VB are:
  - windrvr.cls
  - windrvr_usb.cls
- The files to be included when developing in Delphi are:
  - windrvr.pas
  - windrvr_usb.pas
- You may find it useful to use the WinDriver samples to get to know the WinDriver APIs and as a skeleton for your driver code.

Chapter
19

Trouble-shooting

To determine and verify the cause of your driver problems – Open the DebugMonitor (described in chapter 6) and set your desired trace level. This will help narrow down your debugging process and lead you in the right direction.

WD_Open() (or xxx_Open()) fails.

The following may cause WD_Open() to fail:

1. **Cause:** WinDriver's kernel is not loaded.

Action: Run WDREG.EXE install '(in the \windriver\util directory). This will let Windows know how to add WinDriver to the list of device drivers loaded on boot. Also, copy WINDRVR.SYS (for WinNT) or WINDRVR.VXD (for Win95) to the device drivers directory. A detailed explanation may be found in chapter 21 - 'Distributing your driver'.

2. **Cause:** The 30 day evaluation license is over.

Action: In this case, the WinDriver will inform you your evaluation license is over, in a message box. Please contact sales@krftech.com to purchase WinDriver.

3. **Cause (for PnP cards only):** The VendorID / DeviceID requested in xxx_Open() do not match that of the board. (In licensed versions).

Action: Run Your_card_name_DIAG.EXE, (generated by the DriverWizard or from the PLX /Galileo /V3 /AMCC directories), and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

4. **Cause:** The device is not installed or configured correctly.

Action: Run Your_Card_Name_DIAG.EXE and choose PCI scan. Check that your device returns all the resources needed.

5. **Cause:** Your device is in use by another application.

Action: Close all other applications that might be using your device.

WD_CardRegister() fails

WD_CardRegister fails if one of the resources defined in the card cannot be locked.

First, check out what resource (out of all the card's resources) cannot be locked:

Activate the KernelTracer and set the trace mode to trace.

This will output all warning and error debug messages. Now, run your application and you will get a printout of the resource that failed.

After finding out the resource that cannot be locked, check out the following:

Is the resource in use by another application? In order for several resource lock requests to the same IO, Memory or interrupt to succeed, both applications

must enable sharing of the resource. This is done by setting fNonSharable = FALSE for every item that can be shared.

Can't open USB device using the Wizard. Or WD_UsbDeviceRegister fails.

When a driver already exists in Windows for your device, you must create an .INF file (Driver Wizard automates this process) and install it. For exact instructions, see the sections explaining how to create and install .INF file in chapters 4 and 21.

Can't get interfaces for USB devices.

In some operating systems (such as Windows 98), when there is no driver installed for your USB device (Symptom - In the Wizard's "Card information" screen, the device's physical address is 0x0.) you must create an .inf file (Driver Wizard automates this process) and install it. For exact instructions, see the sections explaining how to create and install .INF file in chapters 4 and 21 .

PCI Card has no resources when using the Wizard

In some operating systems (such as Windows 98), when there is no device driver for a new device the operating system does not allocate resources to the device. The symptom – When trying to open the card in the Wizard's "Card information" screen, a message pop-up notifying no resources found on card. In addition, card configuration registers, such as memory ‘BAR’ are zeroed. When this happens, you need to create and install an INF file for the new card. For exact instructions, see the sections explaining how to create and install .INF file in chapters 4 and 21. .

Computer hangs on interrupt

This can occur with level-sensitive interrupt handlers. PCI cards interrupts are usually level sensitive.

Level sensitive interrupts are generated as long as the physical interrupt signal is high. If the interrupt signal is not lowered by the end of the interrupt handling by the kernel, The Windows OS will call the WinDriver kernel interrupt handler again - This will cause the PC to hang!

Acknowledging a level sensitive interrupt is hardware specific. Acknowledging an interrupt means lowering the interrupt level generated by the card. Normally, writing to a register on the PCI card can terminate the interrupt, and lower the interrupt level.

When calling WD_IntEnable() it is possible to give the WinDriver kernel interrupt handler a list of transfer commands (IO and memory read/write commands) to perform upon interrupt, at the kernel level - before WD_IntWait() returns.

These commands can be used to write to the needed register to lower the interrupt level, thereby re-setting the interrupt.

Before calling WD_IntEnable(), prepare two transfer command structures (to read the interrupt status and then write the status to lower the level).

```
WD_TRANSFER trans[1];
BZERO (trans);
trans[0].cmdTrans = WP_DWORD; // Write Port Dword
// address of IO port to write to
trans[0].dwPort = dwAddr;
// the data to write to the IO port
trans[0].Data.Dword = 0;
Intrp.dwCmds = 1;
Intrp.Cmd = trans;
Intrp.dwOptions = INTERRUPT_LEVEL_SENSITIVE;
WD_IntEnable(hWD, &Intrp);
```

This will tell WinDriver's kernel to Write to the register at dwAddr a value of '0', upon an interrupt.

The user-mode interrupt handler (The thread waiting on WD_IntWait() - this is your code).

Here you only do your normal stuff to handle the interrupt. You do not need to clear the interrupt level since this was already done by the kernel of WinDriver, with the transfer command you gave WD_IntEnable().

WD_DMALock() fails to allocate buffer

The efficient method for memory transfer is scatter/gather DMA. If your hardware does not support scatter/gather, you will need to allocate a DMA buffer using WD_DMALock().

WD_DMALock() fails when the Windows OS has run out of contiguous physical memory.

When calling WD_DMALock() with dwOptions = DMA_KERNEL_BUFFER_ALLOC, WinDriver requests the Windows OS for a physical contiguous memory block.

On WinNT you can allocate a few hundred kilobytes by default. If you want to allocate a few megabytes, you will have to reserve memory for it, by setting the following value in the registry:

On Windows NT:

- run REGEDIT.EXE, and access the following key:
  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Co
  ntrol\
  Session Manager\Memory Management
- Increment the value of NonPagesPoolSize.
- This change will take place only after re-boot.

On Windows 95:

Win95 does not support contiguous buffer reservation, therefore, the earlier you allocate the buffer, the larger the block you can allocate.

Chapter
20

Dynamically loading your driver

Windows NT/2000 and 9x

Dynamic loading - background

When adding a new driver to the Windows operating system, you must re-boot the system, for the Windows to load your new driver into the system.

Dynamic loading enables you to install a new driver to your operating system, without needing to re-boot.

WinDriver is a dynamically loadable driver, and provides you with the utility needed to dynamically load the driver you create.

You may dynamically load your driver whether you have created a User Mode or a Kernel Mode driver.

Why do you need a dynamically loadable driver?

A dynamically loadable driver enables your customers to start your application immediately after installing it, without the need to re-boot.

Dynamically loading and unloading your driver

The utility you use to dynamically load and unload your driver is called WDREG.EXE, and may be found in the \windriver\util\WDREG.EXE.

USAGE: WDREG [-name] [-file] [[CREATE] [START] [STOP] [DELETE] [INSTALL] [REMOVE]]

WDREG.EXE has 4 basic operations:

1. CREATE - Instructs the Windows to load your driver next time it boots, by adding your driver to registry.

2. START - Dynamically loads your driver into memory for use. On Windows NT/2000, you must first 'CREATE 'your driver before 'START 'ing it.

3. STOP - Dynamically unloads your driver from memory.

4. DELETE - Removes your driver from the registry , so that it does not load on next boot.

For example, To reload WinDriver use:

WDREG STOP START

WDREG.EXE has 2 'shortcut 'operations for your convenience:

1. INSTALL - Creates and starts your driver (same as using WDREG CREATE START).

2. REMOVE - Unloads your driver from memory, and removes it from registry so that it does not load on next boot (same as using WDREG STOP DELETE).

You may dynamically load your driver via command line or from within your application as follows:

1. Dynamically loading your driver via command line:

    - From the command line, type WDREG INSTALL. This loads the driver into memory, and instructs Windows to load your driver on the next boot.

2. Dynamically loading your driver in your installation application:

    - Add the WDREG source code to your installation application.

    - The full source code for WDREG may be found at \windriver\samples\wdreg\

Dynamically loading your Kernel PlugIn

If you have used WinDriver to develop a Kernel PlugIn, you must dynamically load your Kernel PlugIn as well as the WinDriver.

To Dynamically load / unload your Kernel PlugIn driver ([Your driver name].VXD / [Your driver name].SYS):

Use the WDREG command as described above, with the addition of the "-name" flag, after which you must add the name of your Kernel PlugIn driver.

For example, to load your Kernel PlugIn driver called KPTest.VXD or KPTest.SYS, use:

WDREG -name KPTest install

(You should not add the .VXD or .SYS extension to your driver name).

WDREG allows you to install your driver in the registry under a different name than the physical file name.

USAGE: WDREG -name [Your new driver name] -file [Your original driver name] install

For example, typing the following:

WDREG -name "Kernel PlugIn Test" -file KPTest install

Installs the KPTest.VXD or KPTest.SYS driver under a different name.

Linux

To dynamically load WinDriver on Linux, execute:

```
> /sbin/insmod –f /lib/modules/misc/windrvr.o
```

To dynamically unload WinDriver, execute:

```
> /sbin/rmmod windrvr
```

Chapter

21

Distributing your driver

Read this chapter in the final stages of your development. This chapter guides you in creating the distributable package from your driver alongside of the WinDriver.

Get a valid license for your WinDriver

To purchase your WinDriver license, fill in your order form (\windriver\docs\order.txt), and fax or email it to KRFTech (you may find the full details on the order form itself).

Alternatively, you may order WinDriver on-line. See http://www.krftech.com for more details.

Windows 9x and NT/2000

Copy VxD or SYS files to target computer

In the driver installation script you create, you must copy the following files to the target computer (the one you will install your driver on):

- For Windows NT: Copy WINDRVR.SYS file to C:\WINNT\SYSTEM32\DRIVERS
- For Windows 95: Copy WINDRVR.VXD file to C:\WIN95\SYSTEM\VMM32

USB developers:

- Copy WINDRVR.SYS and WDUSB.SYS files to C:\WINNT\SYSTEM32\DRIVERS (Windows 2000) C:\WIN98\SYSTEM32\DRIVERS (Windows 98)

If you have created a WinDriver Kernel PlugIn as well, copy your Kernel PlugIn driver ([Your driver name].VXD or [Your driver name].SYS) to the relevant directory..

Add WinDriver to the list of Device Drivers Windows loads on boot

This is done by calling 'WDREG.EXE install'. You can add the WDREG source code (found in \windriver\samples\wdreg\wdreg.cpp) to your own installation code, in order to install WinDriver.

If you have created a WinDriver Kernel PlugIn as well, call "WDREG.EXE -name [Your driver name] install". You can add the 'WDREG 'source code (found in \windriver\samples\wdreg\wdreg.cpp) to your own installation code, in order to install WinDriver.

Please see the chapter on 'Dynamically loading your driver 'for more details on WDREG.EXE

Creating an .INF file

Device information (INF) files are text files, that provide information used by the "Plug and Play" mechanism in Windows 95/98/2000 to install software that supports a given hardware device. INF files are required for hardware that identifies itself, such as USB and PCI. The INF file includes all necessary information about the device(s) and the files to be installed. When hardware manufactures introduce new products, they must create INF files to explicitly define the resources and files required for each class of device.

In some cases, the .INF file of your specific device is included in the .INF files supplied with the operating system. In other cases, you will need to create an .INF file for your device. The Driver wizard can generate an INF specific for your Card/device. The INF is used to tell the OS that the selected device is now handled by WinDriver.

Why should I create an INF file?

1. To stop the 'new hardware wizard' of the Windows operating system from poping up after boot.

2. In some cases the OS doesn't initilaze the PCI configuration registers in Win98 without an INF file.

3. In some cases the OS doesn't assign phyiscal address to USB devices without an INF file.

4. To load the new driver created for the card\device. Creating an INF file is required whenever developing a new driver for the hardware.

5. To replace the existing driver with a new one.

How do I install an INF file when no driver exists?

When no driver exists for your hardware, use the DriverWizard to generate an INF file for your card\device (the INF file includes your device VID/PID and loads WDUSB.SYS as your device driver).

- Save the file under C:\temp\mydevice.INF (or any other name or location you choose). For instructions on how to generate the INF file see the 'DriverWizard' chapter.

- Go to: Start | Settings | Control Panel | System

- Use the operating system 'Add new hardware' wizard to add and register the .INF file created with WinDriver. In the relevant screen enter the path of the new .INF file created with WinDriver.

How do I replace an existing driver using the INF file?

Windows NT 2000:

- Use the DriverWizard to generate an INF file for your card\device (the INF file includes your device VID/PID and loads WDUSB.SYS as your device driver). Save the file under C:\temp\mydevice.INF (or any other name or location you choose). For instructions on how to generate the INF file, see the 'DriverWizard' chapter.

- Go to: Start | Settings | Control Panel | System

- Click Button: Device Manager in the "Hardware" tab

- Menu: View | Devices by connection

- For PCI cards: In Tree: Standard PC | PCI bus | <your_card>
  For USB devices: In Tree: Standard PC | PCI bus | PCI to USB Universal Host Controller | USB Root Hub | <the current driver for the device>

- Menu: Action | Properties

- Click Button: Update Driver in the "Driver" tab.
- In DriverWizard: click button: Next
- Radio selection: Search for a suitable driver for my device
- Click "Next" Button
- Check box selection: only Specify a location
- Click "Next" Button
- Enter the path: c:\temp\<your .INF file>
- Click "OK" Button
- Click "Next" Button
- Click "Finish" Button
- Reboot

Windows 98:

- Use the DriverWizard to generate an INF file for your card\device (the INF file includes your device VID/PID and loads WDUSB.SYS as your device driver). Save the file under C:\temp\mydevice.INF (or any other name or location you choose). For instructions on how to generate the INF file see the 'The DriverWizard' chapter.
- Go to: Start | Settings | Control Panel | System
- Click radio: View devices by connection in the "Device Manager" tab.

- For PCI cards: In Tree: Standard PC | PCI bus | <your_card>
  For USB devices: In Tree: Computer | Plug and Play BIOS | PCI bus | PCI to USB Universal Host Controller | USB Root Hub | <the current driver for the device>

- Button: Properties

- Click Button: Update Driver in the "Driver" tab

- Click "Next" Button

- Check box selection: only Specify a location

- Enter the path: c:\temp\<your .INF file>

- Click Button: OK

- Click "Next" Button

- Click "Finish" Button

- Reboot

For Windows CE

Copy WinDriver Kernel DLL file to target computer

In the driver installation script you create, you must copy the following files to the target computer (the one you will install your driver on):

For Windows CE handheld computer installations:

- Copy WINDRVR.DLL file to \WINDOWS on your target Windows CE computer

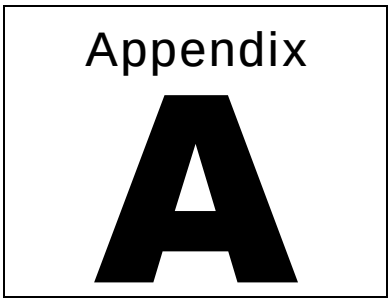
For Windows CE PC:

- Copy WINDRVR.DLL %_FLATRELEASEDIR% and use MAKEIMG.EXE to build a new Windows CE kernel NK.BIN. You should modify PLATFORM.REG and PLATFORM.BIB appropriately before doing this by appending the contents of the supplied files PROJECT_WD.REG and PROJECT_WD.BIB respectively. This process is similar to the process of installing WinDriver CE Beta on a CE PC / ETK installation as described in Chapter 3 – INSTALLATION AND SETUP.

Add WinDriver to the list of Device Drivers Windows CE loads on boot

For Windows CE handheld computer installations, please modify the registry according to the entries documented in the file PROJECT_WD.REG. This can be done using the Windows CE Pocket Registry Editor on the handheld CE computer or by using the Remote CE Registry Editor Tool supplied with the Windows CE Platform SDK. You will need to have Windows CE Services installed on your Windows NT Host System to use the Remote CE Registry Editor Tool.

For Windows CE PC/ETK, the required registry entries are made by appending the contents of the file PROJECT_WD.REG to the Windows CE ETK configuration file PROJECT.REG before building the Windows CE image using MAKEIMG.EXE. If you wish to make the WinDriver kernel file a permanent part of the Windows CE kernel NK.BIN, you should append the contents of the file PROJECT_WD.BIB to the Windows CE ETK configuration file PROJECT.BIB as well.



Appendix

PC-Based Development Platform Parallel Port Cable Specification (For Windows CE)

To use the parallel port shell utility (Ppsh) to transfer a Windows CE image from your development workstation to a PC-based hardware development platform, a custom parallel cable is required. This cable requires a DB-25 male connector at both ends, with pins mapped as follows:

1	10
2	Same
3	Same
4	Same
5	Same
6	Same

7	Same
8	Same
9	Same
10	1
11	14
12	16
13	17
14	11
15	Not Connected On Either End
16	12
17	13
18	Same
19	Same
20	Same
21	Same
22	Same
23	Same
24	Same
25	Same

To order this cable, contact Redmond Cable:

Redmond Cable

15331 NE 90th Street

Redmond, WA 98052

Telephone: (425) 882-2009

Fax: (425) 883-1430

Part Number: 64355913

Limitations on demo versions

Windows 9x and NT/2000

- A DEMO MESSAGE will appear at every first use of WinDriver in each session.
- WinDriver will function for only 30 days after the original installation.

Windows CE

- A DEMO MESSAGE will appear at every first use of WinDriver in each session.
- The WinDriver CE Kernel (windrvr.dll) will operate for no more then 10 minutes at a time.
- WinDriver CE emulation on Windows NT will stop working after 30 days.

Linux

- The Linux Kernel will work for no more then 10 minutes at a time.

Version history list

NEW IN V2.02

- Header files can now be compiled under Borland C/C++ compiler.
- Anonymous unions were changed in structures WD_TRANSFER and WD_CARD.

NEW IN V2.10

- For memory mapped cards, changed item dwUserAddr to dwTransAddr.
- Use dwTransAddr when calling WD_Transfer(). added dwUserDirectAddr for direct memory transfers without calling WD_Transfer(). dwUserDirectAddr NOT YET IMPLEMENTED.

NEW IN V2.11

- For PCI cards: Structure used for calls to WD_PciScanCards() was changed.
  Use pciScan.searchId.dwVendorId and pciScan.searchId.VendorId and the same for dwDeviceId.

NEW IN V2.12

- For memory mapped cards: you can now directly access the memory region, without calling WD_Transfer(). the pointer to the memory region is returned in dwUserDirectAddr returned by WD_CardRegister().

- DMA transfers: DMA contiguous buffer allocation by WinDriver is available by setting dwOptions = DMA_KERNEL_BUFFER_ALLOC, when calling WD_DMALock(). the linear address of the buffer allocated will be returned in pUserAddr, and the physical address in Page[0]. the buffer is available until calling WD_DMAUnlock().

NEW IN V3.0

- Added DriverWizard to package. DriverWizardenables the programmer to 'talk' and 'listen' to his card via a windows user-interface. TheDriverWizardthen creates the source code for the driver.

- DMA option DMA_LARGE_BUFFER added for locking regions larger then 1MB.

- Removed limitation of 20 concurrent DMA buffers in use.

NEW IN V3.01

- Support for Win98 and Windows 2000

NEW IN V3.02

- Minor improvements in DriverWizard

- Supports Windows NT checked build

NEW IN V3.03

- Enhanced support for Multi-CPU Multi-PCI bus
- Corrected the interrupt count value returned by WD_IntWait.

NEW IN V4.0

- WinDriver Kernel PlugIn - allows running parts of the driver code from the Kernel Mode.
- Sleep function - For accessing slow hardware.
- ISA Plug and Play support.
- Debugging monitor - Allows tracking of errors, warnings and trace messages from the WinDriver's kernel module.
- Dynamic driver loader - WinDriver enables the driver created to be loaded and unloaded without rebooting the machine.
- Enhanced source code generation for interrupts - DriverWizard creates full interrupt source code.
- PLX 9050 library enhancements - EEPROM read/write support functions and Enhanced interrupt handling.

NEW IN VERSION 4.1

- New support for Linux, Windows CE and Alpha NT.
- Support for ISA PnP cards.
- Support for PCMCIA cards in Windows CE.

- Graphical KernelTracer introduced.
- Robust support for Delphi and VB (Visual Basic). More Delphi and VB samples.
- New support for the PLX 9054 and 9080 chipsets. Support includes EEPROM access and bus master DMA implementation.
- Support for Galileo GT64 chipsets.
- V4.1 Includes The Enhanced WinDriver Wizard:
    - Automatic Vendor and Device detection.
    - Automatic handling and code generation for Level sensitive interrupts.
    - Wizard allows multiple concurrent register and memory dialogs.
    - Improved GUI.

Purchasing WinDriver

Choose the WinDriver product that suits your needs:

- Choose WinDriver 'for NT/2000 or 9x support.
- Choose WinDriver Bundle 'for Windows NT/2000 and 9x support (no re-writing or re-compiling needed).
- Choose WinDriver CE 'for Windows CE support.
- Choose WinDriver Linux 'for Linux support.
- Choose WinDriver Alpha NT 'to run your driver on the Alpha NT platform.
- Choose WinDriver ToolBox 'to receive all the above operating systems support in one package. The driver you develop will run under all supported environments.

Fill in the order form found in Start/WinDriver/Order Form 'on your Windows start menu, and send it back to KRFTech via email/fax/mail (see details below).

Your WinDriver package will be sent to you via Fedex / Postal mail. The WinDriver license string will be emailed to you immediately.

E-MAIL

Support: support@krftech.com

Sales: sales@krftech.com

Services: services@krftech.com

PHONE / FAX

Phone:

USA (Toll-Free): 1-877-514-0537

Worldwide: +972-9-8859365

Fax::

USA (Toll-Free): 1-877-514-0538

Worldwide: +972-9-8859366

WEB

http://www.krftech.com/

ADDRESS

KRFTech

7 Giborei Isarel St.

P.O.B. 8493

Netanya 42504

ISRAEL

Distributing your driver - legal issues

WinDriver is licensed per-seat. The WinDriver license allows for one developer to develop an unlimited number of device drivers, and to freely distribute the created driver without royalties.

You may not distribute the windrvr.h file, or any source file that describes the WinDriver's functions. Please see the \windriver\docs\license.txt file for the full WinDriver license agreement.