WinDriver V5 User's Guide

COPYRIGHT

Copyright © **1997-2001 Jungo Ltd. All Rights Reserved**

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

Windows, Win32, Windows 95, Windows 98, Windows ME, Windows CE, Windows NT and Windows 2000 are trademarks of Microsoft Corp. WinDriver and KernelDriver are trademarks of Jungo. Other brand and product names are trademarks or registered trademarks of their respective holders.

WinDriver Overview

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

Introduction to WinDriver
Background
WinDriver Advantages
WinDriver Feature List
WinDriver Architecture
What Platforms does WinDriver Support?
Can I Try WinDriver Before I Buy?
Limitations of the different evaluation versions
How do I develop my Driver with WinDriver? (Overview)
What Does the WinDriver Toolkit Include?
Can I distribute the driver created with WinDriver?
Device Driver Overview
Matching the right tool for your driver

WinDriver USB Overview

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

Introduction to USB
Feature List
USB Components
Data Flow in USB Devices
USB Data Exchange
USB Data Transfer Types
USB Configuration
WinDriver USB
WinDriver USB Architecture
What drivers can I write with WinDriver USB?

Installation and Setup

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

Systems Requirements
Installing WinDriver
Upgrading Your Installation
Checking Your Installation
Uninstalling WinDriver

The DriverWizard

An Overview
DriverWizard Walkthrough
DriverWizard Notes
Remote WinDriver

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 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
Writing the Device Driver without the DriverWizard
Win CE - Testing on CE
Using the Help Files

Debugging

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

User Mode Debugging
DebugMonitor

WinDriver Function Reference

The WinDriver API is available from the user mode and the Kernel Plugin to WinDriver users, and from the kernel mode for KernelDriver users.

Use this Chapter as a quick reference to the WinDriver functions. The definition of the structures used in the following functions may be found in the 'WinDriver Structure Reference' [WinDriver Structure Reference].

NOTE: If you are a registered user, you need to read the file register.txt under *windriver/redist/register* or *kerneldriver/redist/register* to understand the process of enabling your driver to work with the registered version.

WD_Open()
WD_Close()
WD_Version()
WD_PciScanCards()
WD_PciGetCardInfo()
WD_PciConfigDump()
WD_PcmciaScanCards()
WD_PcmciaGetCardInfo()
WD_PcmciaConfigDump()
WD_IsapnpScanCards()
WD_IsapnpGetCardInfo()
WD_IsapnpConfigDump()
WD_CardRegister()
WD_CardUnregister()
WD_Transfer()
WD_MultiTransfer()
WD_IntEnable()
WD_IntDisable()
WD_IntWait()
WD_IntCount()
WD_DMALock()
WD_DMAUnlock()
WD_Sleep()
WD_UsbScanDevice()
WD_UsbGetConfiguration()
WD_UsbDeviceRegister()
WD_UsbDeviceUnregister()
WD_UsbTransfer()
WD_UsbResetPipe()
InterruptThreadEnable()
InterruptThreadDisable()

WinDriver Structure Reference

The WinDriver API is available from the user mode and the Kernel Plugin to WinDriver users, and from the kernel mode for KernelDriver users. Use this Chapter as a reference to the structures used by the WinDriver API.

WD_TRANSFER
WD_DMA
WD_DMA_PAGE
WD_INTERRUPT
WD_VERSION
WD_CARD_REGISTER
WD_CARD
WD_ITEMS
WD_SLEEP
WD_PCI_SLOT
WD_PCI_ID
WD_PCI_SCAN_CARDS
WD_PCI_CARD_INFO
WD_PCI_CONFIG_DUMP
WD_ISAPNP_CARD_ID
WD_ISAPNP_CARD
WD_ISAPNP_SCAN_CARDS
WD_ISAPNP_CARD_INFO
WD_ISAPNP_CONFIG_DUMP
WD_PCMCIA_SLOT
WD_PCMCIA_ID
WD_PCMCIA_SCAN_CARDS
WD_PCMCIA_CARD_INFO
WD_ PCMCIA_CONFIG_DUMP
WD_USB_ID
WD_USB_PIPE_INFO
WD_USB_CONFIG_DESC
WD_USB_INTERFACE_DESC
WD_USB_ENDPOINT_DESC
WD_USB_INTERFACE
WD_USB_CONFIGURATION
WD_USB_HUB_GENERAL_INFO
WD_USB_DEVICE_GENERAL_INFO
WD_USB_DEVICE_INFO
WD_USB_SCAN_DEVICES
WD_USB_TRANSFER
WD_USB_DEVICE_REGISTER
WD_USB_RESET_PIPE

Creating Your Driver Using Jungo's PLX9050 API

This chapter takes you through the driver development process for the PLX9050 PCI bridge -- Read this chapter only if you are using the PLX9050 PCI bridge.

Overview of the Development Process for the PLX9050 Using WinDriver's PLX9050 API
What is the PLX9050 Diagnostics?
Using PLX9050 Diagnostics
Creating the driver project without using the P50_DIAG as the skeletal driver code
WinDriver's PLX9050 API Function Reference
PLX 9050 API Structure Reference
Trouble-Shooting

Creating Your Device Driver Using Jungo's PLX9060 / PLX9080 API

This chapter takes you through the driver development process for the PLX9060 or PLX9080 PCI bridge - Read this chapter only if you are using the PLX9060 or PLX9080 PCI bridge.

Overview of the Development Process for the PLX9060 / PLX9080 Using WinDriver's PLX9060 API
What is PLX9060 Diagnostics?
Using PLX9060 Diagnostics
Creating the Driver Project without using the P60_DIAG as the Skeletal Driver Code
WinDriver's PLX 9060/9080 API Function Reference
PLX 9060 API Structure Reference
Trouble-Shooting

Creating Your Device Driver Using Jungo's AMCC S5933 API

This chapter takes you through the driver development process for the AMCC S5933 PCI bridge - Read this chapter only if you are using the AMCC S5933 PCI bridge.

Overview of the Development Process for the AMCC S5933 Using WinDriver's AMCC API
What is AMCC Diagnostics?
Using the AMCC Diagnostics
Creating the Driver Project without using the AMCCDIAG as the Skeletal Driver Code
Sample Code
WinDriver's AMCC S5933 API Function Reference
AMCC S5933 API Structure Reference
Trouble-Shooting

Creating Your Driver Using Jungo's V3 PBC API

This chapter takes you through the driver development process for the V3 PBC PCI bridge -- Read this chapter only if you are using the V3 PBC PCI bridge.

Overview of the Development Process for the V3 PBC using WinDriver's V3 PBC API
What is V3 PBC Diagnostics?
Using the V3 PBC Diagnostics
Creating the Driver Project without using the PBC_DIAG as the Skeletal Driver Code
Sample Code
WinDriver's V3 PBC API Function Reference
PCI Configuration Registers Definitions
Trouble-Shooting

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, Altera, 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
Handling Interrupts
USB Control Transfers
Performing Control Transfers with WinDriver

Improving Performance

Improving performance of your device driver - Overview
Performance improvement checklist
Improving the performance of your User Mode driver

Kernel PlugIn

This chapter provides you with a brief description of the "Kernel PlugIn" feature of WinDriver.

Background
Do I need to write a Kernel PlugIn?
What kind of Performance can I expect
Overview of the development process

WinDriver Kernel PlugIn Architecture

This chapter explains the architectural details of the "Kernel PlugIn" feature of WinDriver.

Introduction
WinDriver Kernel and Kernel Plugin Interaction
Kernel Plugin Components
Kernel PlugIn Event Sequence

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
Directory structure and sample for the WinDriver Kernel PlugIn
Kernel PlugIn implementation
KPTest -- A sample Kernel PlugIn Driver
Handling Interrupts in the Kernel PlugIn
Message passing

Writing a Kernel PlugIn

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 skeletal 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
Preparing the User Mode source code
Creating a new Kernel PlugIn Project
Creating a handle to the WinDriver Kernel PlugIn
Interrupt Handling in the Kernel PlugIn
I/O handling in the Kernel PlugIn
Compiling your Kernel PlugIn Driver
Installing your Kernel PlugIn Driver

Kernel PlugIn Function reference

User Mode functions
Kernel functions

Kernel PlugIn structure reference

User Mode structures
Kernel Mode structures

Developing in Visual Basic and Delphi

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

Using DriverWizard

DriverWizard can be used to diagnose your hardware and verify that it is working properly before you start coding. DriverWizard's automatic source code generator generates code in C and Delphi only. Automatic generation of code in Visual Basic will be supported in later versions of WinDriver.

To create your driver code in C and Delphi, please refer to the "DriverWizard" The DriverWizard chapter.

Samples

Samples for drivers writtten 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 User Mode, must use C when writing their Kernel PlugIn.

Creating your Driver

The method of development in Visual Basic is the same as the method in 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 that it is working properly.

- Write your driver code in the User Mode using the WinDriver API. For details and explanations please see Section Writing the Device Driver without the DriverWizard that explains how to write the device driver without the DriverWizard.

- The files to be included when developing in VB are:

1. windrvr.cls

2. windrvr_usb.cls

- The files to be included when developing in Delphi are:

1. windrvr.pas

2. windrvr_usb.pas

- You may find it useful to use the WinDriver samples to get to know the WinDriver API and as your skeletal driver code.

Trouble-Shooting

To determine and verify the cause of your driver problems Open the DebugMonitor 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.
WD_CardRegister() fails
Can't open USB device using the DriverWizard
Can't get interfaces for USB devices.
PCI Card has no resources when using the DriverWizard
Computer hangs on interrupt

Dynamically loading your driver

Windows NT/2000 and 95/98/ME
Linux
Solaris

Distributing your Driver

Read this chapter in the final stages of driver development. This chapter guides you in creating your driver package for distribution.

Get a valid license for your WinDriver
Windows 95/98/ME and NT/2000
Creating a .INF file
Windows CE
Linux
Solaris
VxWorks

PC-Based Development Platform Parallel Port Cable Info

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 ends
  - 16 - 12
  - 17 - 13
  - 18 - Same
- 19 - Same
  - 20 - Same
  - 21 - Same
  - 22 - Same
  - 23 - Same
  - 24 - Same

- 25 - Same

To order this cable, please contact:
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 95/98/ME 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 than 30 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 30 minutes at a time.

Solaris

- The Solaris Kernel will work for no more then 30 minutes at a time.

VxWorks

- The VxWorks Kernel will work for no more then 30 minutes at a time.

DRIVERWIZARD GUI

- An evaluation dialog pops up whenever you attempt to interact with the hardware.

Version history list

New in Version 2.02

- Header files can now be compiled under Borland C/C++ compiler.
    - Anonymous unions were changed in structures WD_TRANSFER and WD_CARD.

New in Version 2.10

- 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 Version 2.11

- 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 Version 2.12

- 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 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 till WD_DMAUnlock() is called.

New in Version 3.0

- Added DriverWizard to the package. DriverWizard enables the programmer to 'talk' and 'listen' to his card via a windows user-interface. DriverWizard then 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 Version 3.01

- Support for Win98/ME and Windows 2000

New in Version 3.02

- Minor improvements in DriverWizard
    - Supports Windows NT checked build

New in Version 3.03

- Enhanced support for Multi-CPU Multi-PCI bus
    - Corrected the interrupt count value returned by WD_IntWait().

New in Version 4.0

- 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.
    - Debug 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, and Windows CE.
    - 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.

- Includes The Enhanced DriverWizard.
  - Automatic Vendor and Device detection.
  - Automatic handling and code generation for Level sensitive interrupts.
  - Wizard allows multiple concurrent register and memory dialogs.
  - Improved GUI.

New in Version 4.14

- ISA PnP support
  - Wizard can generate Kernel Code, supported by KernelDriver.
  - Wizard generates Borland CBuilder Ver 3 and Ver 4 make files.
  - PLX 9052 support.
  - PLX 9054 and 9080 added DMA function enabling non-busy wait for DMA to complete.
  - Added WD_VB_GetAddress() for VB to get an address of a variable.
  - Overcome Windows NT inability to map addresses at the end of physical memory (???-0xffffffff)

- Fixed 9054 and 9080 EEPROM access and corrected register names.
  - WD_VERSION structure correction in Delphi.
  - Fixed interrupt handling in Windows CE.
  - Fixed WD_PciGetCardInfo() on Windows 98 / ME.
  - Fixed Wizard Halting after interrupt is disabled.

New in Version 4.20

- Wizard now also generates driver code in Delphi.
  - Enhanced support for Altera PCI cores.
  - Automatic generation of INF file for windows 95/98/ME.
  - Wizard generates makefiles simultaneously for all operating systems and IDEs chosen by the user.
  - All samples include makefiles for all supported IDEs.
  - Debug Monitor is now integrated into the Wizard environment.

- GUI enhancements in the Driver Wizard.

- The default when adding new ISA interrupts in Wizard is now sharable interrupts.
  - Contiguous DMA also for Linux 2.2 (only 2.0 was supported previously).
  - Fixed Scatter / Gather DMA in Windows 98/ME.
  - Unix make files: Fixed DOS slashes.

New in Version 4.30

- USB support in WinDriver.
  - Wizard enables programmers to detect their USB devices and select the desired configuration.
  - Wizard enables USB hardware testing (read, write and listen to pipes).
  - Wizard generates USB driver source code in C\C++ or Delphi.
  - Wizard generates .INF file for USB devices for Windows 98/ME/2000 (as well as for PCI devices).
  - WinDriver's Kernel PlugIn supports all WinDriver USB APIs.

- WinDriver's Kernel PlugIn supports also Solaris and Linux, assures optimal performance for all supported operating systems and full code compatibility among all supported operating systems.
  - In version 4.3 WinDriver and KernelDriver are integrated into one driver development suite.
  - KernelDriver now also supports Windows 95/98/ME (VxD. Drivers).
  - KernelDriver now also supports Linux.
  - USB support in KernelDriver API is available now also in C in KernelDriver for Windows NT/2000 (previously only in C++).

- GUI enhancements: Registers can now be defined as auto-read.
  - Wizard includes pre-defined resources for parallel port which enable quick access to the port.
  - Fixed: WD_DMAUnlock() with KERNEL_BUFFER_ALLOC on Windows NT and Linux.
  - WD_IntWait(): Can now terminate applications waiting on IntWait without locking machine.

New in version 4.31

- USB support: Improved performance of bulk transfer and isochronous transfer.
- Major improvements in handling interrupts in Visual Basic.
- Interrupt handling sample in Visual Basic is now available.
- New graphical sample in Visual Basic for accessing the parallel port.
- VxD files generated with KernelDriver or with 'WinDriver Kernel PlugIn' now communicate with windrvr.sys on Windows 98/ME.
- Fix: interrupt handling on Windows 98/ME, WindowsNT/2000 (bug only in V4.30).
- Fix: WD_DMAUnlock() on Windows 98/ME, WindowsNT/2000 (bug only in V4.30).
- Fix: error message when installing USB INF file on Windows 2000.

New in version 4.32

- Name change: The company name has been changed from KRFTech to Jungo.
- New feature: supports USB devices with multiple interfaces.
- Major improvement in WinDriver for Solaris: The PCI scan is now performed faster, and we eliminated abnormally long scan times on AXi boards.
- Improvement in WinDriver USB: Isoch transfer auto adjust are better capable of overcoming failures in transfers.
- Improvement in WinDriver USB: Supports multiple interfaces.
- Fix: On Windows 95/98/ME, non-sharable edge-triggered interrupts were in some cases received only once.
- Fix: On WinNT/98/ME (SYS) and Solaris, resources were not released on abnormal termination.
- Fix: WinDriver USB would display a message "duplicate object name" after installation.
- Fix: In WinDriver USB, A Device can now be stopped.
- Fix: In WinDriver USB, No more report error in short USB transfers and transfer of size 0.
- Fix: In WinDriver for Linux, Installation fixes on Linux 2.0.kernel.
- Fix: In Wizard, Saving INF file without an extension can now be done without causing problems.
- Fix: In WinDriver for Solaris, Interrupts are now acknowledged by default which prevents sys log overflow and lock ups.

- Fix: In KernelDriver, WinNT samples and the generated code can now be compiled currently using the build utility.

New in Version 4.33

- New feature: On WinNT enabled access to additional PCI devices that are not accessible by Windows.
- New feature: On Windows2000 added support to service pack 1.
- New feature: On Windows2000 added the capability to generate Windows 2000 INF file for PCI devices. INF file and PnP service is required for PCI cards on Windows2000.
- New feature: In WinDriver for Solaris(Sparc), added support for Ultra 220, Ultra 450, CP1500 and CP1400.
- New feature: In WinDriver for Solaris, enabled abnormal application termination.
- Fix: In WinDriver and KernelDriver, fixed the crash that occurred when WD_DMAUnlock() was called from Kernel PlugIn and KernelDriver.
- Fix: In WinDriver USB, WD_USB_MAX_DEVICE_NUMBER changed from 20 to 127. This requires programs to be recompiled.
- Fix: On Windows eliminated crashes that resulted from the Delphi code for PCI interrupts that was generated by the wizard.
- Fix: On Windows2000 corrected computation of interrupt slot number.
- Fixed Interrupt related bugs that caused system hang.
- Fix: On Windows2000 fixed crashes that occurred when USB user write buffer is declared as const void * in user mode code.
- Fix: In WinDriver for Linux, fixed Kernel PlugIn for registered version.
- Fix: In WinDriver for Linux, fixed compilation warning about undeclared function int close(int).
- Fix: In WinDriver for Linux, removed extra license check which broke the registered version of KernelPlugIn.
- Fix: In WinDriver for Solaris, fixed memory leak in abnormal application termination.
- Fix: In WinDriver for Solaris, fixed crash on WD_KernelPlugInOpen().
- Fix: In WinDriver for Solaris(Sparc), shortened scan of PCI bus from 40 seconds to 1-2 seconds.
- Fix: In WinDriver for Solaris(Sparc), fixed PCI interrupt handling for shared PCI interrupts and cards behind PCI bridges.

New in Version 4.34

- Fix: Eliminated application crashes when listening repeatedly (listen/stop listening/listen) to ISA interrupts on Windows 95/98. For example, through the Wizard.
- Fix: Corrected shared interrupt acknowledgment when WinDriver user mode program is forcibly terminated.
- Enhancement: Windows 98, Kernel PlugIn user mode sample program - Added a new comment to usermode.c advising users not to use long file names.
- Fix: Corrected Windows 95/98 and NT kernel plugin libraries that gave linkage errors on 4.33.
- Fix: The file windrvr_ce_emu.lib was missing from the 4.33 package.
- Fix : Fixed crashes on Solaris 8 that occured when calling WD_CardRegister()

New in Version 5.0

- New feature: Added a Graphical User Interface (GUI) to WinDriver for Linux and WinDriver for Solaris.
- New feature: A Remote Access feature is incorporated into the WinDriver Wizard. Remote Host capability is currently supported on Windows NT/2000, Linux and Solaris. Remote Target capability is supported on Windows 95/98/ME/NT/2000/CE, VxWorks, Solaris and Linux.
- New feature: The evaluation timeout per session on Linux/Solaris/VxWorks/WinCE is increased to 30 minutes.
- New feature: The electronic documentation (PDF) now has thumbnails, bookmarks, and cross-reference hyperlinks in the references sections. HLP (WinHelp) files have cross reference hyperlinks. CHM format documentation is included with Windows versions; HTML format documentation is included with Unix versions.
- Fix: For DriverWizard, several fixes have been incorporated in the file saving process and in reading from the registry.
- Fix: The INF files generated by DriverWizard under Windows 2000 now comply with MS recommendations for Windows 2000.

- Enhancement: DriverWizard Wizard for Windows allows choosing between WinDriver and KernelDriver help files when invoked.

- Fix: KernelDriver USB generates C and C++ SYS files for Windows 2000.

Purchasing WinDriver

Choose the WinDriver product that suits your needs:

- Choose '**WinDriver**' for NT/2000 or 95/98 support.
- Choose '**WinDriver Bundle**' for Windows NT/2000 and 95/98 support (no re-writing or re-compiling needed).
- Choose '**WinDriver CE**' for Windows CE support.
- Choose '**WinDriver Linux**' for Linux support.
- Choose '**WinDriver ToolBox**' to receive all the above operating systems support in one package. The driver you develop will run under all supported environments.

in the order form found in '***Start | WinDriver | Order Form***' on your Windows start menu, and send it back to Jungo 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 - M A I L

Support: support@jungo.com

Sales: sales@jungo.com

Services: services@jungo.com

P H O N E / F A X

Phone:

USA (Toll-Free): 1-877-514-0537

Worldwide: +972-9-8859365

Fax:

USA (Toll-Free): 1-877-514-0538

Worldwide: +972-9-8859366

W E B:

Jungo `http://www.jungo.com`

POSTAL A D D R E S S

Jungo Ltd,

P.O.Box 8493,

Netanya 42504,

ISRAEL

Distributing your driver - legal issues

WinDriver is licensed per-seat. The WinDriver license allows one developer on a single computer to develop an unlimited number of device drivers, and to freely distribute the created driver without royalties, as outlined in the license agreement below.

SOFTWARE LICENSE AGREEMENT OF **WinDriver V5.0**

Jungo © 1999-2001

JUNGO ("LICENSOR") IS WILLING TO LICENSE THE ACCOMPANYING SOFTWARE TO YOU ONLY IF YOU ACCEPT ALL OF THE TERMS IN THIS LICENSE AGREEMENT. PLEASE READ THE TERMS CAREFULLY BEFORE YOU INSTALL THE SOFTWARE, BECAUSE BY INSTALLING THE SOFTWARE YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THESE TERMS, LICENSOR WILL NOT LICENSE THIS SOFTWARE TO YOU, AND IN THAT CASE YOU SHOULD IMMEDIATELY DELETE ALL COPIES OF THIS SOFTWARE YOU HAVE IN ANY FORM.

OWNERSHIP OF THE SOFTWARE

1. The enclosed Licensor software program ("Software") and the accompanying written materials are owned by Licensor or its suppliers and are protected by United States of America copyright laws, by laws of other nations, and by international treaties.

   GRANT OF LICENSE

2. Jungo grants to you as an individual, a personal, non-exclusive "one-user" license to use the Software on a single computer in the manner provided below at the site for which the license was given. If you are an entity, Jungo grants you the right to designate one individual within your organization to have the right to use the Software on a single computer in the manner provided below at the site for which the license was given.

3. If you have not yet purchased a license to the Software, Licensor grants to you the right to use the Software for an evaluation period of 30 days. If you wish to continue using the Software and accompanying written materials after the evaluation period, you must register the Software by sending the required payment to Licensor. You will then receive a license for continued use and a registration code that will permit you to use the Software on a single computer free of payment reminders. The Software may come with extra programs and features that are available for use only to registered users through the use of their registration code.

   RESTRICTIONS ON USE AND TRANSFER

4. You may not distribute any of the headers or source files which are included in the Software package.

5. The license for WinDriver allows you for royalty free distribution of the following files only when complying with **5a, 5b, 5c** and **5d** of this agreement: WINDRVR.SYS (Windows NT), WINDRVR.VXD (Windows 95/98/ME), WINDRVR.DLL (Windows CE),

WDPNP.SYS(98/ME/2000),windrvr.o (Linux) - as generated from 'make install',   windrvr and windrvr.cnf (Solaris), and windrvr.o (Vxworks).

5a. These files may be distributed only as part of the application you are distributing, and only if they significantly contribute to the functionality of your application.

5b. You may not distribute the WinDriver header file (WINDRVR.H). You may not distribute any header file which describes the WinDriver functions, or functions which call the WinDriver functions and have the same basic functionality as the WinDriver functions themselves.

5c. You may not modify the distributed files specified in section 5 of this agreement.

5d. WinDriver may not be used to develop a development product, an API, or any products which will eventually be part of a development product or environment, without the written consent of the licensor.

10. You may make printed copies of the written materials accompanying Software provided that they used only by users bound by this license.

11. You may not distribute or transfer your registration code or transfer the rights given by the registration code.

12. You may not rent or lease the Software or otherwise transfer or assign the right to use the Software.

13. You may not reverse engineer, decompile, or disassemble the Software.

DISCLAIMER OF WARRANTY

14. THIS SOFTWARE AND ITS ACCOMPANYING WRITTEN MATERIALS ARE PROVIDED BY LICENSOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, ARE DISCLAIMED.

15. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, SAVINGS, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

16. This Agreement is governed by the laws of the United States of America.

17. If you have any questions concerning this Agreement or wish to contact the Licensor for any reason, please write to:

Jungo © 1999-2001

Address:

Jungo Ltd,

P.O.Box 8493

Netanya 42504

ISRAEL.

Web site:

`http://www.jungo.com`

E-mail:

`info@jungo.com`

Voice:

1-877-514-0537(USA)

+972-9-8859365(Worldwide)

Fax:

1-877-514-0538(USA)

+972-9-8859366(Worldwide)

U.S. GOVERNMENT RESTRICTED RIGHTS

18. The Software and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraph (c)(1) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or sub-paragraphs (c)(1)(ii) and (2) of Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable.

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 is source code compatible between all supported operating systems (WinDriver currently supports Windows 95/98/ME/NT/2000/CE, Linux, Solaris, VxWorks and OS/2.). It is binary compatible between Windows 95, 98, ME, NT and 2000. Bus architecture support includes PCI/PCMCIA/ ISA/ISA PnP/EISA/CompactPCI 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 major part 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 is 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 into the Kernel Mode. Therefore, the driver achieves optimal Kernel Mode performance, with User Mode ease of use.

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

Good luck with your project!

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/ME/NT / CE / Linux / Solaris...)

2. Learn how to write a device driver.

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

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 the first four steps for each new operating system on which the code should run.

WinDriver Advantages

Easy development - WinDriver enables Windows programmers to create **PCI/PCMCIA/ISA/ISA PnP/EISA/CompactPCI and 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, Borland C++, Visual Basic, GCC or any other 32-bit compiler. WinDriver eliminates the need for you to be familiar with the operating system internals, kernel programming or with the DDK,ETK,DDI/DKI or have any device driver knowledge.

Cross Platform - The driver created with WinDriver will run on **Windows 95/ 98/ME/NT/2000/CE, Linux, Solaris, VxWorks and OS/2**, - i.e. write once - run on any of these platforms.

Friendly Wizards - 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, DriverWizard creates the skeletal driver source code, giving access functions to all the resources on the hardware.

Kernel Mode Performance **-** WinDriver's API is optimized for performance. For drivers that need kernel mode performance, WinDriver offers the "Kernel PlugIn". This powerful feature enables you to create and debug your code in 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 with Windows CE or VxWorks, there is no need to use the Kernel PlugIn since Windows CE and VxWorks have no separation between user mode and kernel mode. This enables you to 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 about 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 DriverWizard and detect the hardware and its resources.

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

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,on VxWorks 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.
    - DriverWizard automatically generates the driver code for the project in C\C++ or Delphi (Pascal).
    - Supports any PCI/PCMCIA/ISA/ISA PnP/EISA/CompactPCI and USB device regardless of manufacturer.
    - Enhanced support for the PLX 9030/9050/9052/9054/9060/9080/IOP 480, Altera, Galileo, QuickLogic, PLDA, V3 and AMCC PCI bridges, thereby hiding the PCI bridge details from the developer.
- Applications are binary compatible across Windows 95,98,ME,NT and 2000.
    - Applications are source code compatible across Windows 95, 98, ME, NT, 2000, CE, Linux, Solaris, VxWorks and OS/2.
    - WinDriver can be used with common development environments including MSDEV, Visual C/C++, Borland Delphi, Borland C++, Visual Basic, GCC or any other 32 compiler.
    - 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.
    - Four months of free technical support.
    - No run time fees or royalties.

Notes

1. PCMCIA is only supported in the Windows CE version.

WinDriver Architecture

WinDriver Architecture

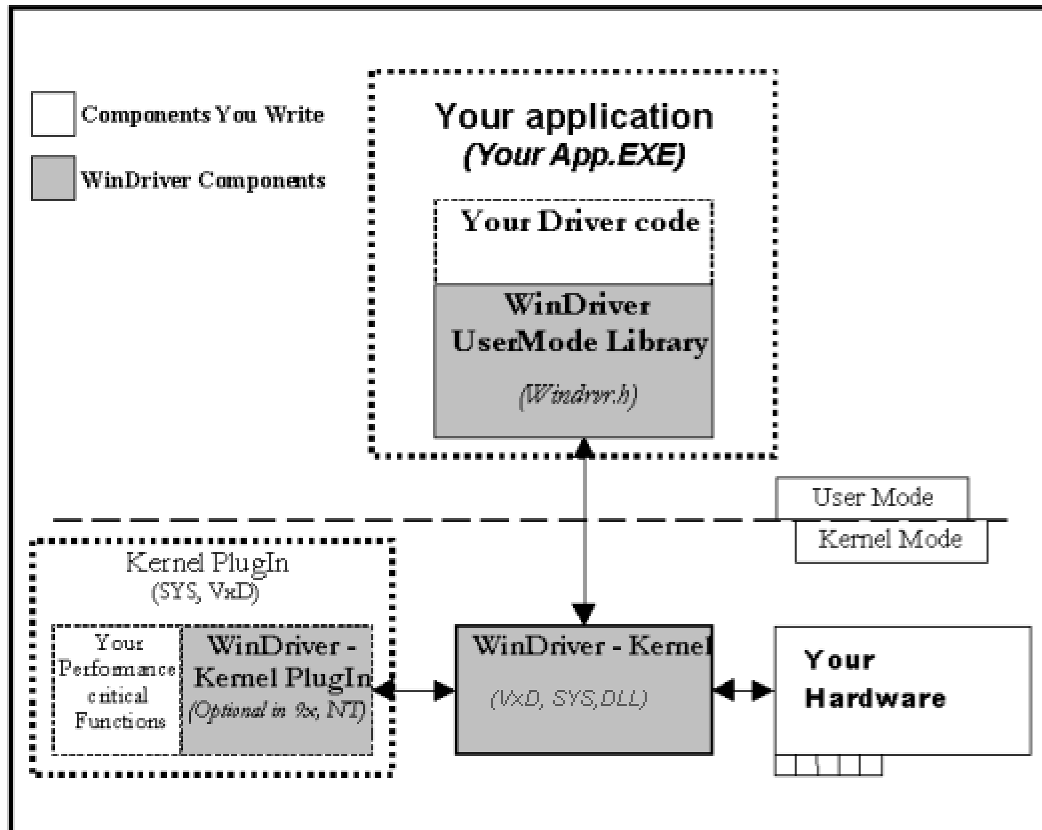


Figure 1.1: WinDriver Architecture

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 minimizes performance hits on your code, even though it is running in the User Mode. However, some hardware drivers need performance, which cannot be achieved 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) into the WinDriver Kernel PlugIn without changing a single line of it. Now, WinDriver Kernel calls 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 and VxWorks** there is no separation between User Mode and Kernel Mode, therefore 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/ME/NT/2000/CE, Solaris, VxWorks, OS/2 and Linux. The same source code will run on all supported platforms. The same executable you write will operate on Windows 95, 98, ME, 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 Jungo web site at `http://www.jungo.com/dnload.html`

Limitations of the different evaluation versions

All the evaluation versions of WinDriver are full featured. No functions are limited or crippled in any way. The following is a list of the differences between the evaluation versions and the registered ones:

- At first use of the driver, an 'unregistered' message appears.
- A dialog box with a message stating that an evaluation version is being run , is popped up on every interaction with the hardware.
- In the Linux, Solaris, VxWorks and CE versions - The driver is operational for 30 minutes after which it has to be restarted.
- The Windows evaluation version expires 30 days from the date of installation.

How do I develop my Driver with WinDriver? (Overview)

On Windows 95, 98, ME, NT and 2000
On Windows CE
On Linux and Solaris
On Embedded Operating Systems

What Does the WinDriver Toolkit Include?

- The WinDriver CD.
- A printed version of this manual.
- Four months of free technical support (Phone - Fax - Email).
- The WinDriver CE license enables you to run your CE driver code on your NT machine via the CE emulation.
- The WinDriver Linux and Solaris licenses enable 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

Utilities

WinDriver's SPECIFIC CHIP-SET SUPPORT

Samples

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

Windows 95/98/ME Drivers

NT Driver Model

Unix Device Drivers

Linux Device Drivers

Solaris Device Drivers

Matching the right tool for your driver

Jungo offers two driver development products lines: WinDriver and KernelDriver. WinDriver is a tool designed for monolithic type user mode drivers. WinDriver enables you to access your hardware directly from within your Win32 application, without writing a kernel mode device driver. Using WinDriver you can either access your hardware directly from your application (in user mode) or write a DLL that you can call from many different applications.

WinDriver also provides a complete solution for high performance drivers. Using WinDriver's ***Kernel PlugIn***, you can 'drop' your user mode code into the kernel and reach full kernel mode performance.

A driver created with WinDriver runs on **Windows 95, 98, ME, NT, 2000, CE, Linux, Solaris, VxWorks 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).

There are situations that require drivers to be running in the kernel mode. Network drivers under Linux and Windows for example, almost always need to reside in the kernel. In addition under Windows NT, for layered or miniport drivers, kernel programming is necessary. To simplify this difficult task, Jungo provides "KernelDriver" - a tool kit for writing kernel mode drivers for Windows platforms (95/98/ME/NT/2000) and Linux. In addition, KernelDriver has special support for NT/2000 - a C++ toolkit that 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.

Introduction to USB

USB, short for *Universal Serial Bus*, 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 USB, was fueled because of several considerations. Among them are the needs for an inexpensive and widespread connectivity solution for peripherals in general and for the "Computer Telephony Integration" in particular, the need for an 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 needs stated above. 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 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 generation (USB2.0) supports a faster signalling rate of 480 Mb/S that is 40 times faster than USB 1.1. USB2.0 is fully forward and backward compatible with USB1.1 and uses the existing cables and connectors.

USB2.0 supports a connection for higher bandwidth, higher functionality PC peripherals. In addition, it has the capability to handle more simultaneously running peripherals.

USB2.0 will benefit many applications like Interactive Gaming, Broadband Internet Access, Desktop and Web Publishing, Internet Services and Conferencing.

Feature List

- External connection; easy to use for the 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.
- Optimized 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 is where the USB host controller is installed, and where the client software / device driver runs. The USB host controller is the interface between the host and the USB peripherals. The host is responsible for detecting 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 devices provide 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, the 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 associations between an endpoint on the USB device and software on the host. The data is moved to and from the device 'through' a pipe. A 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:

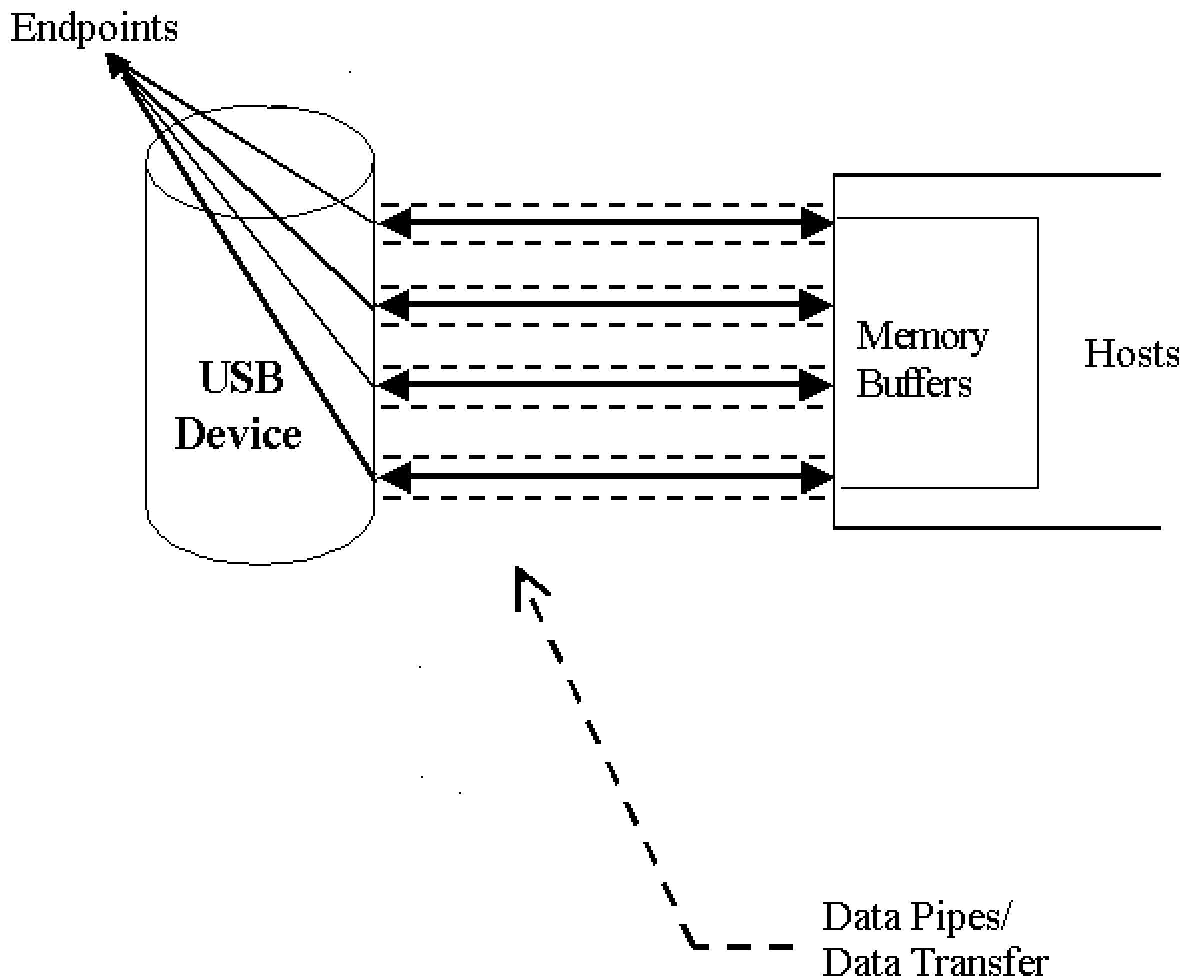


Figure 2.1: USB Endpoints

USB Data Exchange

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

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

- Control exchange is used to configure a device when it is first attached and can also be used for other device-specific purposes, including control of other pipes on the device. The control exchange is transferred via the control pipe (Pipe 00). The control transfer consists of a setup stage (in which a setup packet is sent from the host to the device), an optional data stage and a status stage.

More information on how to implement the control transfer by sending Setup Packets can be found in Chapter WinDriver Implementation Issues that deals with WinDriver Implementation Issues.

The screen shot below shows a USB device with one bi-directional control and three functional data transfer pipes/endpoints:

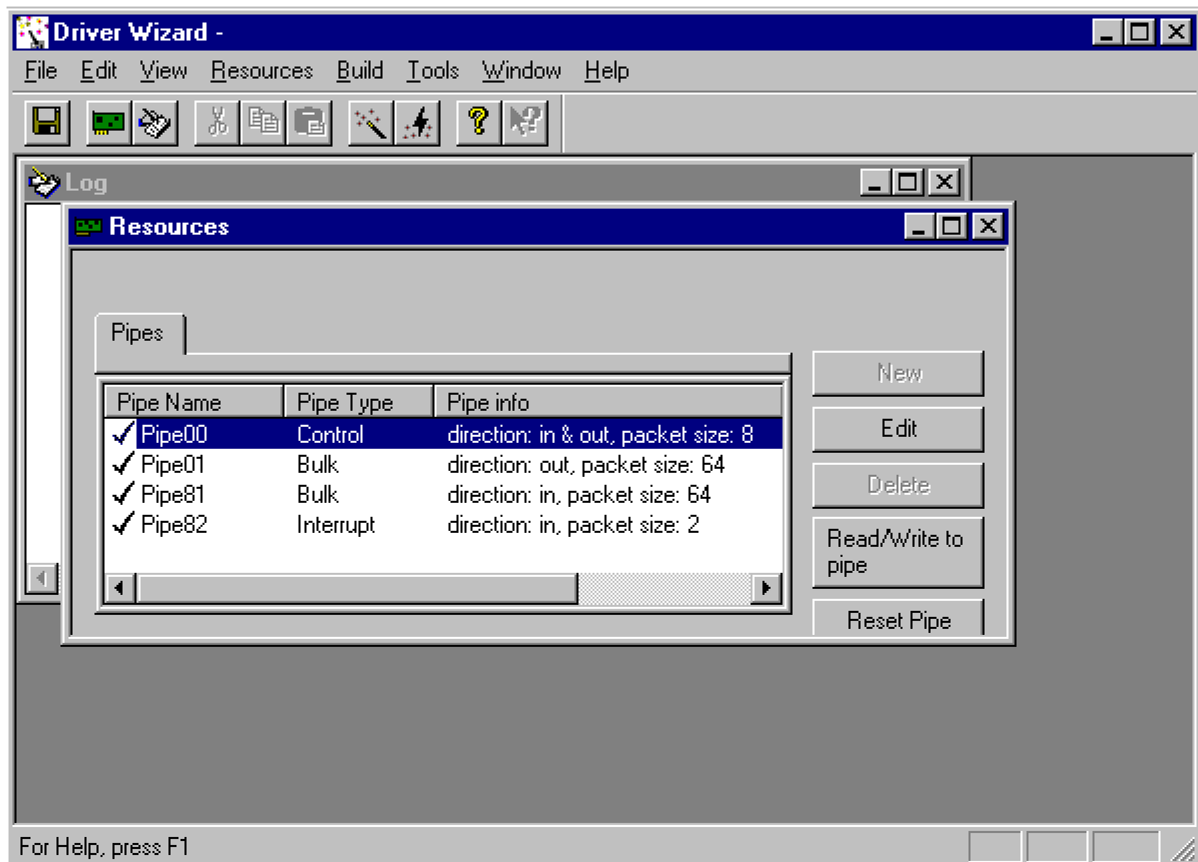


Figure 2.2: USB Pipes

USB Data Transfer Types

The USB device (function) communicates with the host by transferring data through a pipe between a memory buffer on the host and an endpoint on the device. USB 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: 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 provides access to the configuration, status and control information. The control pipe is a bi-directional pipe. Control transfer is 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 bi-directional isochronous communication there's a need to use two isochronous pipes, one in each direction. USB guarantees the isochronous transfer access to the USB bandwidth (that is it reserves the required amount of bytes of the USB frame) with bounded latency and guarantees the data transfer rate through the pipe unless there is less data transmitted. 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 the error that 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 amounts 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 non-periodic, large packet, bursty communication. Bulk transfer typically supports devices that transfer large amounts of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners. Bulk transfer allows access to the 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, the system will use it for bulk transfer. Like previous stream pipes (isochronous and interrupt) the bulk pipe is also uni-directional. Bulk transfer can only be used 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. A descriptor is the defined structure and format in which the data is transferred. A complete description of the USB descriptors can be found in Chapter 9 of the USB Specification (See `http://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 Level: At the top level is the 'device descriptor', that 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 Level: 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 as self-powered, remote wakeup, maximum power consumption and more). At a given time, only one configuration is loaded. An example of different configurations of the 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 Level: The interface is a related set of endpoints that present a specific functionality or feature of the device. Each interface may operate independently. The interface descriptor describes the number of the interface, number of endpoints used by this interface, and the interface specific class, subclass and protocol values when the interface operates independently. In addition, an interface may have alternate settings. The alternate settings allow the endpoints or their characteristics to be varied after the device is configured.

Endpoint Level: The lowest level is the endpoint descriptor that provides the host with information regarding the data transfer type of the 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 attributes of the endpoints 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 diagnostics application, scan the USB bus, detect all USB devices and their

different configurations, interfaces, settings and endpoints, and enables the developer to pick the desired configuration before starting driver development.

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

WinDriver USB

WinDriver USB enables developers to quickly develop high performance drivers for USB based devices, without having to learn the USB specifications 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, Windows ME and Windows 98.

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, please check Jungo's web site at
`http://www.jungo.com`

WinDriver USB encapsulates the USB specification and architecture, letting you focus on your application logic. WinDriver USB features DriverWizard, with which you can detect your hardware, configure it and test it before writing a single line of code. 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 ensure that 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 types of configurations.

After your hardware is diagnosed, DriverWizard automatically generates 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. The 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 diagnostics application, which just needs 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 everything you need to develop your driver in VB.

DriverWizard also automates the creation of a .INF file where needed. The .INF file is a text file used by the Plug-&-Play mechanisms of Windows 95/98/ME 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 identify themselves, 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 a .INF file for your device. WinDriver automates this process for you. More information on how to create your own .INF file with DriverWizard can be found in Chapter The DriverWizard that explains the DriverWizard. Installation instructions of .INF files can be found in Chapter Distributing your Driver that illustrates how to distribute your driver.

Using WinDriver USB, all development is done in the user mode, using familiar development and debugging tools and your favorite compiler (such as MSDEV, Visual C/C++, Borland Delphi, Borland C++, Visual Basic).

WinDriver USB API is designed to give you optimized performance. In 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 Chapter WinDriver USB Overview that covers the USB features of WinDriver.

WinDriver USB Architecture

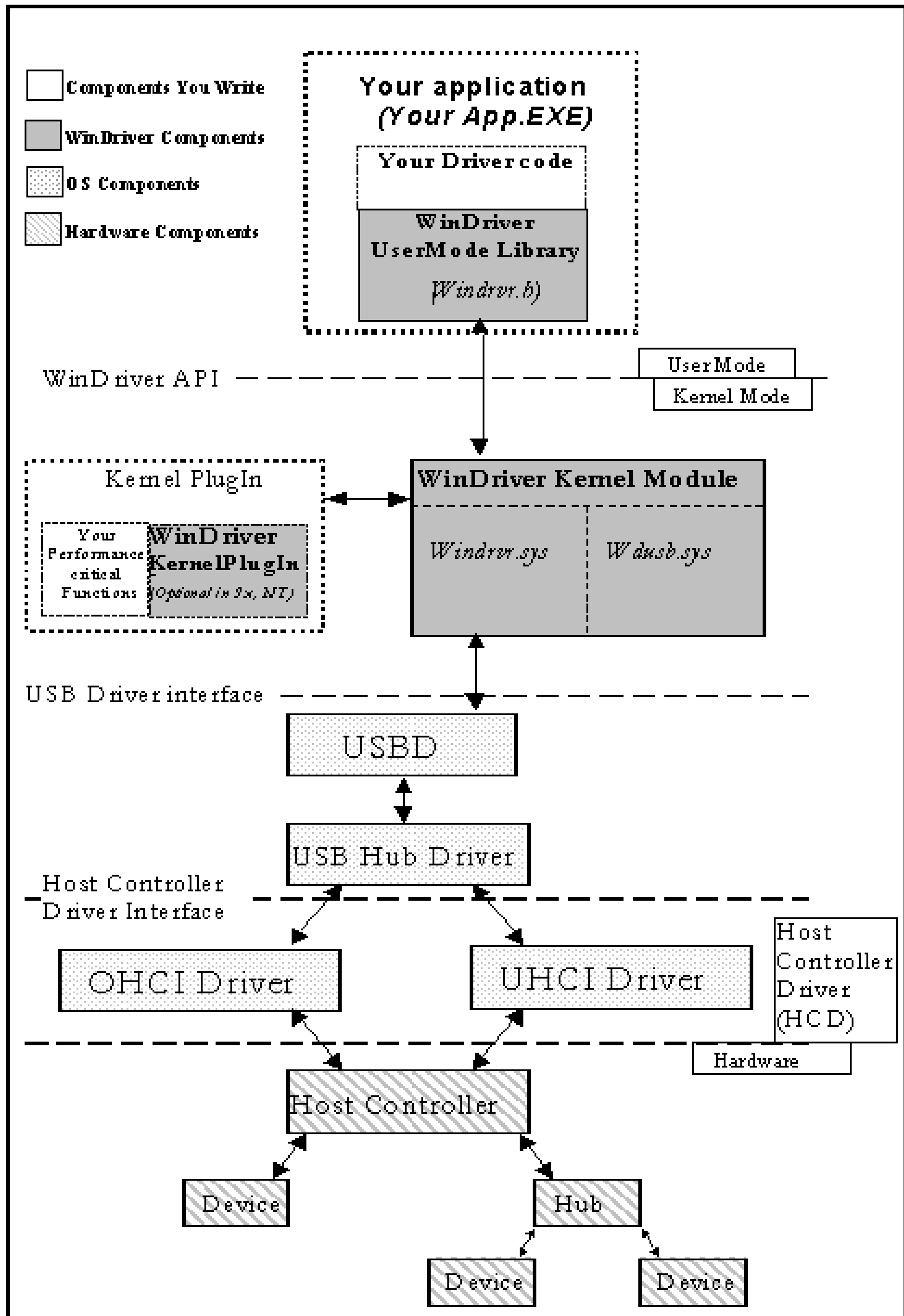


Figure 2.3: WinDriver USB Architecture

NOTE: Any occurrence of wdusb in the above figure should be read as wdpnp. As of version 5.0 the file wdusb.sys has been changed to wdpnp.sys and supports PCI, PCMCIA and USB devices.

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 wdpnp.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 recognize and configure the device. The software implementing the configuration can include the hub driver, the device driver and other software.

WinDriver USB abstracts the configuration procedure and hardware access described above for the developer. With WinDriver USB API, developers can do all the hardware-related operations 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 need 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 Jungo).

For quicker development time, select WinDriver USB over KernelDriver USB wherever possible.

Systems Requirements

For Windows 95 / 98 / ME
For Windows NT/2000
For Windows CE
For Linux
For Solaris
For VxWorks

Installing WinDriver

The WinDriver CD contains all versions of WinDriver for all the different operating systems. The CD's root directory contains the Windows 95/98/ME and NT/2000 version. This will automatically begin when you insert the CD into your CD drive. The other versions of WinDriver are located in subdirectories i.e. **\Linux, \Wince** and so on.

Installing WinDriver for Windows 95,98,ME,NT and 2000

Installing WinDriver CE

Testing Your Applications on the X86 HPC Emulator

Installing WinDriver for Linux

Installing DriverWizard on your Windows machine

Installing WinDriver for Solaris

Installing DriverBuilder for VxWorks

Upgrading Your Installation

To upgrade to a new version of WinDriver, follow the steps outlined in Section Installing WinDriver for Windows 95,98,ME,NT and 2000 that illustrates the process of installing WinDriver for Windows 95/98/ME/NT/2000. You can either choose to overwrite the existing installation or install to a separate directory.

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

To upgrade your source code, navigate to the RegisterWinDriver() function call in your source code, and pass the new license string as a parameter to this function. For more information on RegisterWinDriver(), please refer to the file **register.txt** in the directory **WinDriver\redist\register**.

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

Checking Your Installation

On your Windows machine
On your Windows CE machine
On your Linux machine
On your Solaris machine
On VxWorks

Uninstalling WinDriver

If for some reason you wish to uninstall either the evaluation or registered version of WinDriver, please refer to this section.

Uninstalling Windriver from Windows (95/98/ME/NT/2000)

Uninstalling Windriver from Linux

Uninstalling WinDriver from Solaris

Uninstalling DriverBuilder for Vxworks

An Overview

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 (where 'status register' is a register you have defined on your hardware).

If you are developing a driver for a PLX based card, it is recommended to read the chapter that explains WinDriver's enhanced support for specific PCI chipsets, before starting your driver development(**Note**: this chapter does not appear in the HLP format documentation). You can use DriverWizard to diagnose your hardware. You should use DriverWizard to generate an INF file for your card for Windows operating systems. You should avoid using DriverWizard to generate code for your PLX card because DriverWizard generates generic code and you will have to modify the code before it can be useful. We supply complete source code libraries and sample applications tailored for various PLX chipsets in the package.

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/PCMCIA/ISA/ISA PnP/EISA/CompactPCI bus or attach your new USB device to the USB port in your machine, and use DriverWizard to check that the hardware is performing as expected.

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

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

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

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

- *A project workspace* that you can use to automatically load all the above project information and files into your development environment. In WinDriver Linux and WinDriver Solaris, DriverWizard generates the makefile for the relevant operating system.

DriverWizard Walkthrough

Following are the five steps in using DriverWizard:

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

2. Run DriverWizard.

- Click *Start* | *WinDriver* | *DriverWizard* from the start menu or double click the DriverWizard icon on your desktop.

- The start-up dialog will appear. Click your mouse to start 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).

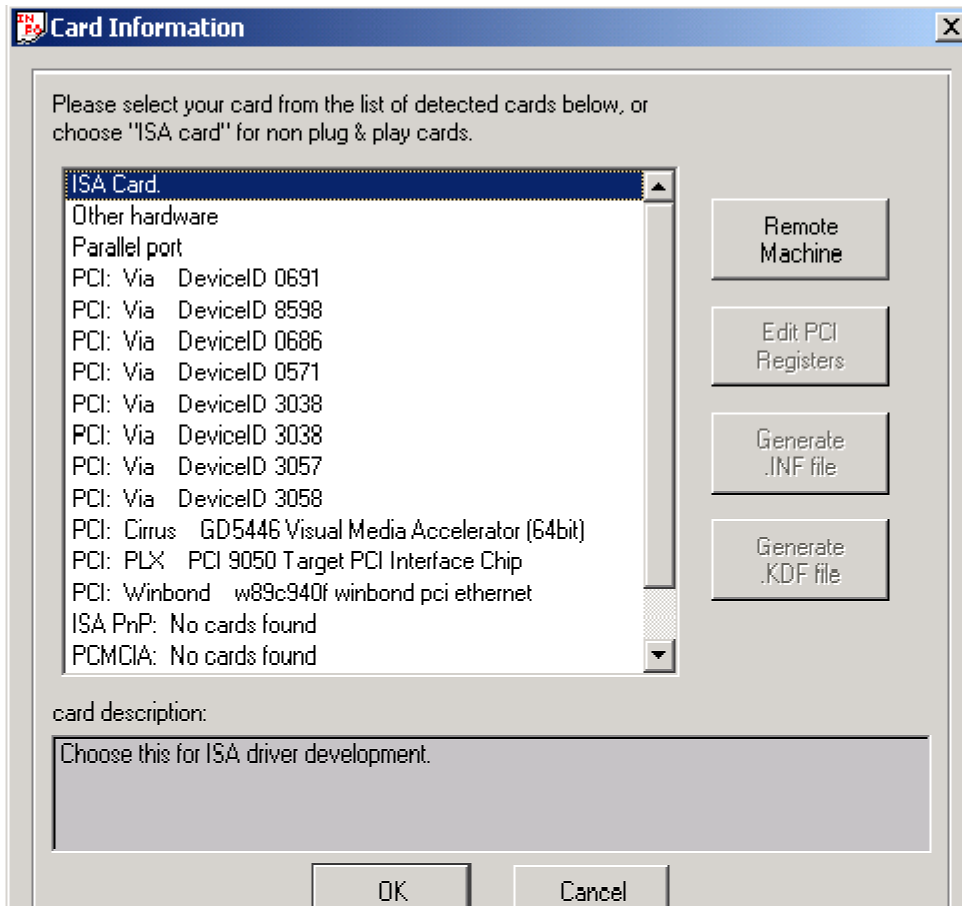


Figure 4.1: Selection of PnP Device

- In some cases, 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 then 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 DriverWizard is 'my_device.inf').

To install the .INF file follow the instruction displayed by DriverWizard or refer to Section Creating a .INF file that explains how to create an INF file.

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/ME and Windows 2000 without an INF file. In such cases, you will not be able to diagnose your hardware with DriverWizard until after 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 DriverWizard until after 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 PCI/PCMCIA/ISA/ISA PnP/EISA/CompactPCI cards should skip this step):

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

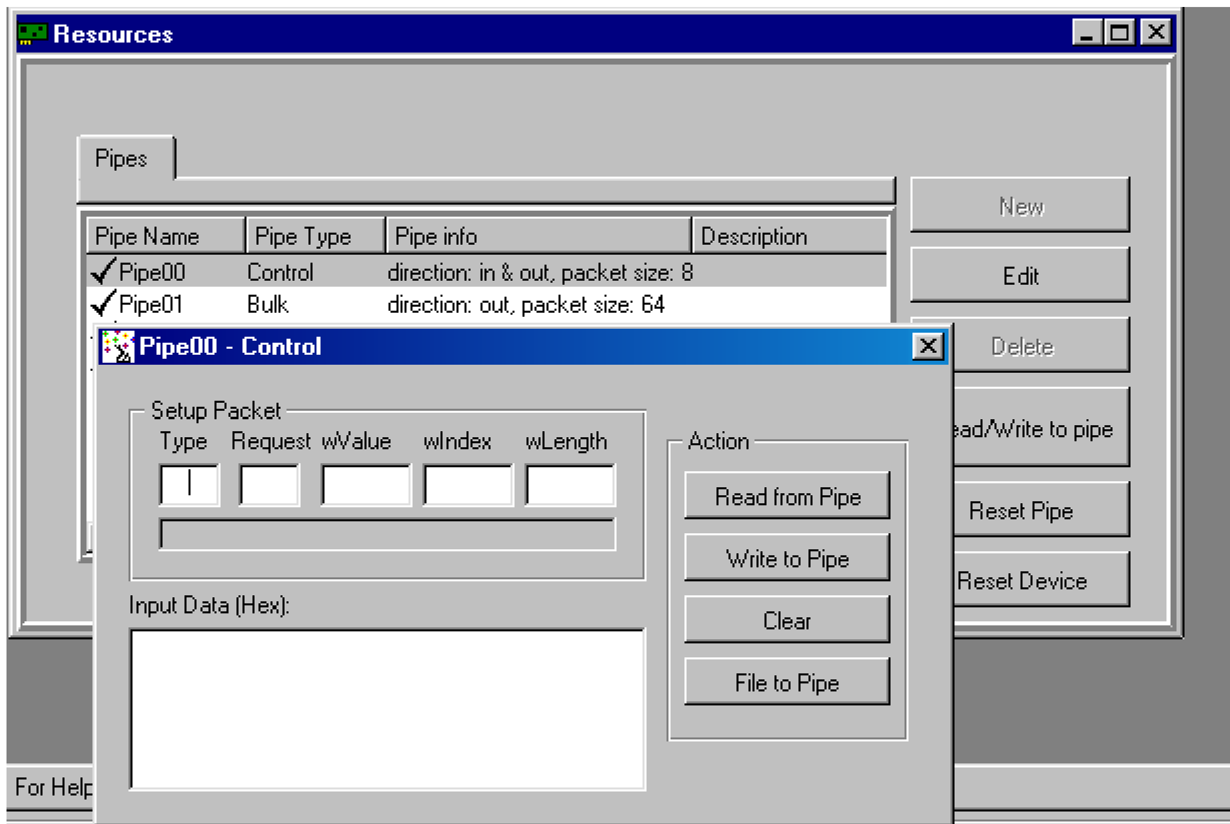


Figure 4.2: USB Device Configuration

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 DriverWizard Logger, so that you may later analyze your testing.
- Make sure your card is performing as expected.

A PCI diagnostic screen

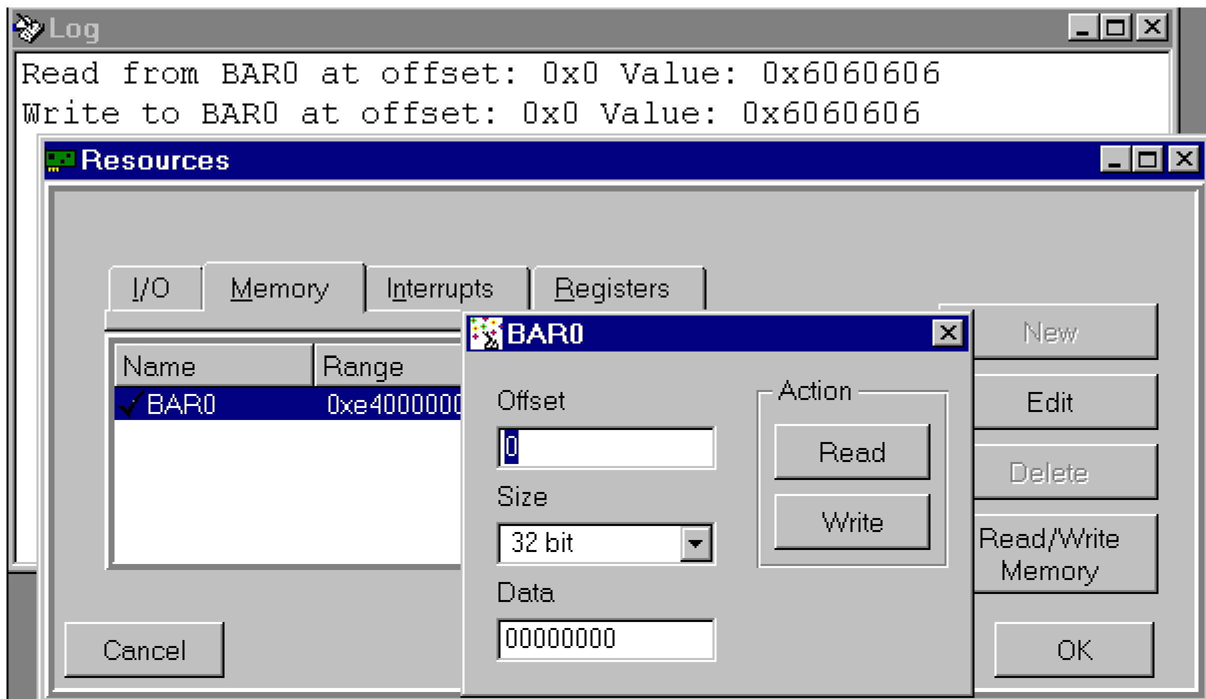


Figure 4.3: A PCI Diagnostics Screen

- For USB testing: DriverWizard shows the pipe detected according to the selected configuration.

In order to perform data transfers follow the steps given below:

- Select the desired pipe.

- For control pipe (a bi-directional pipe) - press 'read/write to pipe'. A new dialog will pop up where you enter a setup packet and for a 'writing operation' you also input data. The setup packet should be 8 bytes long (little endian) and should confirm to the USB specification parameters (bmRequestType, bRequest, wValue, wIndex, wLength).

  More detailed information on how to implement the control transfer and how to send Setup packets can be found under Chapter WinDriver Implementation Issues that explains the WinDriver Implementation Issues

- For input pipe (moves data from device to the host) - click 'listen to pipe'. To successfully accomplish this operation with devices other than 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 DriverWizard will notify you 'Transfer failed').

- To stop reading click 'stop listen to pipe'.

- For output pipe (host to device) - press 'write to pipe'. A new dialog will pop up asking you to enter the data to write. DriverWizard Logger will contain the outcome of the operation.

A USB diagnostics screen

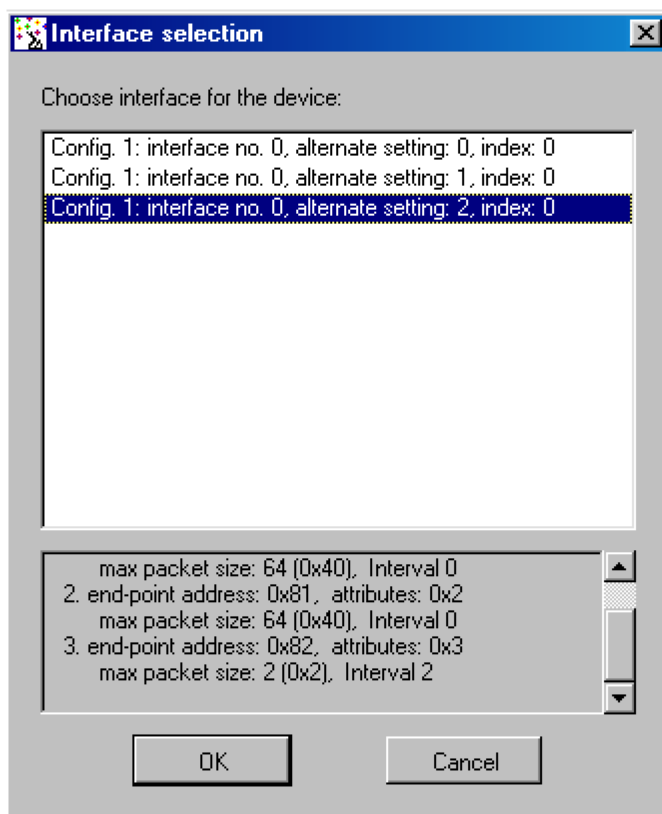


Figure 4.4: USB Diagnostics Screen

5. Generate the skeletal driver code.

- Choose the '**Generate Code**' option from the **Build** menu.

▪

Figure 4.5: Generate Code Option

Select the **WinDriver** option from 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 Jungo `http://www.jungo.com`site url for more details (**Note:** this screen appears only when both WinDriver and KernelDriver are installed on your machine.)

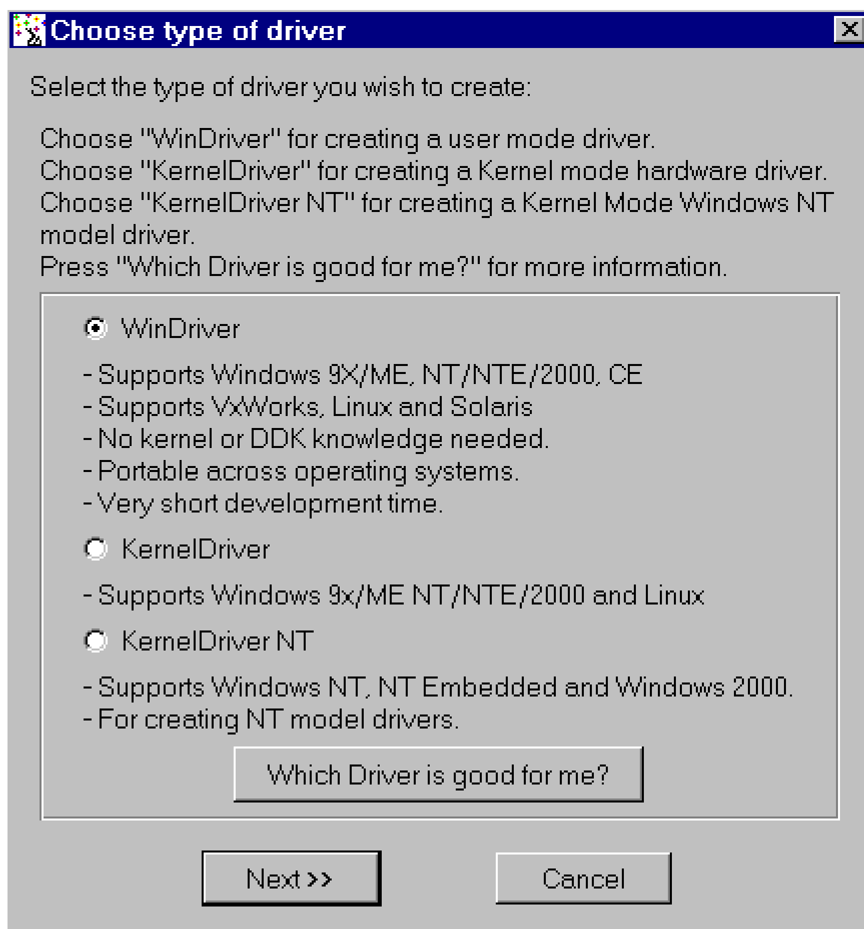


Figure 4.6: Select Driver Type

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

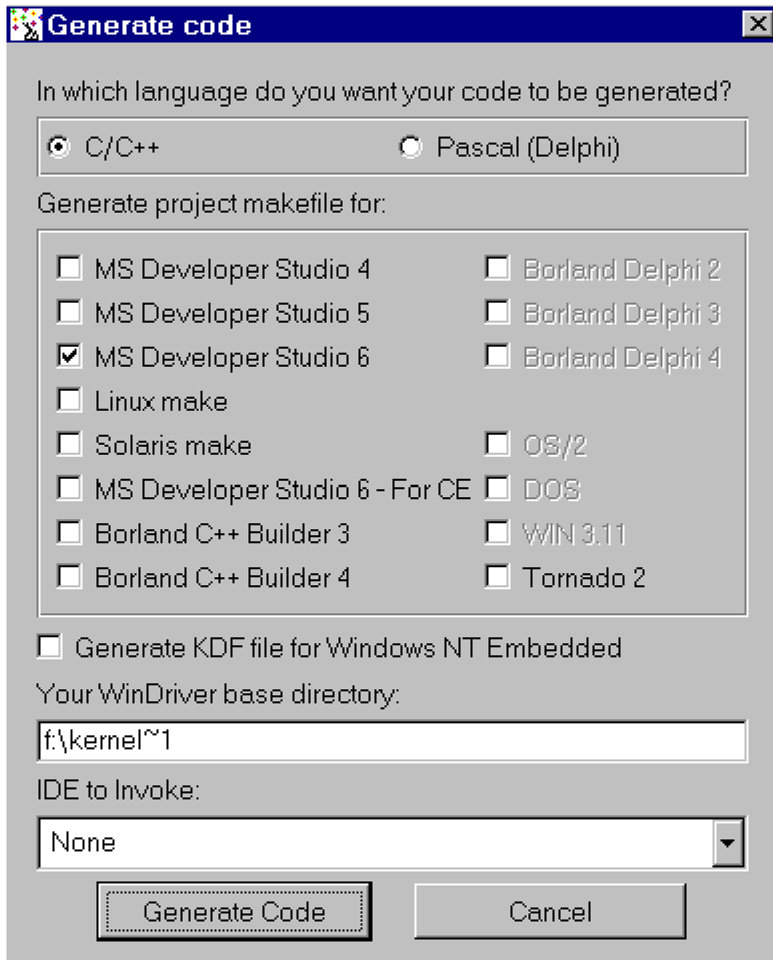


Figure 4.7: Options for Generating Code

- Click the **Generate Code** button at the bottom of the screen.

6. Compile and run the generated code.

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

- The source code DriverWizard creates can be compiled with any 32-bit compiler, and will run on ALL supported platforms (95, 98, ME, NT, 2000, CE, Linux, Solaris,VxWorks and OS/2) without

needing modification.

DriverWizard Notes

Sharing a Resource
Disabling a Resource
DriverWizard Logger
Automatic Code Generation

Remote WinDriver

Remote WinDriver enables driver developers and hardware engineers to develop PCI/PCMCIA/ISA/ISA PnP/EISA/CompactPCI and USB based device drivers on any remote target including embedded systems. Remote WinDriver accesses the hardware on the remote target system from a host machine, tests it and generates a driver for it.

Remote WinDriver consists of two components:

1. The client component which is integrated into DriverWizard.

2. The server component (WinDriver Remote Agent) which runs on the target system and communicates with one or more clients over the TCP/IP network.

Remote WinDriver clients are available on Windows, Linux and Solaris. In version 5.0, server components are supplied for Windows(95/98/ME/NT/2000/CE), Linux, Solaris and VxWorks.

How does it work?

Using Remote WinDriver (For all Windows OSes)

Using Remote WinDriver (For Windows CE)

Using Remote WinDriver (For Linux and Solaris)

Using Remote WinDriver (For VxWorks)

Using the DriverWizard to Build a Device Driver

- Use DriverWizard to diagnose your card. Read / Write to the IO / Memory ranges / registers that your card supports and to the pipes of your USB device. Check that your device operates as expected.

- Use DriverWizard to generate skeletal code for your device in C, C++ or in Delphi. (See Chapter The DriverWizard for details about DriverWizard).

- If you are using one of the supported chip sets (PLX / AMCC / V3 / Altera / Galileo) as your PCI bridge -- it is recommended that you use the **p9030_diag.exe | p9054_diag.exe | p9050_diag.exe | p9080_diag.exe | p9060_diag.exe | iop480_diag.exe | p480_diag.exe | gt64_diag.exe | amccdiag.exe | pbc_diag.exe** (respectively) as your skeletal 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 chapters for more details on using the diagnostics applications).

  NOTE: WinDriver PLX 9050 library is fully compatible with PLX 9052. Therefore, use the files in plx/9050 directory for the PLX 9052 chip.

- Use any 32-bit compiler (such as MSDEV, Visual C/C++, Borland Delphi, Borland C++, Visual Basic, GCC) to generate the skeletal driver you need.

- For Linux and Solaris use gcc to build your code.

- That is all you need to create your User Mode driver. If you discover that better performance is needed, see Chapter Improving Performance for details on performance improvement. That chapter suggests some performance enhancements you can make in your User Mode driver, and instructs you on how to move parts of your code to the WinDriver Kernel PlugIn. This will eliminate any performance problem.

See Chapter WinDriver Function Reference that details the function reference for WinDriver , Chapter WinDriver Structure Reference that details the structure reference for WinDriver and Chapter WinDriver Implementation Issues that explains the WinDriver Implementation Issues for more details.

Writing the Device Driver without the DriverWizard

- It is recommended to use DriverWizard to generate the skeletal driver you need. If you choose to write your driver directly without using DriverWizard, proceed according to the steps outlined below, 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"
```

1. Call **WD_Open()** at the beginning of your program to get a handle for WinDriver.

2. Call **WD_Version()** to make sure that the WinDriver version installed is up to date.

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

4. 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()**.

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

6. 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 an *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.

1. For USB devices call **WD_UsbScanDevice()** to get the unique ID of your device.

2. For USB devices an optional step is to call **WD_UsbGetConfiguration()** to learn about your device configurations and interfaces.

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

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

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

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

Win CE - Testing on CE

Emulation

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 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.

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 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).

1. **WINCE210\PUBLIC\COMMON\DDK\INC\DBGPRINT.H**

2. **\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 / wdpnp.sys*). Using this tool you can monitor how each command sent to the kernel is executed.

Using DebugMonitor

WD_Open()

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

NOTE: If you are a registered user, you need to read the file register.txt under *windriver/redist/register* or *kerneldriver/redist/register*to understand the process of enabling your driver to work with the registered version.

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\n");
}
```

WD_Close()

Closes the WinDriver device. This 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 that is 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, I/O & 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()**[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. If FALSE, then 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" ,
           aBuffer[0], aBuffer[1]);
}
```

WD_PcmciaScanCards()

Scans 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 PCMCIA 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");
}
```

WD_PcmciaGetCardInfo()

Get PCMCIA card information: interrupts, I/O & 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()** [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)

- **pcmciaSlot** - 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. If 0, then 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, I/O & 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()** [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()** [WD_IsapnpScanCards()]
    - **dwOffset** - offset in ISA PnP configuration space to read/write from
    - **fIsRead** - if TRUE, then read config. If FALSE, then 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 devices, see WD_UsbDeviceRegister.

Must be called in order to use interrupts and perform I/O & 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, I/O & memory)
  - **Card.dwItems** - number of items in Card.Item array.
  - **Card.Item[]**- items of card. Each item can be an I/O 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 AN I/O RANGE ITEM

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

FOR A 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 AN INTERRUPT ITEM

- **Card.Item[i].I.Int.dwInterrupt** - interrupt IRQ to install.
  - **Card.Item[i].I.Int.dwOptions** - usually 0. For level sensitive interrupts use INTERRUPT_LEVEL_SENSITIVE.
  - **Card.Item[i].I.Int.hInterrupt** -returns an interrupt handle to use with WD_IntEnable().
  - **fCheckLockOnly** -should be set to FALSE to register the card. In order to just check whether a card can be registered (i.e.: not used by someone else), it should be TRUE.

- **hCard** - returns the handle of the card, or 0 if card cannot be registered. If the *fCheckLockOnly* flag is set to TRUE, then hCard will return 1 if the 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\n");
```

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 I/O 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 I/O, or User address for memory transfer. User address for a memory mapped card is returned by **WD_CardRegister()**[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** - If TRUE then I/O or memory address should be incremented for transfer. If FALSE, then 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 I/O & memory transfers.

Prototype

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

Parameters

- **pTransArray** - array of transfer commands, same as in WD_Transfer()[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 DriverWizard, and letting DriverWizard generate the code for you. (In Plug-n-Play cards, DriverWizard 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()** [WD_CardRegister()], in the Card structure.
- **Cmd** - an 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()** [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\n");
else
{
 BZERO(Intrp);
 Intrp.hInterrupt =
     cardReg.Card.Item[0].I.Int.hInterrupt;
 Intrp.Cmd = NULL;
 Intrp.dwCmds = 0;
 Intrp.dwOptions = 0;
 WD_IntEnable(hWD, &Intrp);
}
if (!Intrp.fEnableOk)
   printf("failed enabling interrupt\n");
}
```

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.

1. 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.
2. 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 1

User buffer DMA (scatter gather locking)

```
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");
```

Example 2

The following code shows kernel buffer DMA

```
BZERO(Dma)
Dma.dwBytes =20 * 4096; //(20 pages)
Dma.dwOptions=DMA_KERNEL_BUFFER_ALLOC;
{
WD_DMALock (hWD, &Dma);
// on return Dma.Page has the list of physical addresses
if (Dma.hDma==0)
 printf("Failed allocating kernel buffer for DMA\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 installed devices.

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 unique USB ID's where matching devices were detected.
- **deviceGeneralInfo[]** - general information (device address, number of configurations.....) about the devices.

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 atleast one card
{
 uniqueId = scan.uniqueId[0];
}
else
{
 printf("No matching USB devices found\n");
}
```

WD_UsbGetConfiguration()

Get information about a USB device.

Prototype

void WD_UsbGet Configuration(HANDLE hWD, WD_USB_CONFIGURATION *pConfig);

Parameters(WD_USB_CONFIGURATION elements)

- **uniqueId** - the unique ID of the device as received from WD_UsbScan Device()
  - **dwConfigurationIndex** - the index of the configuration to get (zero based). The number of configurations are received from **WD_UsbScanDevice()**[WD_UsbScanDevice()] in the deviceGeneralInfo
  - **configuration** - configuration general data - (value, attributes...)
  - **dwInterfaceAlternatives** - how many interfaces (and alternate interfaces) are on the device.
- **Interface[]** - list of interface 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 unique deviceID as received from **WD_UsbScanDevice()**[WD_UsbScanDevice()]

  - **dwConfigurationIndex** - the index of the configuration to register (zero based). The number of configurations are received from **WD_UsbScanDevice()** [WD_UsbScanDevice()] in the deviceGeneralInfo

  - **dwInterfaceNum** - interface number to register as received from **WD_UsbGetConfiguration()** [WD_UsbGetConfiguration()]

- **hDevice** - the handle of the device returned

  - **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.dwInterfaceAlternative = 1;
WD_DeviceRegister(hWD, &device);
if(!device.{hDevice}
 printf("error - could not register device\n");
else
 printf("device 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 from / 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 the USB device as received from **WD_UsbDeviceRegister()** [WD_UsbDeviceRegister()]
  - **dwPipe** - pipe number of the device
  - **fRead** - perform Read or Write
  - **dwOptions** - can be USB_TRANSFER_HALT to halt the previous transfer on the same pipe
  - **pBuffer** - pointer to buffer to read/write
  - **dwBytes** - size of the buffer
- **dwTimeout** - timeout for the transfer in milliseconds. 0 →No timeout
  - **dwBytesTransferred** - returns the number of bytes actually read / written.
  - **SetupPacket[8]** - 8 bytes setup packet for control pipe transfer
  - **fOK** - return true if transfer is 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("Transferred %d bytes from %d\n",
          trans.dwBytesTransferred,trans.dwBytes);
```

WD_UsbResetPipe()

Reset the pipe to its default state (resets the state machine of the firmware's pipe to its initial state)

Prototype

void WD_UsbResetPipe(HANDLE hWD, WD_USB_RESET_PIPE *pReset);

Parameters(WD_USB_RESET_PIPE elements)

- **hDevice** - handle of the USB Device
    - **dwPipe** - The pipe number to reset

Example

```
WD_USB_RESET_PIPE reset;

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

InterruptThreadEnable()

Convenience function for setting up interrupt handling. This function is implemented as a static function in the header file *windrvr_int_thread.h* found under **windriver/include**

Prototype

BOOL InterruptThreadEnable(HANDLE *phThread, HANDLE hWD, WD_INTERRUPT *pInt, HANDLER_FUNC func, PVOID pData)

Parameters

- **phThread** - returns the handle of the spawned interrupt thread. This should be passed to InterruptThreadDisable when shutting down the interrupt handling
- **hWD** - the handle to WinDriver as returned by **WD_Open()**[WD_Open()]
- **pInt** - the pointer to an initialized WD_INTERRUPT[WD_INTERRUPT]structure describing the interrupt to connect to.
- **func** - the interrupt handling function. This function will be called once at every interrupt occurence. HANDLER_FUNC is defined in *windrvr_int_thread.h*
- **pData** - this pointer is passed to the interrupt handling function as an argument.

Example

```
VOID interrupt_handler (PVOID pData)
{
    WD_INTERRUPT * pIntrp = (WD_INTERRUPT *) pData;
    // do your interrupt routine here
    printf ("Got interrupt %d\n", pIntrp->dwCounter);
}

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

```
    ...
    hWD = WD_Open();
    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_INTERRUPT;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.Int.dwInterrupt = MY_IRQ;
    cardReg.Card.Item[0].I.Int.dwOptions = 0;
    ...
    WD_CardRegister (hWD, &cardReg);
    ...
    PVOID  pData = NULL;
    BZERO(Intrp);
    Intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
    Intrp.Cmd = NULL;
    Intrp.dwCmds = 0;
    Intrp.dwOptions = 0;
    printf ("starting interrupt thread\n");
    pData = &Intrp;
    if (!InterruptThreadEnable(&thread_handle, hWD, &Intrp,
         interrupt_handler, pData))
      {
        printf ("failed enabling interrupt\n");
      }
    else
      {
        printf ("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);
        // this calls WD_IntDisable()
        InterruptThreadDisable(thread_handle);
      }
    WD_CardUnregister(hWD, &cardReg);
    ....
}
```

InterruptThreadDisable()

Convenience function for shutting down interrupt handling. This function is implemented as a static function in the header file *windrvr_int_thread.h* found under **windriver/include**

Prototype

VOID InterruptThreadDisable(HANDLE hThread)

Parameters

- **hThread** - The handle of the spawned interrupt thread which was created by InterruptThreadEnable

Example

```
main()
{
    ....
    if (!InterruptThreadEnable(&thread_handle, hWD, &Intrp,
         interrupt_handler, pData))
      {
   printf ("failed enabling interrupt\n");
      }
    else
      {
       printf ("Press Enter to uninstall interrupt\n");
       fgets(line, sizeof(line), stdin);
       // this calls WD_IntDisable()
       InterruptThreadDisable(thread_handle);
      }
    WD_CardUnregister(hWD, &cardReg);
    ....
}
```

WD_TRANSFER

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

Used by **WD_Transfer()** [WD_Transfer()], **WD_MultiTransfer()**[WD_MultiTransfer()], **WD_IntEnable()** [WD_IntEnable()].

MEMBERS:

TYPE	NAME	DESCRIPTION
DWORD	cmdTrans	Transfer command WD_TRANSFER_CMD
DWORD	dwPort	i/o port for transfer or user mem address
DWORD	dwBytes	Number of bytes for string trans
DWORD	fAutoinc	transfer from one port/address c 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_DMA

Contains information about a DMA buffer. Used by **WD_DMALock()** [WD_DMALock()] and **WD_DMAUnlock()** [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 fl '0' for no option, or: DMA_KERNEL_BUFFER_ALLO DMA_KBUF_BELOW_16M DMA_LARGE_BUFFER
DWORD	dwPages	Number of pages in the buffer
WD_DMA_ PAGE [WD_DMA_PAGE]	Page [WD_DMA_ PAGES]	Array of pages in the buffer

WD_DMA_PAGE

MEMBERS:

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

WD_INTERRUPT

Used to describe an interrupt

Used by **WD_IntEnable()** [WD_IntEnable()], **WD_IntDisable()**[WD_IntDisable()], **WD_IntWait()** [WD_IntWait()], **WD_IntCount()** [WD_IntCount()],**InterruptThreadEnable()** [InterruptThreadEnable()].

MEMBERS:

TYPE	NAME	DESCRIPTION
DWORD	hInterrupt	handle of interrupt
DWORD	dwOptions	interrupt options: Bit masked fla '0' for no option, or: INTERRUPT_LEVEL_SENSITI level sensitive interrupts) or INTERRUPT_CMD_COPY(choo when you need the WinDriver k copy theactions of the read com has done to acknowledge the in back to the user mode)
WD_TRANSFER [WD_TRANSFER]	*Cmd	Pointer to commands to perform 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 dea
DWORD	fStopped	was interrupt disabled during wa

WD_VERSION

Describes version of WinDriver in use. Used by **WD_Version()** [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_CardRegister()], **WD_CardUnregister()**[WD_CardUnregister()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_CARD	Card	card to register
DWORD	fCheckLock Only	only check if card is lockable, re 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 [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 share
union	I	Item specific information
struct	I.Mem	ITEM_MEMORY
DWORD	I.Mem.dw PhysicalAddr	Physical address on card
DWORD	I.Mem.dwBytes	Address range
DWORD	I.Mem.dwTrans Addr	Returns the address to pass on commands
DWORD	I.Mem.dwUser DirectAddr	Returns the address for direct u read/write
DWORD	dwCpuPhysical Addr	returns the CPU physical addres
struct	I.IO	ITEM I/O
DWORD	I.IO.dwAddr	Beginning of I/O address
DWORD	I.IO.dwBytes	I/O range
struct	I.Int	ITEM INTERRUPT
DWORD	I.Int.dwInterrupt	Number of the interrupt to instal
DWORD	I.Int.dwOptions	interrupt options:INTERRUPT_LEVEL_S
DWORD	I.Int.hInterrupt	Returns the handle of the interru installed

WD_SLEEP

Defines a sleep command.

Used by **WD_Sleep()** [WD_Sleep()].

MEMBERS:

TYPE	NAME	DESCRIPTION
DWORD	dwMicro Seconds	Sleep time in micro seconds - 1/ 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 car
DWORD	dwSlot	PCI physical slot number of car
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()** [WD_PciScanCards()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCI_ID	searchId	If searchId.dwVendorId==0, ther vendor IDs. If searchId.dwDevic then scan all device IDs.
DWORD	dwCards	Number of cards found
WD_PCI_ID [WD_PCI_ID]	cardId [WD_PCI_ CARDS]	VendorID & DeviceID of cards fo
WD_PCI_SLOT [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()** [WD_PciGetCardInfo()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCI_ SLOT [WD_PCI_SLOT]	pciSlot	PCI slot
WD_CARD [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()** [WD_PciConfigDump()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCI_ SLOT [WD_PCI_SLOT]	pciSlot	PCI bus,slot and function numbe
PVOID	pBuffer	buffer for read/write
DWORD	dwOffset	offset in PCI configuration space read/write from
DWORD	dwBytes	bytes to read/write from/to buffe the number of bytes read/writter
DWORD	fIsRead	FALSE - write PCI config TRUE config
DWORD	dwResult	0 - PCI_ACCESS_OK - read/wr PCI_ACCESS_ERROR - error 2 PCI_BAD_BUS - bus does not e only) 3 - PCI_BAD_SLOT - slot 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 [WD_ISAPNP_CARD_ID]	cardId	Vendor ID and serial number of
DWORD	dwLogicalDevices	Number of logical devices on the
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()** [WD_IsapnpScanCards()].

MEMBERS:

TYPE	**NAME**	**DESCRIPTION**
WD_ISAPNP_ CARD_ID[WD_ISAPNP_CARD_ID]	searchId	If searchId.cVendorId[0]==0, the vendor IDs. If searchId.dwSerial scan all serial numbers.
DWORD	dwCards	Number of cards found
WD_ISAPNP_ CARD [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()** [WD_IsapnpGetCardInfo()]

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_ CARD_ID [WD_ISAPNP_CARD_ID]	cardId	Vendor ID and serial number of which information is required
DWORD	dwLogicalDevice	Number of the logical device for information is requested
WD_ISAPNP_ COMP_ID	clogicalDeviceId[8]	ascii of logical device id found
DWORD	dwCompatibleDevices	Number of compatible devices f
WD_ISAPNP_ COMP_ID	CompatibleDevice [WD_ISAPNP_COMPATIBLE_ IDS]	Compatible device IDs
WD_ISAPNP_ ANSI	cIdent	Identity of device
WD_CARD [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()** [WD_IsapnpConfigDump()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_ISAPNP_ CARD_ID [WD_ISAPNP_CARD_ID]	cardId	VendorID and serial number of c
DWORD	dwOffset	offset in ISA PnP configuration s read/write from
DWORD	fIsRead	if 1, then read ISA PnP config if 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 (fir 0)
BYTE	uFunction	Specifies the function number (f function is 0)

WD_PCMCIA_ID

Defines the identity of a PCMCIA card.

MEMBERS:

TYPE	NAME	DESCRIPTION
CHAR	cVersion[WD_PCMCIA_ VERSION_LEN]	The Card's PCMCIA version
CHAR	cManufacturer[WD_PCMCIA_MANUFACTURER_LEN]	Manufacturer name
CHAR	cProductName[WD_ PCMCIA_PRODUCTNAME_LEN]	Product name
USHORT	cCheckSum	Card's CRC checksum value

WD_PCMCIA_SCAN_CARDS

Receives information on cards detected on the PCMCIA bus.

Used by **WD_PcmciaScanCards()** [WD_PcmciaScanCards()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCMCIA_ ID [WD_PCMCIA_ID]	searchId	if strlen(searchId.cManufacturer scan all Manufacturers. if strlen(searchId.cProductName) scan all product names.
DWORD	dwCards	Number of cards found
WD_PCMCIA_ ID [WD_PCMCIA_ID]	cardId[WD_ PCMCIA_ CARDS]	Manufacturer Name, Product Na Version and CRC Info of card fo
WD_PCMCIA_ SLOT [WD_PCMCIA_SLOT]	cardSlot[WD_ PCMCIA_ CARDS]	PCMCIA slot/function info of car

WD_PCMCIA_CARD_INFO

Describes a PCMCIA card's resources detected.

Used by **WD_PcmciaGetCardInfo()** [WD_PcmciaGetCardInfo()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCMCIA_ SLOT [WD_PCMCIA_SLOT]	pcmciaSlot	PCMCIA slot information
WD_CARD [WD_CARD]	Card	get card parameters for PCMCI

WD_ PCMCIA_CONFIG_DUMP

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

Used by **WD_PcmciaConfigDump()** [WD_PciConfigDump()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_ PCMCIA_ SLOT [WD_PCMCIA_SLOT]	pcmciaSlot	Slot descriptor of PCMCIA card
PVOID	pBuffer	buffer for read/write
DWORD	dwOffset	offset in **pcmcia** PnP configurat from which to read/write
DWORD	dwBytes	bytes to read from or write to bu Returns the number of bytes rea
DWORD	fIsRead	if 1, then read pci config if 0, the 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 default pipe)
DWORD	dwMaximum PacketSize	the maximum packet size of inte transfers on the pipe
DWORD	type	Control, Isochronous, Bulk or In
DWORD	direction	In=1, out=2 or in&out=3
DWORD	dwInterval	Intervals of data transfer in ms (Interrupt pipes)

WD_USB_CONFIG_DESC

Describes a configuration.

MEMBERS

TYPE	NAME	DESCRIPTION
DWORD	dwNumInter faces	the configuration number
DWORD	dwValue	the device value
DWORD	dwAttributes	the device attributes
DWORD	Maxpower	the device MaxPower

WD_USB_INTERFACE_DESC

Describes an interface.

MEMBERS

TYPE	NAME	DESCRIPTION
DWORD	dwNumber	the interface number
DWORD	dwAlternate Setting	the interface alternate value
DWORD	dwNumEnd points	the number of endpoints in the i
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	dwEndpoint Address	end point address
DWORD	dwAttributes	end point attributes
DWORD	dwMaxPacket Size	maximum packet size
DWORD	dwInterval	interval in milli-seconds

WD_USB_INTERFACE

Holds interface data.

MEMBERS

TYPE	NAME	DESCRIPTION
WD-USB-INTER FACE-DESC [WD_USB_INTERFACE_DESC]	Interface	the interface description
WD-USB-END POINT-DESC [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	dwConfiguration Index	the Configuration Index
WD-USB-CONFIG- DESC [WD_USB_CONFIG_DESC]	configuration	the configuration description
DWORD	dwInterfaceAl ternatives	number of interfaces and their a
WD-USB-INTER FACE [WD_USB_INTERFACE]	Interface[]	list of configuration interfaces

WD_USB_HUB_GENERAL_INFO

Holds hub information (if the selected device is a hub).

MEMBERS

TYPE	NAME	DESCRIPTION
DWORD	fBusPowered	is bus powered or self powered
DWORD	dwPorts	number of ports on this hub
DWORD	dwCharacter istics	hub characteristics
DWORD	dwPowerOn ToPowerGood	port power on till power good in
DWORD	dwHubControl Current	max current in mA

WD_USB_DEVICE_GENERAL_INFO

General information about the device.

MEMBERS

TYPE	NAME	DESCRIPTION
WD_USB_ID [WD_USB_ID]	deviceId	the vendor ID and product ID of
DWORD	dwHubNum	the number of the hub to which t is attached
DWORD	dwPortNum	the number of the port on the hu the device is attached
DWORD	fHub	is the device itself a hub?
DWORD	fFullSpeed	full speed or low speed device?
DWORD	dwConfigurations Num	how many configurations does t have?
DWORD	deviceAddress	the physical address of the devi
WD_USB_ HUB_GENERAL_ INFO [WD_USB_HUB_GENERAL_INFO]	hubInfo	contains information about the d 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 [WD_USB_PIPE_INFO]	Pipe[]	the list of pipes information

WD_USB_SCAN_DEVICES

Define a scan command.

MEMBERS

TYPE	NAME	DESCRIPTION
WD_USB_ ID [WD_USB_ID]	searchId	if dwvendorId ==0, then scan all IDs. if dwProductId ==0, then sc products.
DWORD	dwDevices	Number of devices found
DWORD	uniqueId[]	a list of the uniqueIDs to identify devices
WD_USB_ DEVICE_ GENERAL_ [WD_USB_DEVICE_GENERAL_INFO]	deviceGeneral Info[]	list of general information about devices found

WD_USB_TRANSFER

Defines a transfer command.

MEMBERS

TYPE	NAME	DESCRIPTION
DWORD	hDevice	handle of USB device to read fr to
DWORD	dwPipe	pipe number on device
DWORD	fRead	read or write
DWORD	dwOptions	can be USB_TRANSFER_HAL the previous transfer on the sam
DWORD	pBuffer	pointer to buffer to read / write
DWORD	dwBytes	the size of the buffer
DWORD	dwTimeout	transfer timeout(milliseconds) 0 timeout
DWORD	dwBytes Transfered	returns the number of bytes act written
BYTE	SetupPacket[8]	setup packet for control pipe tra
DWORD	fOK	return TRUE if the transfer is su

WD_USB_DEVICE_REGISTER

Define a device registration command.

MEMBERS

TYPE	NAME	DESCRIPTION
DWORD	uniqueId	the unique ID of the device
DWORD	dwConfiguration Index	the index of the configuration to
DWORD	dwInterfaceNum	interface to register
Dword	dwInterfaceAl ternate	alternate number of the interface register
DWORD	hDevice	handle of the device
WD_USB_ DEVICE_ INFO [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 a reset pipe command.

MEMBERS

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

Overview of the Development Process for the PLX9050 Using WinDriver's PLX9050 API

1. Run the PLX9050 diagnostics (P50_DIAG.EXE) to diagnose your card.

2. Use the P50_DIAGs source code as your skeletal device driver.

3. Modify the P50_DIAGs code to suit your application. Use the PLX9050 function reference to add your own code.

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

What is the PLX9050 Diagnostics?

PLX9050 Diagnostics is a ready-to-run sample diagnostics application for PLX9050. The diagnostics program accesses the hardware via WinDrivers PLX9050 API (P50LIB.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 PLX9050 API.

This application can be used as your skeletal 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). Besides P50_DIAG.C being an example of using the PLX9050 API, it is also a useful diagnostics utility.

Using PLX9050 Diagnostics

Introduction
Main Menu Options

Creating the driver project without using the P50_DIAG as the skeletal driver code

- Add P50LIB.C to your project or your make file.

- Include p50lib.h in your driver source code.

NOTE: In your \windrvr\plx\9050\p50_diag folder, you will find the source code for P50_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 your skeletal code.

- Call P9050_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, and more, using any of these functions:

1. P9050_IsAddrSpaceActive()

2. P9050_ReadReg ()

3. P9050_WriteReg ()

4. P9050_ReadSpaceBlock()

5. P9050_WriteSpaceBlock()

6. P9050_ReadSpaceByte()

7. P9050_ReadSpaceWord()

8. P9050_ReadSpaceDWord()

9. P9050_WriteSpaceByte()

10. P9050_WriteSpaceWord()

11. P9050_WriteSpaceDWord()

12. P9050_ReadBlock()

13. P9050_WriteBlock()

14. P9050_ReadByte()

15. P9050_ReadWord()

16. P9050_ReadDWord()

17. P9050_WriteByte()

18. P9050_WriteWord()

19. P9050_WriteDWord()

20. P9050_IntIsEnabled()

21. P9050_IntEnable()

22. P9050_IntDisable()

23. P9050_EEPROMReadWord()

24. P9050_EEPROMWriteWord()

25. P9050_ReadPCIReg()

26. P9050_WritePCIReg()

- Call P9050_Close() before end of code.

NOTES:

1. Using one of the sample drivers included with WinDriver as your skeletal code may shorten the development process. (see Section Sample Code that illustrates the sample code.)

2. After development of the driver is through, follow the Re-Distribution procedure.

Sample

WinDriver's PLX9050 API Function Reference

Use this section as a quick reference to the WinDrivers PLX 9050 API functions. This reference may also be found in the WinDriver Help files. Advanced users may find more functionality in the WinDrivers API All the functions outlined in the FunctionReference are implemented in the \windrvr\plx\9050\lib\p50lib.c file. The definition of the structures used in the following functions may be found in the PLX 9050 structure reference.

P9050_CountCards()

P9050_Open()

P9050_Close()

P9050_IsAddrSpaceActive()

P9050_ReadReg(), P9050_WriteReg()

P9050_ReadSpaceByte(), P9050_ReadSpaceWord(), P9050_ReadSpaceDWord()

P9050_WriteSpaceByte(), P9050_WriteSpaceWord(), P9050_WriteSpaceDWord()

P9050_ReadSpaceBlock(), P9050_WriteSpaceBlock()

P9050_ReadByte(),P9050_ReadWord()P9050_ReadDWord()

P9050_WriteByte(),P9050_WriteWord(),P9050_WriteDWord()

P9050_ReadBlock(),P9050_WriteBlock()

P9050_IntIsEnabled()

P9050_IntEnable()

P9050_IntDisable()

P9050_EEPROMReadWord(), P9050_EEPROMWriteWord()

P9050_ReadPCIReg(),P9050_WritePCIReg()

PLX 9050 API Structure Reference

These structures are used only in conjunction with the PLX API.

PLX Register Definitions
P9050_MODE
P9050_ADDR
P9050_INT_RESULT

Trouble-Shooting

P9050_Open() fails

Overview of the Development Process for the PLX9060 / PLX9080 Using WinDriver's PLX9060 API

1. Run the PLX9060 diagnostics (P60_DIAG.EXE) to diagnose your card.

2. Use the P60_DIAGs source code as your skeletal device driver.

3. Modify the P60_DIAGs code to suite your application. Use the PLX9060 function reference to add your own code.

4. If the User Mode driver you have created in the above steps contains some parts for which the performance must be enhanced (an interrupt handler for example), see Section Kernel PlugIn. It shows you how to move parts of your source code to WinDriver's Kernel PlugIn.

What is PLX9060 Diagnostics?

PLX9060 Diagnostics is a ready-to-run sample diagnostics application for PLX9060. The diagnostics program accesses the hardware via WinDrivers PLX 9060/9080 API (P60LIB.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 PLX 9060/9080 API.

This application can be used as your skeletal 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). Besides P60_DIAG.C being an example of using the PLX 9060/9080 API, it is also a useful diagnostics utility.

Using PLX9060 Diagnostics

Introduction
Main Menu Options

Creating the Driver Project without using the P60_DIAG as the Skeletal Driver Code

- Add P60LIB.C to your project or your make file.
    - Include p60lib.h in your driver source code.

NOTE: In your \windrvr\plx\9060\p60_diag folder, you will find the source code for P60_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 your skeletal code.

- Call P9060_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, and more, using any of these functions:

1. P9060_IsAddrSpaceActive()
2. P9060_ReadReg()
3. P9060_WriteReg()
4. P9060_ReadSpaceBlock()
5. P9060_WriteSpaceBlock()
6. P9060_ReadSpaceByte()
7. P9060_ReadSpaceWord()
8. P9060_ReadSpaceDWord()
9. P9060_WriteSpaceByte()
10. P9060_WriteSpaceWord()
11. P9060_WriteSpaceDWord()
12. P9060_ReadBlock()
13. P9060_WriteBlock()

14. P9060_ReadByte()

15. P9060_ReadWord()

16. P9060_ReadDWord()

17. P9060_WriteByte()

18. P9060_WriteWord()

19. P9060_WriteDWord()

20. P9060_IntIsEnabled()

21. P9060_IntEnable()

22. P9060_IntDisable()

23. P9060_ReadPCIReg()

24. P9060_WritePCIReg()

- Call P9060_Close() before end of code.

NOTES:

1. Using one of the sample drivers included with WinDriver as your skeletal code may shorten the development process.

2. After development of the driver is through, follow the Re-Distribution procedure.

Sample Code

WinDriver's PLX 9060/9080 API Function Reference

Use this appendix as a quick reference to the WinDrivers PLX 9060/9080 functions. This reference may also be found in the WinDriver Help files. Advanced users may find more functionality in the WinDriver API.

All the functions outlined in FunctionReference are implemented in the \windrvr\plx\9060 \lib\p60lib.c file. The definition of the structures used in the following functions may be found in the PLX 9060 / 9080 structure reference.

P9060_CountCards()

P9060_Open()

P9060_Close()

P9060_IsAddrSpaceActive()

P9060_ReadReg(), P9060_WriteReg()

P9060_ReadSpaceByte() ,P9060_ReadSpaceWord() , P9060_ReadSpaceDWord()

P9060_WriteSpaceByte(),P9060_WriteSpaceWord(), P9060_WriteSpaceDWord()

P9060_ReadSpaceBlock() ,P9060_WriteSpaceBlock()

P9060_ReadByte(),P9060_ReadWord(),P9060_ReadDWord()

P9060_WriteByte(), P9060_WriteWord(), P9060_WriteDWord()

P9060_ReadBlock(),P9060_WriteBlock()

P9060_IntIsEnabled()

P9060_IntEnable()

P9060_IntDisable()

P9060_ReadPCIReg(),P9060_WritePCIReg()

PLX 9060 API Structure Reference

These structures are used only in conjunction with the PLX API.

PLX Register Definitions
P9060_MODE
P9060_INT_RESULT

Trouble-Shooting

P9060_Open() fails.

Overview of the Development Process for the AMCC S5933 Using WinDriver's AMCC API

1. Run the AMCC-S5933 diagnostics (AMCCDIAG.EXE) to diagnose your card.

2. Use the AMCCDIAGs source code as your skeletal device driver.

3. Modify the AMCCDIAGs code to suite your application. Use the AMCC S5933 function reference to add your own code.

4. If the User Mode driver you have created in the above steps contains some parts which in which the performance must be enhanced (an interrupt handler for example), see Section Kernel PlugIn . It will show you how to move parts of your source code to the WinDriver Kernel PlugIn.

What is AMCC Diagnostics?

AMCC Diagnostics is a ready-to-run sample diagnostics application for AMCC S5933. The diagnostics program accesses the hardware via WinDriver's AMCC API (AMCCLIB.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 AMCC API.

This application can be used as your skeletal 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). Besides AMCCDIAG.C being an example of using the AMCC API, it is also a useful diagnostics utility.

Using the AMCC Diagnostics

Introduction
Main Menu Options

Creating the Driver Project without using the AMCCDIAG as the Skeletal Driver Code

- Add AMCCLIB.C to your project or your make file.
    - Include amcclib.h in your driver source code.

NOTE: In your \windrvr\amcc\amccdiag folder, you will find the source code for AMCCDIAG.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 your code.

- Call AMCC_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, and more, using any of these functions:

1. AMCC_IsAddrSpaceActive()
2. AMCC_ReadReg ()
3. AMCC_WriteReg ()
4. AMCC_ReadByte()
5. AMCC_ReadWord()
6. AMCC_ReadDWord()
7. AMCC_WriteByte()
8. AMCC_WriteWord()
9. AMCC_WriteDWord()
10. AMCC_ReadSpaceBlock()
11. AMCC_WriteSpaceBlock()
12. AMCC_IntIsEnabled()
13. AMCC_IntEnable()

14. AMCC_IntDisable()

15. AMCC_ReadPCIReg()

16. AMCC_WritePCIReg()

17. AMCC_DMAOpen()

18. AMCC_DMAClose()

19. AMCC_DMAStart()

20. AMCC_DMAIsDone()

- Call AMCC_Close() before end of code.

NOTES:

1. Using one of the sample drivers included with WinDriver as your skeletal code may shorten the development process. (see Section Sample Code that illustrates the sample code).

2. After development of the driver is through, follow the Re-Distribution procedure.

Sample Code

Sample uses of WinDriver and WinDriver for AMCC are supplied with the WinDriver toolkit. You may find the WinDriver samples under \windrvr\samples, and the WinDriver for AMCC samples under \windrvr\AMCC.

Each directory contains a files.txt file which describes the various samples included. Each sample is located in its own directory. For your convenience, we have supplied an mdp file alongside of each .c file, so that users of Microsoft's developer 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 AMCCLIB in their project) You may learn to use the WinDriver for AMCC API by looking at amcclib.c which is found in \windrvr\amcc\amccdiag.c . amccdiag.c is the source code of the AMCC diagnostics program found under Start \Programs \WinDriver.

WinDriver's AMCC S5933 API Function Reference

Use this appendix as a quick reference to the WinDrivers AMCC S5933 functions. This reference may also be found in the WinDriver Help files. Advanced users may find more functionality in the WinDrivers API All of the functions outlined in FunctionReference are implemented in the \windrvr\amcc\lib\amcclib.c file. The definition of the structures used in the following functions may be found in the AMCC S5933 structure reference.

AMCC_CountCards()

AMCC_Open()

AMCC_Close()

AMCC_IsAddrSpaceActive()

AMCC_ReadRegByte(),AMCC_ReadRegWord(), AMCC_ReadRegDWord()

AMCC_WriteRegByte(),AMCC_WriteRegWord(), AMCC_WriteRegDWord()

AMCC_ReadByte(), AMCC_ReadWord(), AMCC_ReadDWord()

AMCC_WriteByte(),AMCC_WriteWord(),AMCC_WriteDWord()

AMCC_ReadSpaceBlock(), AMCC_WriteSpaceBlock()

AMCC_IntIsEnabled()

AMCC_IntEnable()

AMCC_IntDisable()

AMCC_ReadPCIReg(), AMCC_WritePCIReg()

AMCC S5933 API Structure Reference

PCI Configuration Registers Definitions
AMCC_ADDR
AMCC_INT_RESULT

Trouble-Shooting

AMCC_Open() fails.

Overview of the Development Process for the V3 PBC using WinDriver's V3 PBC API

1. Run the V3-PBC diagnostics (PBC_DIAG.EXE) to diagnose your card.

2. Use the PBC_DIAGs source code as your skeletal device driver.

3. Modify the PBC_DIAGs code to suit your application. Use the V3 PBC function reference to add your own code.

4. If the User Mode driver you have created in the above steps contains some parts which in which the performance must be enhanced (an interrupt handler for example), see Chapter Kernel PlugIn . It shows you how to move parts of your source code to the WinDriver Kernel PlugIn.

What is V3 PBC Diagnostics?

V3 PBC Diagnostics is a ready-to-run sample diagnostics application for V3 PBC. The diagnostics program accesses the hardware via WinDriver's V3 PBC API (PBCLIB.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 V3 PBC API.

This application can be used as your skeletal 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). Besides PBC_DIAG.C being an example of using the V3 PBC API, it is also a useful diagnostics utility.

Using the V3 PBC Diagnostics

Introduction
Main Menu Options

Creating the Driver Project without using the PBC_DIAG as the Skeletal Driver Code

- Add PBCLIB.C to your project or your make file.
  - Include pbclib.h in your driver source code.

***NOTE**: In your \windrvr\V3\pbc_diag folder, you will find the source code for PBC_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 your skeletal code.*

- Call V3PBC_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, and more, using any of these functions:

1. V3PBC_IsAddrSpaceActive()
2. V3PBC_GetRevision()
3. V3PBC_ReadRegByte ()
4. V3PBC_ReadRegWord ()
5. V3PBC_ReadRegDWord ()
6. V3PBC_WriteRegByte ()
7. V3PBC_WriteRegWord ()
8. V3PBC_WriteRegDWord ()
9. V3PBC_ReadSpaceBlock()
10. V3PBC_WriteSpaceBlock()
11. V3PBC_ReadSpaceByte()
12. V3PBC_ReadSpaceWord()
13. V3PBC_ReadSpaceDWord()

14. V3PBC_WriteSpaceByte()

15. V3PBC_WriteSpaceWord()

16. V3PBC_WriteSpaceDWord()

17. V3PBC_ReadBlock()

18. V3PBC_WriteBlock()

19. V3PBC_ReadByte()

20. V3PBC_ReadWord()

21. V3PBC_ReadDWord()

22. V3PBC_WriteByte()

23. V3PBC_WriteWord()

24. V3PBC_WriteDWord()

25. V3PBC_IntIsEnabled()

26. V3PBC_IntEnable()

27. V3PBC_IntDisable()

28. V3PBC_DMAOpen()

29. V3PBC_DMAClose()

30. V3PBC_DMAStart()

31. V3PBC_DMAIsDone()

32. V3PBC_EEPROMInit()

33. V3PBC_EEPROMRead()

34. V3PBC_EEPROMWrite()

35. V3PBC_ReadPCIReg()

36. V3PBC_WritePCIReg()

- Call V3PBC_Close() before end of code.

NOTES:

1. Using one of the sample drivers included with WinDriver as your skeletal code may shorten the development process. (See Section Sample Code that illustrates the sample code).

2. After development of the driver is through, follow the Re-Distribution procedure.

Sample Code

Sample uses of WinDriver and WinDriver for V3 are supplied with the WinDriver toolkit. You may find the WinDriver samples under \windrvr\samples, and the WinDriver for V3 samples under \windrvr\V3.

Each directory contains a files.txt file 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 developer 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 PBCLIB in their project) You may learn to use the WinDriver for V3 API by looking at pbclib.c which is found in \windrvr\V3\pbc_diag.c . pbc_diag.c is the source code of the V3 PBC diagnostics program found under Start \Programs \WinDriver.

WinDriver's V3 PBC API Function Reference

Use this appendix as a quick reference to the WinDrivers V3 PBC functions. This reference may also be found in the WinDriver Help files. Advanced users may find more functionality in the WinDriver API. All the functions outlined in FunctionReference are implemented in the \windrvr\v3\lib\pbclib.c file. The definition of the structures used in the following functions may be found in the V3 PBC structure reference.

V3PBC_CountCards()

V3PBC_Open()

V3PBC_Close()

V3PBC_IsAddrSpaceActive()

V3PBC_GetRevision()

V3PBC_ReadRegByte(), V3PBC_ReadRegWord(), V3PBC_ReadRegDWord()

V3PBC_WriteRegByte(), V3PBC_WriteRegWord(), V3PBC_WriteRegDWord()

V3PBC_ReadSpaceByte(), V3PBC_WriteSpaceByte(), V3PBC_ReadSpaceWord(), V3PBC_WriteSpaceWord(), V3PBC_ReadSpaceDWord(), V3PBC_WriteSpaceDWord()

V3PBC_ReadSpaceBlock(), V3PBC_WriteSpaceBlock()

V3PBC_ReadBlock(), V3PBC_WriteBlock()

V3PBC_ReadByte(), V3PBC_ReadByte(), V3PBC_WriteByte(), V3PBC_ReadWord(), V3PBC_WriteWord(), V3PBC_ReadDWord(), V3PBC_WriteDWord()

V3PBC_IntIsEnabled()

V3PBC_IntEnable()

V3PBC_IntDisable()

V3PBC_ReadPCIReg(), V3PBC_WritePCIReg()

V3PBC_DMAOpen()

V3PBC_DMAClose()

V3PBC_DMAStart()

V3PBC_DMAIsDone()

V3PBC_PulseLocalReset()

V3PBC_EEPROMInit()

V3PBC_EEPROMWrite()

V3PBC_EEPROMRead()

PCI Configuration Registers Definitions

Use for V3PBC_ReadReg() & V3PBC_WriteReg() functions.

1. V3PBC_PCI_VENDOR = 0x00
2. V3PBC_PCI_DEVICE = 0x02
3. V3PBC_PCI_CMD = 0x04
4. V3PBC_PCI_STAT = 0x06
5. V3PBC_PCI_CC_REV = 0x08
6. V3PBC_PCI_HDR_CFG = 0x0c
7. V3PBC_PCI_IO_BASE = 0x10
8. V3PBC_PCI_BASE0 = 0x14
9. V3PBC_PCI_BASE1 = 0x18
10. V3PBC_PCI_SUB_VENDOR = 0x2c
11. V3PBC_PCI_SUB_ID = 0x2e
12. V3PBC_PCI_ROM = 0x30
13. V3PBC_PCI_BPARM = 0x3c
14. V3PBC_PCI_MAP0 = 0x40
15. V3PBC_PCI_MAP1 = 0x44
16. V3PBC_INT_STAT = 0x48
17. V3PBC_INT_CFG = 0x4c
18. V3PBC_LB_BASE0 = 0x54

19. V3PBC_LB_BASE1 = 0x58
20. V3PBC_LB_MAP0 = 0x5e
21. V3PBC_LB_MAP1 = 0x62
22. V3PBC_LB_IO_BASE = 0x6e
23. V3PBC_FIFO_CFG = 0x70
24. V3PBC_FIFO_PRIORITY = 0x72
25. V3PBC_FIFO_STAT = 0x74
26. V3PBC_LB_ISTAT = 0x76
27. V3PBC_LB_IMASK = 0x77
28. V3PBC_SYSTEM = 0x78
29. V3PBC_LB_CFG = 0x7a
30. V3PBC_DMA_PCI_ADDR0 = 0x80
31. V3PBC_DMA_LOCAL_ADDR0 = 0x84
32. V3PBC_DMA_LENGTH0 = 0x88 // mask off upper 8 bits
33. V3PBC_DMA_CSR0 = 0x8b
34. V3PBC_DMA_CTLB_ADR0 = 0x8c
35. V3PBC_DMA_PCI_ADDR1 = 0x90
36. V3PBC_DMA_LOCAL_ADDR1 = 0x94
37. V3PBC_DMA_LENGTH1 = 0x98 // mask off upper 8 bits
38. V3PBC_DMA_CSR1 = 0x9b

39. V3PBC_DMA_CTLB_ADR1 = 0x9c

V3PBC_ADDR

V3PBC_INT_RESULT

Trouble-Shooting

V3PBC_Open() fails

Performing DMA

If you are not using a PCI chip which has enhanced support WinDriver (currently - PLX, Galileo, Altera, V3 or AMCC) these few pages will guide you through the steps of performing 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 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

Contiguous Buffer DMA

Handling Interrupts

Interrupts can easily be handled via DriverWizard. It is recommended that you use DriverWizard to generate the interrupt code for you, by defining (or auto-detecting) your hardware's interrupts, 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

Simplified interrupt handling using *windrvr_int_thread.h*

ISA / EISA and PCI interrupts

Interrupts in Windows CE

PCMCIA interrupts in Windows CE

USB Control Transfers

USB Data Exchange
More about the Control Transfer
The Setup Packet
USB Setup Packet Format
Standard device requests codes
Setup Packet example

Performing Control Transfers with WinDriver

WinDriver allows you to easily send and receive control transfers on Pipe00, while using DriverWizard to test your device and within WinDriver API.

Control Transfers with DriverWizard

Control Transfers with WinDriver API

Improving 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 any 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 into WinDriver's **Kernel PlugIn**.

that the Kernel PlugIn is not implemented under Windows CE and VxWorks since in these OSes there is no separation between kernel mode and user mode. As such, 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 back-seat. 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:

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

	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 'Im performance - Accessing IO ma regions' section later in this Cha this does not solve the problem, the IO at Kernel Mode, by writin PlugIn. (See the Kernel PlugIn r chapters for details)
#2	PCI Card - Accessing an IO mapped range on the card	First, try to change the card fron mapped to memory mapped by bit 0 of the address space PCI configuration register to 0 and th solutions for problem #3. You wi need to re-program the EPROM to initialize BAR0/1/2/3 registers with different values. -I possible, try the solutions sugg problem #1. -If this does not solv problem, handle the IO at Kerne writing a Kernel PlugIn. (See the PlugIn related chapters for det

	Problem	Solution
#3	Accessing a memory mapped range on the card	-Try to access memory without u WD_Transfer() by using direct a memory mapped regions (See t 'Improving Performance -using d access to memory mapped reg section later in this Chapter). -If not solve the problem, then ther hardware design problem. You v able to increase performance b any software design method, or

		writing a Kernel PlugIn, or even a full kernel driver.
#4	Interrupt latency. (Missing interrupts, Receiving interrupts too late)	You need to handle the interrupt Kernel Mode, by writing a kerne (See the Kernel PlugIn related c details)
#5	USB devices: Slow transfer rate	To increase the transfer rate tr increase the packet size by ch different device configuration. If need to do many small transfe 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 I/O mapped regions. The reason is that WinDriver enables the user to directly access the memory mapped regions, without calling the **WD_Transfer()**function.

Using direct access to memory mapped regions

Accessing IO mapped regions

Background

There are some distinct advantages with creating your driver in User Mode.

- No knowledge of the kernel is necessary
  - No knowledge of the MS DDK or kernel debuggers is required
  - Easy development / debugging using standard development / debugging tools.

However, this imposes a fair amount of function call overhead from the Kernel to the User mode and then performance may not be acceptable. In such cases, the Kernel PlugIn feature allows critical section of the driver code to be moved to the kernel while keeping most of the code intact. Using WinDriver's Kernel PlugIn feature, your driver will operate without any degradation in performance.

The advantages of writing a Kernel PlugIn driver over a Kernel Mode driver are:

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

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

3. The parts of the 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 WinDriver's Kernel PlugIn feature, your driver will operate without any degradation.

Do I need to write a Kernel PlugIn?

Not every performance problem requires you to write a Kernel PlugIn. Some performance problems can be solved in the User Mode driver, by better utilization of the features that WinDriver provides.

Chapter Improving Performance guides you in solving your performance issues. This chapter contains a table to help you determine solutions for your driver's performance problems. In some cases a quick User Mode solution is provided, and in other cases, a Kernel Mode PlugIn must be written.

What kind of Performance can I expect

Since you can write your own interrupt handler in the kernel with the WinDriver Kernel PlugIn, you can expect to handle about 100,000 interrupts per sec without missing any one 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 WinDriver's Kernel PlugIn (which runs in the Kernel Mode) thereby eliminating 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 absolutely zero performance degradation.

Introduction

A driver written in the User Mode uses WinDriver's functions(WD_xxx functions) for 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 KernelPlugIn. The code will still work "as is" since WinDriver exposes its WD_xxx interface to the Kernel PlugIn as well.

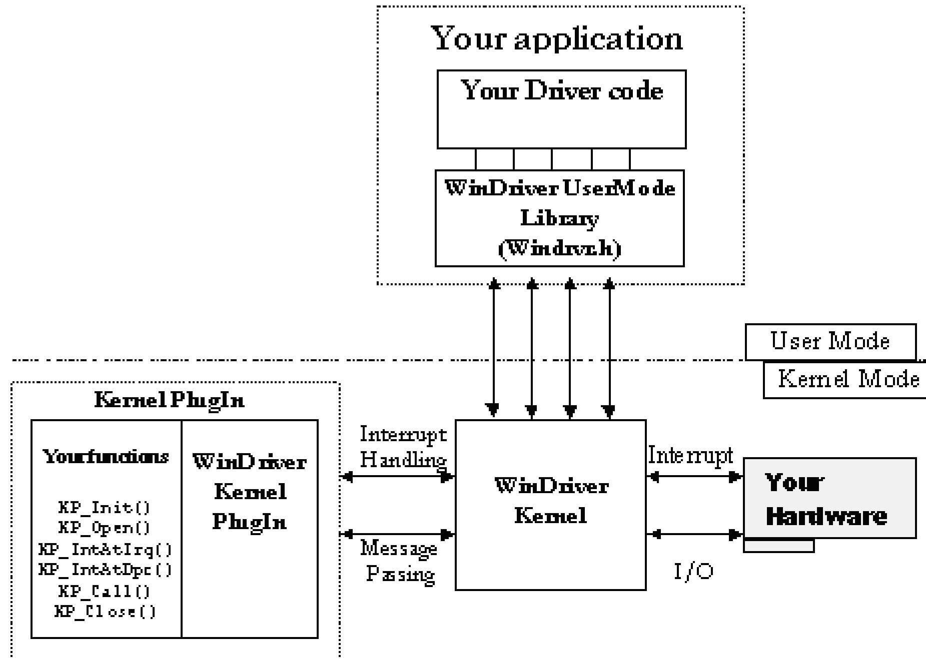


Figure 16.1: KernelPlugIn Architecture

WinDriver Kernel and Kernel Plugin Interaction

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

1. **Interrupt handling**: When WinDriver receives an interrupt, it will activate the interrupt handler in the User Mode by default. However, if the interrupt was set to be handled by the WinDriver Kernel PlugIn, then once WinDriver receives the interrupt, it is processed by the 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.

Kernel Plugin Components

At the end of your Kernel PlugIn development cycle, you 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 wdpnp.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 that you have chosen to bring down to the Kernel level.

Kernel PlugIn Event Sequence

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

Event / Callback	Remarks
Event: Windows loads your Kernel PlugIn driver **KP Call-back:** Your KP_Init() Kernel PlugIn function is called	At boot time, or by dynamic loading, or as instructed by the registry. KP_Init() informs WinDriver of the name of your KP_Open() routine. WinDriver will call this routine when the applicaton 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 names of all the call-back functions that you have implemented in your KP driver, and initiate the KP driver if needed.

Event / Callback	Remarks
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 the 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 a high priority, and therefore should do only 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 / Callback	Remarks
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.

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 ***Win 95/98/ME driver (VXD)*** you do not need the Win 95/98/ME DDK.
  - To create a ***WinNT driver (SYS)*** you need the WinNT DDK, for the following file: ntoskrnl.lib.
  - To compile the Kernel Mode driver on Linux and Solaris, you need gcc, gmake or make.

NOTE: Windows NT/2000 require SYS files for Kernel PlugIn. Windows 95/98/ME need VXD files. SYS files cannot be used on Windows 98/ME as Kernel PlugIn in Version 5.0 and below.

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

Directory structure and sample for the WinDriver Kernel PlugIn

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

\windriver\kerplug\kptest - contains a sample minimal Kernel PlugIn driver.

The sample implements a passing data function to / from the kernel driver and a kernel mode interrupt handler.

The data exchange function gets the version of the WinDriver kernel module and passes it to the user level. This sample can be a base to implement I/O calls with the Kernel PlugIn.

The interrupt handler implements an interrupt counter. The interrupt handler counts five interrupts and notifies the user mode only on one out of every five incoming interrupts.(more details can be found in windriver\kerplug\kptest\files.txt).

KPTest_com.h contains common definitions such as messages, between the Kernel PlugIn and the User-mode.

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

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

Kernel PlugIn implementation

Before you begin
Write your KP_INIT() function
Write your KP_OPEN() function
Write the remaining PlugIn call-backs

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 your skeletal 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.

NOTE: 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.

Handling Interrupts 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 user mode (without Kernel PlugIn)

Interrupt handling in the Kernel (with the Kernel PlugIn)

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 a header file that is common to both the User Mode and Kernel Mode Plugin parts of the driver. This header file is called KPxxx_COM.H by convention -- the corresponding header header file in the KPTest sample is called KPTest_COM.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***. You can see the definition of the message KPTEST_MSG_VERSION in the header file KPTEST_COM.H

The KPTest.EXE sends the message using the ***WD_Kernel PlugInCall()*** function.

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 chapter Improving Performance that explains how to improve the performance of your driver.

Preparing the User Mode source code

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

2. 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).

3. 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

1. Make a copy of the ***KPTest*** directory. For example, to create a new project called ***MyDrv***, copy ***WinDriver\kerplug\KPTest*** to ***\WinDriver\kerplug\MyDrv***.

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

3. Change all occurunces of ***KPTest*** in file names to ***MyDrv***.

Creating a handle to the WinDriver Kernel PlugIn

In your original User Mode source code, call ***WD_Kernel PlugInOpen()*** at the beginning of your code, and ***WD_KernelPlug InClose()*** before terminating.

Interrupt Handling in the Kernel PlugIn

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

2. 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 Section Handling Interrupts in the Kernel PlugIn for an explanation on handling interrupts in the kernel).

I/O handling in the Kernel PlugIn

1. Move your I/O handling code from the User Mode to KP_Call().

2. 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.

3. 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

Run compile.bat to compile and link your KP driver.

- **On Windows platforms:** Run COMPILE.BAT in KERMODE directory. This will create two files: MyDrv.SYS and MyDrv.VXD (in this example, run windriver\kerplug\MyDrv\kermode\compile.bat).
  - **On Linux Platforms:** Run the "make" command in the kermode/LINUX directory.
  - **On Solaris Platform:** Run the "make" command in the kermode/SOLARIS directory.

Installing your Kernel PlugIn Driver

Copy the relevant file to your drivers directory.

On Win32 Platforms:

- **Win95:** Copy the MyDrv.VXD driver that was created to the c:\win95\system\vmm32 directory.
- **WinNT:** Copy the MyDrv.SYS driver that was created to the c:\winnt\system32\drivers directory.
- **Win98:** You can either copy the created VXD driver to the c:\win98\system\vmm32 directory or copy the created SYS driver to the c:\win98\system32\drivers directory.

Register / Load your driver: The Kernel PlugIn driver is dynamically loadable, and therefore you need not re-boot in order to run your driver.

On all Win32 platforms, you need to run WDREG.EXE to install the driver:

(In the following instructions, NAME stands for your Kernel PlugIn driver name).

On Win95 and WinNT:

Run c:\WinDriver\util\WDREG -name NAME install (in this example run WDREG -name mydrv install) to register and load your driver. For further instructions see chapter Dynamically loading your driver that explains how to dynamically load your driver .

On Win98

To install the SYS file, run c:\WinDriver\util\
WDREG -name NAME install(in this example run WDREG -name mydrv install).

To install the VXD file (note the -vxd flag), run c:\WinDriver\util\WDREG -vxd -name NAME install

On Linux

1. Copy the driver created to the modules directory: For example: kptest/LINUX# cp kptest_module.o/lib/modules/misc/
2. Insert the module into the kernel: For example: kptest/LINUX#/sbin/insmod kptest_module.o

On Solaris

1. Copy the driver created to the drivers directory: For example: kptest/SOLARIS# cp kptest/kernel/drv
2. Install the driver: For example: kptest/SOLARIS# add_drv kptest

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()

WD_KernelPlugInClose()

WD_KernelPlugInCall()

WD_IntEnable()

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()

KP_Open()

KP_Close()

KP_Call()

KP_IntEnable()

KP_IntDisable()

KP_IntAtIrql()

KP_IntAtDpc()

User Mode structures

WD_KERNEL_PLUGIN
WD_INTERRUPT
WD_KERNEL_PLUGIN_CALL

Kernel Mode structures

KP_INIT

KP_OPEN_CALL

WD_Open() (or xxx_Open()) fails.

The following may cause WD_Open() to fail:

- ***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/2000/98/ME) or WINDRVR.VXD (for Win95/98/ME) to the device drivers directory. A detailed explanation can be found in Chapter Distributing your Driver that explains how to distribute your driver.

- ***Cause:*** The 30 day evaluation license is over.

  Action: WinDriver will inform you that your evaluation license is over. Please contact Jungo :`sales@jungo.com` to purchase WinDriver.

- ***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 DriverWizard or from the PLX /Galileo /V3 /Altera /AMCC directories), and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

- ***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.

- ***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 I/O, 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 DriverWizard

When a driver already exists in Windows for your device, you must create an .INF file (DriverWizard automates this process) and install it. For exact instructions, see the sections explaining how to create and install .INF files.

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 DriverWizard's "Card Information" screen, the device's physical address is 0x0.), you must create an .inf file (DriverWizard automates this process) and install it. For exact instructions, see the sections explaining how to create and install .INF file.

PCI Card has no resources when using the DriverWizard

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 DriverWizard's "Card Information" screen, a message pops-up notifying that no resources were 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 Chapters The DriverWizard and Distributing your Driver that explain how to create and install a .INF file.

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 is 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 is already done by the WinDriver kernel , 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\Control \SessionManager \MemoryManagement

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.

Windows NT/2000 and 95/98/ME

Dynamic loading - background
Why do you need a dynamically loadable driver?
Dynamically loading and unloading your driver
Dynamically loading your Kernel PlugIn

Linux

To dynamically load WinDriver on Linux, execute:

> /sbin/insmod -f /lib/modules/misc/windrvr.o

To dynamically unload WinDriver, execute:

> /sbin/rmmod windrvr

In addition, you can use the **wdreg** script under Linux to install (load) windrvr.o

Solaris

To dynamically load WinDriver on Solaris, execute:

> /usr/sbin/add_drv -m "* 0666 root sys" windrvr

To dynamically unload WinDriver, execute:

> /usr/sbin/rem_drv windrvr

In addition, you can use the **wdreg** script under Solaris to install (load) windrvr.o

Get a valid license for your WinDriver

To purchase your WinDriver license, fill in your order form found in **\windriver\docs\order.txt**, and fax or email it to Jungo (you can find the full details on the order form itself).

Alternatively, you can order WinDriver on-line. See Jungo's WEB site `http://www.jungo.com`for more details.

When you receive the registered string, you should follow the instructions found under **\windriver\redist\register.txt** to enable your driver to work with the registered version of the WinDriver kernel. This applies for all OS'es.

Windows 95/98/ME and NT/2000

1. Copy VxD or SYS files to the target computer.

In the driver installation script you create, you must copy the following files to the target computer (the system on which you want to install your driver):

- **For Windows NT/2000/98/ME**: Copy WINDRVR.SYS file to WINNT\SYSTEM32\DRIVERS

- **For Windows 95**: Copy WINDRVR.VXD file to \WIN95\SYSTEM\VMM32

  NOTE: WINDRVR.SYS can be used on 98/ME. Due to the limitations of 98/ME, it cannot be loaded dynamically but requires a reboot. If a reboot is not acceptable to you, then use WINDRVR.VXD instead.

USB developers:

- On Windows 2000: Copy Windrvr.sys and wdpnp.sys to WINNT\SYSTEM32\DRIVERS
- On Windows 98/ME: Copy Windrvr.sys and wdpnp.sys to WINDOWS\SYSTEM32\DRIVERS

If you have created a Kernel Plugin

- Copy your Kernel PlugIn driver ([Your driver name].VXD or [Your driver name].SYS) to the relevant directory.

2. Add WinDriver to the list of Device Drivers Windows loads on boot.

This is done by calling ***WDREG 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 programmatically. The utility WDREG.EXE is found it the UTIL subdirectory under the WinDriver installation.

For Windows 98/ME, ***WDREG.EXE install*** installs WINDRVR.SYS and requires a reboot. For a reboot free installation on Windows 98/ME, please install WINDRVR.VXD instead and use the command ***WDREG -vxd install*** to install it.

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 a kernel plugin.

NOTE: When distributing your driver, take care to see that you DO NOT overwrite a newer version of ***windrvr.sys***in the ***windows\system32\drivers***directory or ***windrvr.vxd***in the ***windows\system\vmm32***directory with an older version of the file. You should configure your Install Shield installation program (if you are using Install Shield) or your .INF file such that the installer automatically compares the timestamp on these two files and DOES NOT overwrite a newer version with an older one.

Creating a .INF file

Device information (INF) files are text files, that provide information used by the "Plug and Play" mechanism in Windows ME/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 a .INF file for your device. DriverWizard 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?

How do I install an INF file when no driver exists?

How do I replace an existing driver using the INF file?

Windows CE

Copy WinDriver Kernel DLL file to the 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):

<u>**For Windows CE hand-held computer installations:**</u>

Copy ***WINDRVR.DLL*** file to ***\WINDOWS***on your target Windows CE computer.

<u>**For Windows CE PC:**</u>

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 on a CE PC /ETK installation as described in INSTALLATION AND SETUP.

Add WinDriver to the list of Device Drivers Windows CE loads on boot

For Windows CE hand-held 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 hand-held CE computer or by using the Remote CE Registry Editor Tool supplied with the Windows CE Platform SDK. You will need to have Windows CE Services installed on your Windows 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.

Linux

The Linux kernel is continuously under development, and kernel data structures are subject to frequent change. To support such a dynamic development environment and still have kernel stability, the Linux kernel developers decided that kernel modules must be compiled with the identical header files that the kernel itself was compiled with. They enforce this by *#include*'ing a version number into the kernel header files that is checked against the version number encoded into the kernel. This forces Linux driver developers to facilitate recompilation of their driver based on the target system's kernel version.

WinDriver Kernel Module

Your User Mode Driver

Kernel Plugin Modules

Installation script

Solaris

For Solaris, you need to supply the following items to enable the client to enable target installation of your driver:

- **WinDriver kernel module:** The files windrvr and windrvr.cnf implement the WinDriver kernel module.

- **User mode driver:** The source code or the binaries of your user mode driver.

- **Kernel plugin module:** If you used a kernel plugin module, you should supply the relevant files, for example, mykp and mykp.cnf

Installation script

VxWorks

For VxWorks, you need to supply the following items to enable the client to enable target installation of your driver:

- **WinDriver Kernel:** The file windrvr.o implements the WinDriver kernel module.

- **Your Driver:** The source code or the binaries of your driver, say your_drv.out

The client that you provide these modules to, would want to incorporate all these files into the VxWorks embedded image. There are two steps involved here:

1. windrvr.o and your_drv.out has to be built into the VxWorks image.

   In the Tornado II Project's build specification for the VxWorks image, specify windrvr.o and your_drv.out as EXTRA_MODULES under the MACROS tab, and copy these files under the appropriate target directory tree. Rebuild the project and these files are now included in the image and it should work.

2. During startup, the drvrInit() routine should be called to initialize windrvr.o Your driver's startup routine may also need to be called.

   You have to use the file usrAppInit.c found under the Tornado II project directory and insert code to call drvrInit() -- which is WinDriver's initialization routine -- and your driver applications startup routine. Of course, this means you need to rebuild the VxWorks image.

On Windows 95, 98, ME, NT and 2000

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

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

3. If your code contains performance critical sections, improve their performance by referring to Chapter Improving Performance . 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 to the "Kernel PlugIn".

On Windows CE

1. Plug your hardware into your NT machine.

2. Install the CE ETK on NT.

3. Diagnose your hardware via DriverWizard and then let it generate your driver's skeletal 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.

4. If you cannot plug your hardware into your NT machine, you may still use DriverWizard by manually entering all your resources into it. Let DriverWizard generate your code and then test it on your hardware using ser al 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.

5. If your code contains performance critical sections, improve their performance by referring to the Chapter Improving Performance .

On Linux and Solaris

From version 5.0 and onwards,WinDriver offers a GUI DriverWizard that facilitates Driver Development on Linux and Solaris. Use the GUI DriverWizard for Linux and Solaris in the same way as you use the one on Windows and then generate Linux and Solaris code.

If you are using WinDriver 4.x and below, and you do not use the Linux or Solaris X11 GUI,you may consider Windows as your initial development platform. It is recommended to start the development process on your Windows machine, using DriverWizard in the same way as described above.

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.

On Embedded Operating Systems

For embedded operating systems, like Windows CE or VxWorks, you can use the new Remote WinDriver feature. Just run DriverWizard on a supported Host platform and you can detect and diagnose your hardware on the remote embedded target using the new Remote WinDriver option.

Note: Cross-endian network communication support is not yet provided and therefore both the host and the target machine must have the same endian scheme when using Remote WinDriver. For example, when detecting and diagnosing hardware on a Motorola PPC target architecture, you can use a Sparc machine as a compatible host since both Motorola and Sparc use the same endian scheme.

WinDriver Modules

- WinDriver Version 5 - (***windriver\include***) - The general-purpose hardware access toolkit. The important files here are:

    1. **windrvr.h**: The WinDriver API, data structures and constants are defined in this header file.

    2. **windrvr_int_thread.h**: This is a convenience header file, that contains wrapper functions to simplify interrupt handling.

- DriverWizard (accessible through ***Start Menu | Programs | WinDriver | DriverWizard***) - The graphical tool that diagnoses your hardware and lets you easily code your driver.

- Graphical Debugger (accessible through ***Start Menu | Programs | WinDriver | Monitor Debug Messages***) - A graphical debugging tool which collects debugging information on your driver as it runs. On Linux, Solaris, WinCE and VxWorks, you may use the console version of this program. Starting from version 5.00, the graphical version of the debugger is available on Linux and Solaris also.

- WinDriver distribution package (***windriver\redist***) - The files needed to be included in the driver you distribute to your customers.

- WinDriver Version 5 electronic manual - (accessible through ***Start Menu | Programs | windriver***) - Full WinDriver manual, in pdf (Adobe Acrobat) format, hyperlinked Windows HELP (HLP) format and CHM format for Windows users, and in HTML for Unix users.

- 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.

  NOTE:PCMCIA_SCAN.EXE is found only in the WinDriver CE version.

- ***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 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 9030, 9050, 9052, 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 ALTERA (for Altera PCI cores) -***windriver\altera***
- WinDriver V3 APIs (for the V3 PCI bridges) - ***windriver\v3***

Each of these directories includes the following subdirectories:

- **\lib** - the special chip set API for the PLX/AMCC/V3/ALTERA chip set, written using the WinDriver API.
- **\xxx_diag** - a sample diagnostics application, which was written using the special library functions available for 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.

Monolithic Drivers

These are the device drivers that 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 I / O control commands - (IOCTLs), and drives the hardware through calling the different DDK, ETK, DDI/DKI functions.

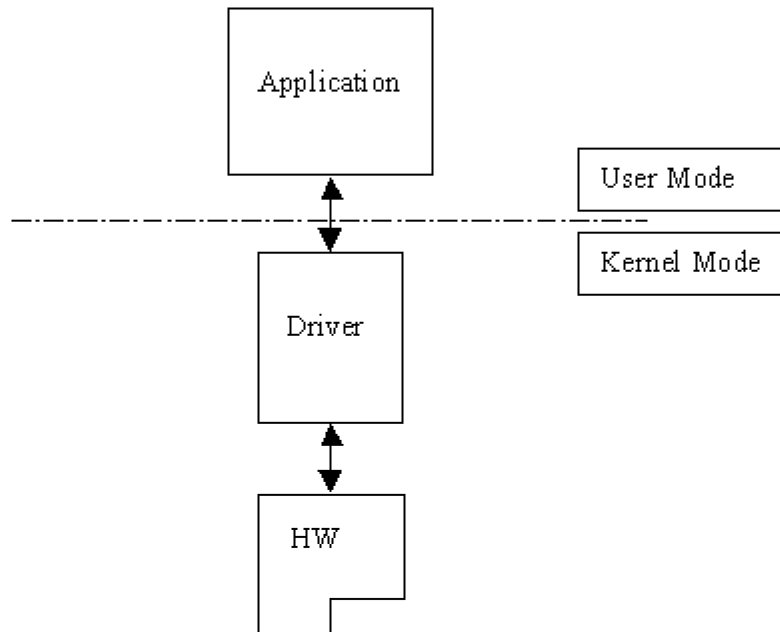


Figure 1.2: Monolithic Drivers

Monolithic drivers are encountered under all operating systems including all Windows platforms (95/98/ME, NT/2000, CE), all Unix platforms (Linux, VxWorks and Solaris), and others like OS/2.

Windows 95/98/ME Drivers

We use the term Windows drivers to mean VxD drivers that run on the related family of OS's, Windows 95, Windows 98 and Windows ME. These drivers do not work on Windows NT. Windows drivers are typically monolithic in nature. They provide direct access to hardware and privileged operating system functions. These drivers are called VxD drivers. Windows drivers can be stacked or layered in any fashion. However, the driver structure itself does not impose any layering.

NT Driver Model

Besides monolithic drivers, Windows NT defines other kinds of drivers that are generally unique to Windows NT, but subsets or minor variations of which, are supported on other Windows operating systems like Windows 95/98/ME and WinCE. These are discussed below.

Layered Drivers
Miniport Drivers

Unix Device Drivers

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

The three classes of devices are:

Character Character *(char)* devices can be accessed as files, and are implemented by char drivers. These drivers usually implement the *open*,*close*, *read*, *write* and *ioctl* system calls. The console and the serial port are examples of devices that are implemented by char drivers. Applications access char devices through files known as device nodes such as*/dev/console* or */dev/ttyS0*.

Block Block devices are also accessed as files, and are implemented by block drivers. Block devices are generally used to represent hardware on which you can implement a file system. Typically, block devices are accessed by multiples of a block of data at a time. Block sizes are typically 512 bytes or 1 Kilobyte (1024 bytes). Block drivers interface with the kernel through a similar interface as a char driver. The device node for a block device shows differently in the filesystem listing.

Network Network interfaces are used to perform network transactions between applications residing on a network. A network interface may work through a hardware device or sometimes be implemented completely in software, like the loopback interface. User applications perform network transactions through interfaces to the kernel network subsystem (usually exposed as API such as sockets and pipes). Network interfaces send and receive network packets on behalf of user applications, without regard to how each individual transaction maps to actual packets being transmitted.

Network interfaces don't easily fit into the block or char philosophy. Hence, they are not visible as device nodes in the filesystem. They are represented by system-wide unique logical names such as *eth0*. Clearly, network interfaces are not accessed via the *open/read/write ...* system calls. Instead they are accessed through network API such as sockets, pipes, RPC etc.

Linux Device Drivers

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

Under Linux, block devices can also be accessed like a character device, but has an additional block oriented interface which is invisible to the user or application.

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

Solaris Device Drivers

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

For Windows 95 / 98 / ME

1. An x86 processor

2. Any 32-bit development environment supporting C, VB or Delphi.

For Windows NT/2000

1. An x86 processor.

2. Any 32-bit 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 hand-held Computer like the HP Jornada or the Sharp Mobilon, you will need the following items in addition to the ones mentioned above:

- Your hand-held computer.

- Serial PC link cable for communication via Windows CE Services (This cable is normally custom manufactured and supplied by the manufacturer of the hand-held 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:

- Your target platform.

- 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 and upgrade your installation before installing WinDriver CE.

- 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.

- A custom parallel port cable for downloading the OS image and dynamic loading of WinDriver CE. Please see Appendix PC-Based Development Platform Parallel Port Cable Info of this manual for the pin-out details of the Parallel Port Cable.

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

For Linux

1. An x86 processor

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

For Solaris

1. An x86 processor

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

For VxWorks

1. Windows NT Workstation 4.0 host platform development

2. Tornado II IDE

3. Selected Target Platform: This should be running a processor that has a BSP (Board Support Package) compatible with the list of CPU/BSP combinations supported by DriverBuilder.

   For an up-to-date list, see the URL below:
   http://www.jungo.com/db-vxworks.html#platforms

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

Installing WinDriver for Windows 95,98,ME,NT and 2000

1. Insert the WinDriver CD into your CD-ROM drive. (When installing WinDriver by downloading it from Jungo's web site instead of using the WinDriver CD, double click the downloaded WinDriver file (**WDxxx.exe**) in your download directory, and go to step 3).

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

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

4. The evaluation package installer proceeds to copy all the files and may prompt you to reboot your system.

   The following steps are For Registered Users only:

5. Choose 'Install registered version' when prompted for the version to install.

6. 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 WinDriver to install. You may choose which APIs will be installed.

7. 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.

8. After completing the set-up, please reboot your computer.

9. Activate DriverWizard from *Start* **|** *Programs* **|***WinDriver* **|** *DriverWizard*. Select the **Register WinDriver** option from the File menu and insert your license string there.

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

Installing WinDriver CE

Installing WinDriver CE for a hand held PC
Using WinDriver CE with ETK/Platform Builder

Testing Your Applications on the X86 HPC Emulator

Emulation on Windows NT

Installing WinDriver for Linux

Since WinDriver installation installs the kernel module windrvr.o, WinDriver should be installed by the system administrator logged in as root, or with root privileges.

- Insert the WinDriver CD into your Linux machine CD drive or copy the downloaded file to your preferred directory (eg. /home/username/).

- Change directory to preferred installation root directory, say **/usr/local** (*cd /usr/local/*)

- Extract the file **WDxxxLN.tgz** (where xxx is the version number) (*/usr/local> tar xvzf /mnt/cdrom/LINUX/WDxxxLN.tgz*) From a CD: (*/usr/local> tar xvzf /mnt/cdrom/LINUX/WDxxxLN.tgz*)

  From a downloaded file: (*/usr/local> tar xvzf /home/username /WDxxxLN.tgz*)

- Change directory to Windriver (this directory gets created by tar) (*/usr/local> cd WinDriver/*)

  Note: In V5.0, this directory gets created by tar, but in versions preceding 5.0, the WinDriver directory does not get created by the extraction. Therefore with older versions like 4.33, first create a directory (say WinDriver) before proceeding with the installation. (*/>mkdir /usr/local/WinDriver*)

- Install WinDriver (*/usr/local/WinDriver> make install*)

- Create a symbolic link so that you can easily launch the GUI DriverWizard (*/usr/local/WinDriver>ln -s /usr/local/WinDriver/wizard/wdwizard /usr/bin/wdwizard*).

- Change the read and execute permissions on the file wdwizard so that ordinary users can access this program

- Change the user and group ids and give read/write permissions to the device file */dev/windrvr* depending on how you wish to allow users to access hardware through the device.

- Start accessing your hardware and generate your skeletal code!

The following steps are for Registered Users only

1. Change directory to **/windriver/redist/register/** (*/usr/local/WinDriver> cd redist/register*)

2. Extract the file wdxxxreg.zip using the password you have received with the WinDriver package (*/usr/local/WinDriver/redist/register> ../../util/unzip WDxxxREG.ZIP*)

3. Change directory to **windriver/redist/** (*/usr/local/WinDriver/redist/register/>cd ..*)

4. Remove the evaluation module if previously installed (*/usr/local/WinDriver/redist/> /sbin/rmmod windrvr*)

5. Clean the evaluation version module directory (*/usr/local/WinDriver/redist/> make clean*)

6. Install registered version (*/usr/local/WinDriver/redist/> make install IS_REGISTERED=1*)

   Once you activate DriverWizard you will prompted for the license string you have received when purchasing the registered version.

7. To activate source code you have developed in the evaluation version, simply follow the instructions in *WinDriver/redist/register/register.txt*.

Restricting hardware access on Linux

Installing 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

Since WinDriver installation installs the kernel module windrvr.o, it should be installed by the system administrator logged in as root, or with root privileges.

1. Insert your CD into your Solaris machine CD drive or copy the downloaded file to your preferred directory (eg. /home/username/).

2. Change directory to preferred installation root directory, say **/usr/local**: (*cd /usr/local/*)

3. Copy the file WDxxxSLS.tgz (Sparc) or WDxxxSL.tgz (Intel) to the current directory (here xxx stands for the version number, for example 500): (*/usr/local/> cp /home/username /WDxxxSL.tgz ./*)

   Note: When installing WinDriver for Solaris x86 use WDxxxSL.tgz instead of WDxxxSLS.tgz.

4. Extract the file (*/usr/local/> gunzip -c WDxxxSLS.tgz* | *tar -xvf WDxxxSLS.tar*)

5. Change directory to WinDriver (**Note**: In V5.0 this directory gets created by tar but in versions preceding 5.0, the WinDriver directory does not get created by the extraction. Therefore with older versions like 4.3, first create a directory (say WinDriver) before proceeding with the installation.)

6. Install WinDriver for Solaris (*/usr/local/WinDriver>./ install_windrvr*)

7. Create a symbolic link so that you can easily launch the GUI DriverWizard (*/usr/local/WinDriver>ln -s /usr/local/WinDriver/wizard/wdwizard /usr/bin/wdwizard*).

8. Change the read and execute permissions on the file wdwizard so that ordinary users can access this program

9. Change the user and group ids and give read/write permissions to the device file */dev/windrvr* depending on how you wish to allow users to access hardware through the device.

The following steps are for Registered Users only:

- Change directory to **WinDriver/redist/register/**: (*/usr/local/WinDriver> cd redist/register*)

- Extract the file found in **WDxxxREG.ZIP** and enter the 8 digit password you have received with the WinDriver package: (*/usr/local/WinDriver/redist/register> unzip WDxxxREG.ZIP*)

- Remove the evaluation kernel with the command */usr/sbin/rem_drv windrvr*

- Replace the evaluation WinDriver kernel (windrvr) with the registered version you have extracted above: (*cp WinDriver/redist/register/windrvr /kernel/drv*)

- Install the registered module by running the command
  /usr/sbin/add_drv -m " 0666 root sys" windrvr*

  Note: after extracting the registered kernel in step 2 you can modify the install_windrvr (used to install the evaluation kernel) script in top dir (WinDriver) to install the registered kernel.For this you need to perform the following change: Open the file install_windrvr and look for the following line: (*util/wdreg redist/eval/windrvr util/windrvr.conf*)
  change it to: (*util/wdreg redist/register/windrvr util/windrvr.conf*)

- To create symbolic links from the /dev dir to the actual nodes in the /devices dir, run the command */usr/sbin/devlinks*

- To activate source code you have developed in the evaluation version, simply follow the instructions in *WinDriver/redist/register/register.txt*

Restricting hardware access on Solaris
Solaris Platform Specific Issues

Installing DriverBuilder for VxWorks

The following describes the installation of DriverBuilder for VxWorks. DriverBuilder development environment works with Tornado 2 for Windows only (on x86 platform). Drivers generated using version 5.0 of DriverBuilder will run on Intel x86 BSPs (pc486, pcPentium and pcPentiumPro), PPC 821/860 with MBX821/860 and PPC 750 (IBM PPC 604) with MCP750. For an up-to-date list, see the URL below: *http://www.jungo.com/db-vxworks.html#platforms*

To install DriverBuilder:

1. Download DriverBuilder for VxWorks

2. Change drive to the preferred root drive for DriverBuilder, for example "c:\"

3. Unpack the file you downloaded - for example: "unzip -d DB500VX.zip c:\". The extraction creates a directory called DriverBuilder under which all the DriverBuilder installation files can be found.(this feature was added in version 5.00. If you are working on a previous version,please create a directory for DriverBuilder,for example: "c:\cd_vxworks" and unpack the file to it: "unzip -d DBxxxVX.zip c:\db_vxworks").

4. Create a shortcut on your desktop to DriverWizard found under C:\DriverBuilder\wizard\ wdwizard.exe so that you can easily launch the GUI DriverWizard

Note: The WinDriver samples for VxWorks have the .out extension. For example, *pci_diag.out*. To invoke these programs, use Windsh to load them, and execute the routine xxx_main. For example:

wddebug.out : *wddebug_main*

pci_diag.out : *pci_diag_main*

On your Windows machine

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

Registered Users

- Make sure that your WinDriver license is installed (see Section Installing WinDriver that explains how to install WinDriver). If you are an evaluation version user, you do not need to install a license.

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

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

On your Windows CE machine

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

- Make sure that your WinDriver license is installed (see Section Installing WinDriver for WinDriver installation details). If you are an evaluation version user, you do not need to install a license.

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

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

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

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

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

On your Linux machine

- Run the pre-compiled speaker sample found in WinDriver/samples/speaker/LINUX/speaker

- If the sample program works, then you have installed WinDriver for Linux properly.

On your Solaris machine

- Run the precompiled speaker sample found in WinDriver/samples/speaker/Solaris/speaker

- If the sample program works, then you have installed WinDriver for Solaris properly (this program only works under X86). For Sparc Solaris you can run the GUI DriverWizard to check the installation.

On VxWorks

1. In x86 only: Make sure MMU is set to basic support ("hardware/memory/MMU/MMU Mode"). DriverBuilder for "no MMU" support will be available shortly.

2. Load DriverBuilder: download the object file (DriverBuilder \redist\eval\intelx86\PENTIUM\windrvr.o).

3. Initialize DriverBuilder: from the WindShell:

```
=> drvrInit()

function returned (return value = 0)

 =>
```

4. Run a sample driver: load C:\DriverBuilder \samples\pci_diag\PENTIUM\pci_diag.out from the WindShell:

```
=> pci_diag_main()
```

now you can scan the PCI bus, open cards and 'talk' to them.

Uninstalling Windriver from Windows (95/98/ME/NT/2000)

1. Uninstall the Windriver service using the command
   windriver\util\wdreg remove

2. Use the "Add/Remove Programs" applet from the Control Panel, select Windriver, and click on the "Remove" button.

3. Delete the following files if they exist:

   - Windows NT/2000: Windrvr.sys and wdpnp.sys from Winnt\system32\drivers

   - Windows 98: Windrvr.sys and wdpnp.sys from Windows\system32\drivers

   - Windows 95/98/ME: windrvr.vxd from Windows\System\Vmm32

Uninstalling Windriver from Linux

NOTE: You must be logged in as root to do the uninstallation.

1. Uninstall the Windriver service

    - Do a /sbin/lsmod to check if the WinDriver module is in use by any application or by other modules

    - Make sure that no programs are using WinDriver

    - if any application or module is using WinDriver, close all applications and do a /sbin/rmmod to remove any module using WinDriver

    - Run the command "/sbin/rmmod windrvr"

    - rm -rf /dev/windrvr (Remove the old device node in the /dev directory)

    - Remove the file .windriver.rc in the /etc directory.
      Run the command rm -rf /etc/.windriver.rc to do this.

    - Remove the file .windriver.rc in $HOME.
      Run the command rm -rf $HOME/.windriver.rc to do this.

2. If you created a symbolic link to DriverWizard, delete the link using the command "rm -f /usr/bin/wdwizard"

3. Delete the Windriver installation directory. Use the command "rm -rf /usr/local/WinDriver"

Uninstalling WinDriver from Solaris

1. Uninstall the WinDriver service

   - You must be root to do the uninstallation
   - Make sure no programs are using WinDriver
   - If any applications or modules are usng WinDriver, then close them and do a /usr/sbin/rem_drv to remove any modules using WinDriver
   - Run the command "/usr/bin/rem_drv windrvr" to unload the kernel module
   - Run the command
     "rm -rf /kernel/drv/windrvr /kernel/drv/windrvr.conf"
     to clean up the old device node.
   - Remove the file .windriver.rc in the /etc directory.
     Run the command rm -rf /etc/.windriver.rc to do this.
   - Remove the file .windriver.rc in $HOME.
     Run the command rm -rf $HOME/.windriver.rc to do this.

2. If you created a symbolic link to DriverWizard, delete the link using the command "rm -f /usr/bin/wdwizard"

3. Delete the WinDriver installation directory. Use the command "rm -rf /usr/local/WinDriver"

Uninstalling DriverBuilder for Vxworks

1. Delete the DriverBuilder installation directory (for example C:\DriverBuilder) using Windows Explorer

2. If you created any shortcuts to DriverWizard on your desktop, delete the shortcut.

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 a newly defined interrupt is 'shared'. If you wish to define it as an 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.

To disable a resource

1. Select the resource.

2. 'Right - click' on the resource name.

3. Choose '**Disable**' from the menu.

DriverWizard Logger

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. 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 and 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 diagnostics program. Here you can find all your hardware specific API created by 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 DriverWizard. Use this diagnostics program as your skeletal device driver.

   pci_diag_lib.c This is the source code of the diagnostics program DriverWizard creates.

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

After creating your code, compile it with your favorite Win32 compiler, and see it work!

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

Step 2 - Compiling the generated code

For Windows 95, 98, ME, 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 code generation, the chosen IDE (Integrated development environment) will be launched automatically. You can immediately compile and run the generated code.

For Linux and Solaris

DriverWizard creates a makefile for your project.

Compile the source code using the makefile generated by DriverWizard.

Use GCC to build your code.

For Other OSs or IDEs

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

Include the source files created by 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.

How does it work?

Remote WinDriver utilizes a TCP/IP connection to communicate between the host development system and a target development system to which the hardware is plugged. On the remote target machine, WinDriver Remote Agent is installed together with the WinDriver Kernel Module, and permits access to the hardware directly from the user level in the local host development machine.

Using Remote WinDriver (For all Windows OSes)

- Install Remote WinDriver Agent (wdremote_gui.exe) on the remote machine (target). This file is in the directory WinDriver\util

- Run DriverWizard (wdwizard.exe) on the local machine. DriverWizard is in the directory WinDriver\wizard

- Open a new project from DriverWizard

- Click on the "Remote Machine" button in the Card Information screen

- The following screen is then displayed:

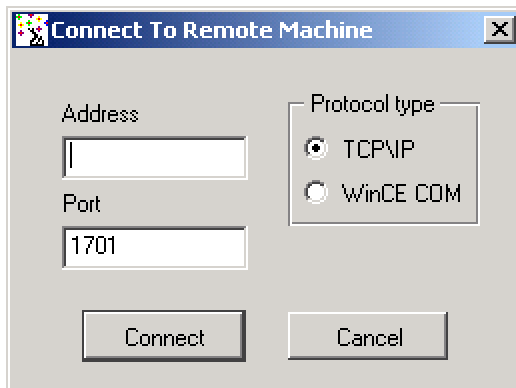


Figure 4.8: Remote WinDriver on all Windows OS

- Enter the IP address of the remote machine (on which wdremote_gui.exe runs)

- Select the protocol type as TCP/IP

- Enter the Port number to listen to in the Port text box. The default port number is 1701

- DriverWizard will scan the buses of the remote machine and display the Plug and Play cards plugged into the remote machine.

- Select your hardware and click on the OK button

- Test and diagnose your hardware on the remote machine and generate the driver code as with any other WinDriver product. For more information refer to Chapters The DriverWizard and Creating Your Driver . These illustrate the process of developing a device driver.

Notes: By default, wdremote_gui.exe listens to the TCP port number 1701 for incoming remote connections. If this port number is already used on your system, you can change the port number to a different one that you know is unused. If you do so, then on the host-side, you need to specify the same port number when you attempt to connect using DriverWizard.

There are two ways of running this program wdremote_gui.exe:

1. Just run it from the command prompt as wdremote_gui.exe

2. Launch the program from Programs | WinDriver | WinDriver Remote Server

Using Remote WinDriver (For Windows CE)

Remote WinDriver configuration for Windows CE is different from other OSes. The following steps explain the Remote WinDriver Setup and use for Win CE.

1. Copy wdremote.dll (in \WinDriver\util) to the windows directory of your hand-held PC. If your WinCE target is a normal PC, then copy wdremote.dll to the release directory of WinCE in your NT workstation

2. Establish a serial PC link between the host and the target using Microsoft Active Sync

3. Run DriverWizard on the NT workstation

4. Open a new project

5. DriverWizard will display all the hardware components in the local machine

6. Click the "Remote Machine" button

7. The following screen is then displayed:

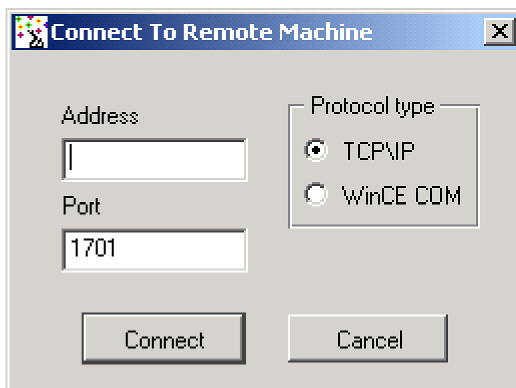


Figure 4.9: Remote WinDriver on Windows CE

8. Select the Protocol Type as Wince COM

9. Click the Connect button

10. After a few seconds, the hardware in the Remote Machine is displayed. If you are using an eval version , then as soon as the remote connection is established, a screen pops-up in the target machine informing you that the WinDriver kernel will work only for 30 mins. After you click OK in this screen, DriverWizard will display the list of devices on the target platform.

    In the eval versions, the remote agent in the CE machine will stop working after 10 mins.

11. Select your hardware and click OK

12. Test, diagnose and generate code for your driver as with any other WinDriver product.

For more information, refer to Chapters The DriverWizard and Creating Your Driver that illustrate the process of developing a device driver.

Using Remote WinDriver (For Linux and Solaris)

- On the Target machine(contains the hardware for which you are writing the driver), do the following:

  1. Change your directory to WinDriver/util

  2. Start the wdremote server utility by running the command **wdremote**

     NOTE: The options for the wdremote are the TCP/IP port number you wish use to communicate with the client. The default port number is 1701.

- On the Client machine, do the following:

  1. Start the wizard in the directory WinDriver/wizard

  2. Open a new project

  3. Click on the "Remote Machine" button

     The following screen is then displayed:

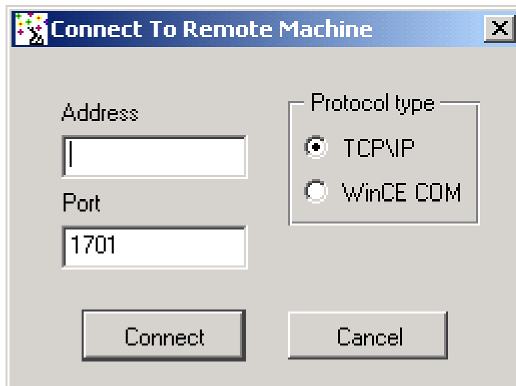


Figure 4.10: Remote WinDriver on Linux

  4. Enter the IP address of the machine which is running the wdremote server (your target machine)

5. Enter the TCP/IP port number you wish to use to communicate with the server. The default port number is 1701

   Note: Use the same port number you used to start your server.

6. Click OK

   NOTE: This takes a while, so please wait for a while for the wizard to display the information of all the cards on your target machine

7. Test and diagnose your hardware on the remote machine and generate the driver code.

Note: wdremote is a command line program that takes one option - the TCP port number to listen on. If no option is given, the port number defaults to 1701.

Just run this program as wdremote from the command line. Call this program from /etc/rc.d/rc.local/ to have it started automatically at boot time.

Using Remote WinDriver (For VxWorks)

- Run Remote WinDriver Agent *(wdremote.out)* on the remote machine (target). This file is in the directory DriverBuilder\util\<CPU>, where <CPU> stands for the BSP you are using. For example, *DriverBuilder\util\Pentium\wdremote.out*. You need to use the **Tornado II IDE** utility *Windsh* to load this program and launch it, by using the entry point *wdremote_main*

- Run DriverWizard (wdwizard) on the local machine (Windows, Linux or Solaris). DriverWizard is in the directory WinDriver\wizard or DriverBuilder\wizard.

- Open a new project from DriverWizard

- Click on the "Remote Machine" button in the Card Information screen

- Enter the IP address of the remote machine (on which wdremote_main runs)

- Select the protocol type as TCP/IP

- Enter the Port number to listen to in the Port text box. The default port number is 1701

- DriverWizard scans the buses of the remote machine and display the Plug and Play cards plugged into the remote machine.

- Select your hardware and click on the OK button

- Test and diagnose your hardware on the remote machine and generate the driver code.

For more information, refer to Chapters The DriverWizard and Creating Your Driver that illustrate the process of developing a device driver.

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
DebugMonitor - Console Mode
DebugMonitor on Linux and Solaris
DebugMonitor on Windows CE
DebugMonitor on VxWorks

Introduction

The PLX9050 Diagnostics (P50_DIAG.EXE) accesses the hardware using WinDriver. Therefore WinDriver must be installed before being able to run P50_DIAG. Once WinDriver is running, you may run P50_DIAG.

Click 'Start / Programs / WinDriver / PLX 9050 Diagnostics'. The application will first try to locate the card, with the default VendorID and DeviceID assigned by PLX (VendorID = 0x10b5, DeviceID = 0x9050). If such a card is found you will get a message "PLX 9050 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 PLX 9050 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 PLX 9050 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.

PLX 9050 local registers: This option is available only after choosing an active card. A list of the PLX9050 registers and their READ value are displayed. In order to WRITE to a PLX9050 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 the active mode between BYTE/WORD/DWORD, to fit the hardware you are accessing. The PLX9050 has four memory ranges to access a local region. There is no difference between them, therefore you need not change the active memory range.

- 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.

  LASxBA register is set with the base address, and LASxBRD register is set with the mode (BYTE/WORD/DWORD as chosen as active mode).

Sample

Code Sample uses of WinDriver and WinDriver for PLX are supplied with the WinDriver toolkit. You may find the WinDriver samples under *\windrvr\samples*,and the WinDriver for PLX samples under *\windrvr\plx*.

Each directory contains a files.txt file 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 Developer 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 p50_lib in their project) You may learn to use the WinDriver for PLX API by looking at P50_diag.c which is found in \windrvr\plx\P50_diag.c . P50_diag.c is the source code of the PLX 9050 diagnostics program found under Start \ Programs \ WinDriver.

P9050_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 P9050_Open, to choose which board to open. Normally, only one board is in the bus and this function will return 1.

Prototype

DWORD P9050_CountCards(DWORD dwVendorID, DWORD dwDeviceID);

Parameters

1. dwVendorID - Vendor ID of your PLX 9050 card. If 0, detects cards from all vendors.

2. dwDeviceID - Device ID of your PLX 9050 card. If 0, detects all devices.

Return Value

Returns the number of matching PCI cards found.

Example

```
nCards = P9050_CountCards( 0x10b5, 0x9050 );
```

P9050_Open()

Used to open a handle to the PLX 9050 card. If several 9050 cards are installed, the specific card to open may be specified by using P9050_CountCards before using P9050_Open, and calling open with a specific card number (See prototype below) If open is successful function returns TRUE, and a handle to the PLX card.

Prototype

BOOL P9050_Open (P9050HANDLE *phPlx, DWORD dwVendorID, DWORD dwDeviceID, WORD nCardNum, DWORD options);

Parameters

1. dwVendorID, dwDeviceID - DeviceID / VendorID of the board to access.

2. nCardNum - If there are a few cards with the given VendorID and DeviceID, this will choose what one of them to open. This number should be from 0 (the first board) up to the value returned by P9050_CountCards() minus one. Most programs that do not need to handle multiple boards, can set nCardNum to 0.

3. options - options regarding the board opened. If P9050_OPEN_USE_INT, then the board must have an interrupt, and the interrupt will be installed. If P9050_OPEN_FIX_BIT7, then a workaround will be applied on the errata on some PLX chips regarding BIT7 of BAR0.

Return Value

TRUE if OK.

Example

```
if (!P9050_Open( &hPlx, 0x10b5, 0x9050, 0, P9050_OPEN_USE_INT ))

{

  printf("Error opening device\n");
}
```

P9050_Close()

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

Prototype

void P9050_Close(P9050HANDLE hPlx);

Parameters

hPlx - handle to the PLX card to close.

Return Value

none

Example

```
P9050_Close(hPlx);
```

P9050_IsAddrSpaceActive()

Checks if the specified address space is enabled. The enabled address spaces are determined by the EEPROM, which at boot time sets the PLX's memory ranges requests. Use this function after calling P9050_Open() to make sure that the address space(s) that your driver is going to use are enabled.

Prototype

BOOL P9050_IsAddrSpaceActive(P9050HANDLE hPlx, P9050_ADDR addrSpace);

Parameters

1. hPlx - PLX card handle

2. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM.

Return Value

TRUE if address space is enabled

Example

```
if ( !P9050_IsAddrSpaceActive(hPlx, P9050_ADDR_SPACE2) )

{

  printf ("Address space2 is not active!\n");

}
```

P9050_ReadReg(), P9050_WriteReg()

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

Prototype

DWORD P9050_ReadReg (P9050HANDLE hPlx, DWORD dwReg);

void P9050_WriteReg (P9050HANDLE hPlx, DWORD dwReg, DWORD dwData);

Parameters

1. hPlx - PLX card handle.

2. dwReg - offset of register to read or write. The constant values of registers are already defined in P50LIB.H, for example: P9050_LAS0RR is the offset of LAS0RR register.

3. dwData - data to write to register. (for P9050_WriteReg() only).

Return Value

Data read from register (for P9050_ReadReg() only).

P9050_ReadSpaceByte(), P9050_ReadSpaceWord(), P9050_ReadSpaceDWord()

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

Prototype

1. BYTE P9050_ReadSpaceByte (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);

2. WORD P9050_ReadSpaceWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);

3. DWORD P9050_ReadSpaceDWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Choose the address space to access.

3. dwOffset - offset in the address space to read from.

Return Value

Data read from the board.

P9050_WriteSpaceByte(), P9050_WriteSpaceWord(), P9050_WriteSpaceDWord()

Writes a byte / word / dword to address space on board.

Prototype

1. void P9050_WriteSpaceByte (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, BYTE data);

2. void P9050_WriteSpaceWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, WORD data);

3. void P9050_WriteSpaceDWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, DWORD data);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Choose the address space to access.

3. dwOffset - offset in the address space to write to.

Return Value

None

P9050_ReadSpaceBlock(), P9050_WriteSpaceBlock()

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

Prototype

1. void P9050_ReadSpaceBlock (P9050HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);

2. void P9050_WriteSpaceBlock (P9050HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);

Parameters

1. hPlx - PLX card handle.

2. dwOffset - offset in the address space to read from or write to.

3. buf - a caller given buffer, into which data is read from or written to.

4. dwBytes - size to read in bytes

5. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Choose the address space to access.

6. mode - access mode: P9050_MODE_BYTE/WORD/DWORD.

Return Value

None

P9050_ReadByte(),P9050_ReadWord()P9050_ReadDWord()

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

Prototype

1. BYTE P9050_ReadByte (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr);

2. WORD P9050_ReadWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr);

3. DWORD P9050_ReadDWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space used to access the local address. If accesses to card are not done from two threads simultaneously (this is the normal case), then all accesses can use the same address space.

3. dwLocalAddr - local address on board to read from.

Return Value

Data read from board.

P9050_WriteByte(),P9050_WriteWord(),P9050_WriteDWord()

Writes a byte / word / dword to memory on board.

Prototype

1. void P9050_WriteByte (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr, BYTE data);

2. void P9050_WriteWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr, WORD data);

3. void P9050_WriteDWord (P9050HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwLocalAddr, DWORD data);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space used to access the local address. If accesses to card are not done from two threads simultaneously (this is the normal case), then all accesses can use the same address space.

3. dwLocalAddr - local address on board to write to.

4. data - data to write to board.

Return Value

None

P9050_ReadBlock(),P9050_WriteBlock()

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

Prototype

1. void P9050_ReadBlock (P9050HANDLE hPlx, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);

2. void P9050_WriteBlock (P9050HANDLE hPlx, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);

Parameters

1. hPlx - PLX card handle.

2. dwLocalAddr - local address on board to read from or write to.

3. buf - a caller given buffer, into which data is read from or written to.

4. dwBytes - size to read in bytes.

5. addrSpace - Can be P9050_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space used to access the local address. If accesses to card are not done from two threads simultaneously (this is the normal case), then all accesses can use the same address space.

6. mode - local address access mode: P9050_MODE_BYTE/WORD/DWORD.

Return Value

None

P9050_IntIsEnabled()

Checks whether interrupts are enabled or not.

Prototype

BOOL P9050_IntIsEnabled(P9050HANDLE hPlx);

Parameters

hPlx - PLX card handle.

Return Value

TRUE if interrupts are already enabled (i.e. if P9050_IntEnable() was called).

P9050_IntEnable()

Enable interrupt processing.

IMPORTANT NOTE*: The PLX 9050 uses level sensitive interrupts, therefore you must edit the implementation of this function (found in \windrvr\plx\lib\9050 \p50lib.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 'WinDriver'manual, in the section regarding PCI interrupts implementation.*

Prototype

BOOL P9050_IntEnable(P9050HANDLE hPlx, P9050_INT_HANDLER funcIntHandler);

Parameters

1. hPlx - PLX card handle.

2. FuncIntHandler - callback interrupt handler. Upon each incoming interrupt, this function will be called.

Return Value

TRUE if successful.

P9050_IntDisable()

Disable interrupt processing.

Prototype

void P9050_IntDisable(P9050HANDLE hPlx);

Parameters

hPlx - PLX card handle.

Return Value

None

P9050_EEPROMReadWord(), P9050_EEPROMWriteWord()

Read from or write to the serial EEPROM.

Prototype

1. BOOL P9050_EEPROMReadWord (P9050HANDLE hPlx, DWORD dwAddr, PWORD pwData);

2. BOOL P9050_EEPROMWriteWord (P9050HANDLE hPlx, DWORD dwAddr, WORD wData);

Parameters

1. hPlx - PLX card handle.

2. dwAddr - address to read or write, ranges 0-0x3f.

3. pwData - pointer to data to read (P9050_EEPROMReadWord() only).

4. wData - data to write (P9050_EEPROMWriteWord() only).

Return Value

TRUE if successful.

P9050_ReadPCIReg(),P9050_WritePCIReg()

Read from or write to the PCI configuration registers.

Prototype

1. DWORD P9050_ReadPCIReg (P9050HANDLE hPlx, DWORD dwReg);

2. void P9050_WritePCIReg (P9050HANDLE hPlx, DWORD dwReg, DWORD dwData);

Parameters

1. hPlx - PLX card handle.

2. dwReg - offset of PCI configuration register to read or write.

3. dwData - data to write (P9050_WritePCIReg() only).

Return Value

Data read from configuration register (for P9050_ReadPCIReg only).

PLX Register Definitions

Use these definitions for accessing the PLX 9050 specific registers.
Use when calling P9050_ReadReg() and P9050_WriteReg().

1. P9050_LAS0RR = 0x00
2. P9050_LAS1RR = 0x04
3. P9050_LAS2RR = 0x08
4. P9050_LAS3RR = 0x0c
5. P9050_EROMRR = 0x10
6. P9050_LAS0BA = 0x14
7. P9050_LAS1BA = 0x18
8. P9050_LAS2BA = 0x1c
9. P9050_LAS3BA = 0x20
10. P9050_EROMBA = 0x24
11. P9050_LAS0BRD = 0x28
12. P9050_LAS1BRD = 0x2c

13. P9050_LAS2BRD = 0x30

14. P9050_LAS3BRD = 0x34

15. P9050_EROMBRD = 0x38

16. P9050_CS0BASE = 0x3c

17. P9050_CS1BASE = 0x40

18. P9050_CS2BASE = 0x44

19. P9050_CS3BASE = 0x48

20. P9050_INTCSR = 0x4c

21. P9050_CNTRL = 0x50

P9050_MODE

Used for P9050_ReadBlock() and P9050_WriteBlock

1. P9050_MODE_BYTE = 0

2. P9050_MODE_WORD = 1

3. P9050_MODE_DWORD = 2

P9050_ADDR

Used for P9050_ReadBlock() and P9050_WriteBlock().

1. P9050_ADDR_SPACE0 = 1
2. P9050_ADDR_SPACE1 = 2
3. P9050_ADDR_SPACE2 = 3
4. P9050_ADDR_SPACE3 = 4
5. P9050_ADDR_EPROM = 5

P9050_INT_RESULT

Used for the interrupt handler callback, given to P9050_IntEnable() to receive information on the interrupt.

Members

TYPE	NAME	DESCRIPTION
DWORD	dwCounter	number of interrupts received
DWORD	dwLost	number of interrupts not yet dea
BOOL	fStopped	true if interrupt was disabled dur
DWORD	dwIntStatusReg	value of status register when int occurred

P9050_Open() fails

The following may cause P9050_Open() to fail:

Cause: WinDriver's kernel is not loaded.

Action: Run WDREG.EXE install (in the \windrvr\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.

Cause: The 30 day evaluation license is over.

Action: In this case, WinDriver will inform you that your evaluation license is over, in a message box. Please contact sales@jungo.com to purchase WinDriver.

Cause: The VendorID / DeviceID requested in P9050_Open() do not match that of the board. (In licensed versions).

Action: Run P50_DIAG.EXE and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

Cause: The PLX device is not installed or configured correctly.

Action: Run P50_DIAG.EXE (found in your \windrvr\util directory), and choose PCI scan. Check that your PLX device returns all the resources needed.

Cause: Your PLX device is in use by another application.

Action: Close all other applications that might be using the PLX device.

Introduction

The PLX9060 Diagnostics (P60_DIAG.EXE) accesses the hardware using WinDriver. Therefore, WinDriver must be installed before being able to run P60_DIAG. Once WinDriver is running, you may run P60_DIAG.

Click 'Start / Programs / WinDriver / PLX 9060 Diagnostics'. The application will first try to locate the card, with the default VendorID and DeviceID assigned by PLX (VendorID = 0x10b5, DeviceID = 0x9060). If such a card is found you will get a message "PLX 9060 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 PLX 9060 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 PLX 9060 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.

PLX 9060 local registers: This option is available only after choosing an active card. A list of the PLX9060 registers and their READ value are displayed. In order to WRITE to a PLX9060 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. REMAP_PTOL_0 register is set with the base address, and REGION_DESC register is set with the mode (BYTE/WORD/DWORD as chosen as active mode).

Sample Code

Sample uses of WinDriver and WinDriver for PLX are supplied with the WinDriver toolkit. You may find the WinDriver samples under \windrvr\samples, and the WinDriver for PLX samples under \windrvr\plx.

Each directory contains a files.txt file 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 developer 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 p60_lib in their project) You may learn to use the WinDriver for PLX API by looking at P60_diag.c which is found in \windrvr\plx\P60_diag.c . P60_diag.c is the source code of the PLX 9060 diagnostics program found under Start \ Programs \ WinDriver.

P9060_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 P9060_Open, to choose which board to open. Normally, only one board is in the bus and this function will return 1.

Prototype

DWORD P9060_CountCards(DWORD dwVendorID, DWORD dwDeviceID);

Parameters

1. dwVendorID - Vendor ID of your PLX 9060/9080 card. If 0, detects cards from all vendors.

2. dwDeviceID - Device ID of your PLX 9060/9080 card. If 0, detects all devices.

Return Value

Returns the number of matching PCI cards found.

Example

```
nCards = P9060_CountCards( 0x10b5, 0x9060 );
```

P9060_Open()

Used to open a handle to the PLX 9060 card. If several 9060 cards are installed, the specific card to open may be specified by using P9060_CountCards before using P9060_Open, and calling open with a specific card number (See prototype below). If open is successful, the function returns TRUE, and a handle to the PLX card.

Prototype

BOOL P9060_Open (P9060HANDLE *phPlx, DWORD dwVendorID, DWORD dwDeviceID, WORD nCardNum, DWORD options);

Parameters

dwVendorID, dwDeviceID - DeviceID / VendorID of the board to access.

nCardNum - If there are a few cards with the given VendorID and DeviceID, this will choose what one of them to open. This number should be from 0 (the first board) up to the value returned by P9060_CountCards() minus one. Most programs that do not need to handle multiple boards, can set nCardNum to 0.

options - options regarding the board opened. If P9060_OPEN_USE_INT, then the board must have an interrupt, and the interrupt will be installed.

Return Value

TRUE if OK.

Example

```
if (!P9060_Open( &hPlx, 0x10b5, 0x9060, 0, P9060_OPEN_USE_INT ))

{

  printf("Error opening device\n");

}
```

P9060_Close()

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

Prototype

void P9060_Close(P9060HANDLE hPlx);

Parameters

hPlx - handle to the PLX card to close.

Return Value

None

Example

```
P9060_Close(hPlx);
```

P9060_IsAddrSpaceActive()

Checks if the specified address space is enabled. The enabled address spaces are determined by the EEPROM, which at boot time sets the PLX's memory ranges requests. Use this function after calling P9060_Open() to make sure that the address space(s) that your driver is going to use are enabled.

Prototype

BOOL P9060_IsAddrSpaceActive(P9060HANDLE hPlx, P9060_ADDR addrSpace);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9060_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM.

Return Value

TRUE if address space is enabled.

Example

```
if ( !P9060_IsAddrSpaceActive(hPlx, P9060_ADDR_SPACE2) )

{

  printf ("Address space2 is not active!\n");

}
```

P9060_ReadReg(), P9060_WriteReg()

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

Prototype

1. DWORD P9060_ReadReg (P9060HANDLE hPlx, DWORD dwReg);

2. void P9060_WriteReg (P9060HANDLE hPlx, DWORD dwReg, DWORD dwData);

Parameters

1. hPlx - PLX card handle.

2. dwReg - offset of register to read or write. The constant values of registers are already defined in P60LIB.H, for example: P9060_DOORBELL_PTOL is the offset of DOORBELL_PTOL register.

3. dwData - data to write to register. (for P9060_WriteReg() only).

Return Value

Data read from register (for P9060_ReadReg() only).

P9060_ReadSpaceByte() ,P9060_ReadSpaceWord() , P9060_ReadSpaceDWord()

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

Prototype

1. BYTE P9060_ReadSpaceByte (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset);

2. WORD P9060_ReadSpaceWord (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset);

3. DWORD P9060_ReadSpaceDWord (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9060_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space to access.

3. dwOffset - offset in address space to read from.

Return Value

Data read from board.

P9060_WriteSpaceByte(),P9060_WriteSpaceWord(), P9060_WriteSpaceDWord()

Writes a byte / word / dword to address space on board.

Prototype

1. void P9060_WriteSpaceByte (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset, BYTE data);

2. void P9060_WriteSpaceWord (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset, WORD data);

3. void P9060_WriteSpaceDWord (P9060HANDLE hPlx, P9060_ADDR addrSpace, DWORD dwOffset, DWORD data);

Parameters

1. hPlx - PLX card handle.

2. addrSpace - Can be P9060_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space to access.

3. dwOffset - offset in address space to write to.

4. data - data to write to board

Return Value

None

P9060_ReadSpaceBlock() ,P9060_WriteSpaceBlock()

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

Prototype

1. void P9060_ReadSpaceBlock (P9060HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9060_ADDR addrSpace, P9060_MODE mode);

2. void P9060_WriteSpaceBlock (P9060HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9060_ADDR addrSpace, P9060_MODE mode);

Parameters

1. hPlx - PLX card handle.

2. dwOffset - offset in address space to read from or write to.

3. buf - a caller given buffer, into which data is read or written.

4. dwBytes - size to read in bytes

5. addrSpace - Can be P9060_ADDR_SPACE0 / SPACE1 / SPACE2 / SPACE3 / EPROM. Chooses the address space to access.

6. mode - access mode: P9060_MODE_BYTE/WORD/DWORD.

Return Value

None

P9060_ReadByte(),P9060_ReadWord(),P9060_ReadDWord()

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

Prototype

1. BYTE P9060_ReadByte (P9060HANDLE hPlx, DWORD dwLocalAddr);

2. WORD P9060_ReadWord (P9060HANDLE hPlx, DWORD dwLocalAddr);

3. DWORD P9060_ReadDWord (P9060HANDLE hPlx, DWORD dwLocalAddr);

Parameters

1. hPlx - PLX card handle.

2. dwLocalAddr - local address on board to write to.

Return Value

Data read from board.

P9060_WriteByte(), P9060_WriteWord(), P9060_WriteDWord()

Writes a byte / word / dword to memory on board.

Prototype

1. void P9060_WriteByte (P9060HANDLE hPlx, DWORD dwLocalAddr, BYTE data);

2. void P9060_WriteWord (P9060HANDLE hPlx, DWORD dwLocalAddr, WORD data);

3. void P9060_WriteDWord (P9060HANDLE hPlx, DWORD dwLocalAddr, DWORD data);

Parameters

1. hPlx - PLX card handle.

2. dwLocalAddr - local address on board to write to.

3. data - data to write to board

Return Value

None

P9060_ReadBlock(),P9060_WriteBlock()

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

Prototype

1. void P9060_ReadBlock (P9060HANDLE hPlx, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, P9060_MODE mode);

2. void P9060_WriteBlock (P9060HANDLE hPlx, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, P9060_MODE mode);

Parameters

1. hPlx - PLX card handle.

2. dwLocalAddr - local address on board to read from or write to.

3. buf - a caller given buffer, into which data is read or written.

4. dwBytes - size to read in bytes.

5. mode - local address access mode: P9060_MODE_BYTE/WORD/DWORD.

Return Value

None

P9060_IntIsEnabled()

Check whether interrupts are enabled or not.

Prototype

BOOL P9060_IntIsEnabled(P9060HANDLE hPlx);

Parameters

hPlx - PLX card handle.

Return Value

TRUE if interrupts are already enabled (i.e. if P9060_IntEnable() was called).

P9060_IntEnable()

Enable interrupt processing.

IMPORTANT NOTE*: The PLX 9060 uses level sensitive interrupts, therefore you must edit the implementation of this function (found in \windrvr\plx\9060\lib\p60lib.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 'WinDriver' manual, in the section regarding PCI interrupts implementation.*

Prototype

BOOL P9060_IntEnable(P9060HANDLE hPlx, P9060_INT_HANDLER funcIntHandler);

Parameters

1. hPlx - PLX card handle.

2. FuncIntHandler - callback interrupt handler. Upon each incoming interrupt, this function will be called.

Return Value

TRUE if successful.

P9060_IntDisable()

Disable interrupt processing.

Prototype

void P9060_IntDisable(P9060HANDLE hPlx);

Parameters

hPlx - PLX card handle.

Return Value

None

P9060_ReadPCIReg(),P9060_WritePCIReg()

Read from or write to the PCI configuration registers.

Prototype

1. DWORD P9060_ReadPCIReg (P9060HANDLE hPlx, DWORD dwReg);

2. void P9060_WritePCIReg (P9060HANDLE hPlx, DWORD dwReg, DWORD dwData);

Parameters

1. hPlx - PLX card handle.

2. dwReg - offset of PCI configuration register to read or write.

3. dwData - data to write (P9060_WritePCIReg() only).

Return Value

Data read from configuration register (for P9060_ReadPCIReg only).

PLX Register Definitions

Use these definitions for accessing the PLX 9060 specific registers.
Use when calling P9060_ReadReg() and P9060_WriteReg().

1. P9060_RANGE_PTOL_0 = 0x00
2. P9060_REMAP_PTOL_0 = 0x04
3. P9060_RANGE_PTOL_EPROM = 0x10
4. P9060_REMAP_PTOL_EPROM = 0x14
5. P9060_REGION_DESC = 0x18
6. P9060_MAILBOX_0 = 0x40
7. P9060_MAILBOX_1 = 0x44
8. P9060_MAILBOX_2 = 0x48
9. P9060_MAILBOX_3 = 0x4c
10. P9060_MAILBOX_4 = 0x50
11. P9060_MAILBOX_5 = 0x54
12. P9060_MAILBOX_6 = 0x58
13. P9060_MAILBOX_7 = 0x5c
14. P9060_DOORBELL_PTOL = 0x60
15. P9060_DOORBELL_LTOP = 0x64
16. P9060_INT_STATUS = 0x68
17. P9060_MISC_CONTROL = 0x6c

P9060_MODE

Used for P9060_ReadBlock() and P9060_WriteBlock.

1. P9060_MODE_BYTE = 0

2. P9060_MODE_WORD = 1

3. P9060_MODE_DWORD = 2

P9060_INT_RESULT

Used for the interrupt handler callback, given to P9060_IntEnable() to receive information on the interrupt.

Members

TYPE	NAME	DESCRIPTION
DWORD	dwCounter	number of interrupts received
DWORD	dwLost	number of interrupts not yet dea
BOOL	fStopped	true if interrupt was disabled dur
DWORD	dwIntStatusReg	value of status register when int occurred

P9060_Open() fails.

The following may cause P9060_Open() to fail:

Cause: WinDriver's kernel is not loaded.

Action: Run WDREG.EXE install (in the \windrvr\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.

Cause: The 30 day evaluation license is over.

Action: In this case, the WinDriver will inform you that your evaluation license is over, in a message box. Please contact sales@jungo.com to purchase WinDriver.

Cause: The VendorID / DeviceID requested in P9060_Open() do not match that of the board. (In licensed versions).

Action: Run P60_DIAG.EXE and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

Cause: The PLX device is not installed or configured correctly.

Action: Run P60_DIAG.EXE (found in your \windrvr\util directory), and choose PCI scan. Check that your PLX device returns all the resources needed.

Cause: Your PLX device is in use by another application.

Action: Close all other applications that might be using the PLX device.

Introduction

The AMCC Diagnostics (AMCCDIAG.EXE) accesses the hardware using WinDriver, therefore WinDriver must be installed and running before being able to run AMCCDIAG. Once WinDriver is running, you may run AMCCDIAG.

Click 'Start / Programs / WinDriver / AMCC Diagnostics'. The application will first try to locate the card, with the default VendorID and DeviceID assigned by AMCC. If such a card is found you will get a message "AMCC 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 AMCC 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 AMCC 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.

AMCC S5933 local registers: This option is available only after choosing an active card. A list of the AMCC S5933 registers and their READ value are displayed. In order to WRITE to a S5933 register, enter the register number, and then enter the value to write to it.

Access memory spaces on the board: This option is available only after choosing an active card. This option assumes that the configuration EEPROM has initialised the address spaces. To choose the address range, toggle the active memory space mode. To access the location as BYTE / WORD / DWORD toggle the active mode option.

- To READ the location, choose Read from board. You will be asked for an offset into the space to read from.

- To WRITE to location, choose Write to board. You will for an offset into the space and the data to write.

AMCC_CountCards()

Counts number of cards in the PCI bus whose VendorID / DeviceID is the same as the VendorID / DeviceID given as parameters.

Prototype

DWORD AMCC_CountCards (DWORD dwVendorID, DWORD dwDeviceID);

Parameters

1. dwVendorID - Vendor ID of your AMCC 5933 card. If 0, detects cards from all vendors.

2. dwDeviceID - Device ID of your AMCC 5933 card. If 0, detects all devices.

Return value

Returns the number of matching PCI cards found.

AMCC_Open()

Opens a handle to a AMCC card. The handle will be used in future calls to the card.

Prototype

BOOL AMCC_Open (AMCCHANDLE *phAMCC, DWORD dwVendorID, DWORD dwDeviceID, DWORD nCardNum, DWORD dwOptions);

Parameters

1. dwVendorId - PCI Vendor ID of card to open. If 0, detects cards from all vendors.

2. dwDeviceId - PCI Device ID of card to open. If 0, detects all devices.

3. nCardNum - If there are a few cards with the given VendorID and DeviceID, this will choose which one of them to open. This number should be from 0 (the first board) up to the value returned by AMCC_CountCards() minus one. Most programs that do not need to handle multi-boards, can set nCardNum to 0.

4. dwOptions - 0 if no interrupts needed, or AMCC_OPEN_USE_INT for use of interrupts in session. You may use wildcards in the DeviceID / VendorID settings to detect a number of cards, and loop on AMCC_Open() while incrementing nCardNum, to open all cards detected.

Return value

Returns TRUE if open card succeeds.

AMCC_Close()

Must be called when ending session. Frees handle to card.

Prototype

void AMCC_Close (AMCCHANDLE hAmcc);

Parameters

hAmcc - Handle to the AMCC S5933 card obtained with AMCC_Open.

AMCC_IsAddrSpaceActive()

Returns TRUE if addrSpace exists on card.

Prototype

BOOL AMCC_IsAddrSpaceActive(AMCCHANDLE hAmcc, AMCC_ADDR addrSpace);

Parameters

1. hAmcc - AMCC card handle.

2. addrSpace - The address space to be checked.

Return value

Returns TRUE if address space is active.

AMCC_ReadRegByte(),AMCC_ReadRegWord(), AMCC_ReadRegDWord()

These functions read a Byte/Word/DWord from a register.

Prototypes

1. BYTE AMCC_ReadRegByte (AMCCHANDLE hAmcc, DWORD dwReg);

2. WORD AMCC_ReadRegWord (AMCCHANDLE hAmcc, DWORD dwReg);

3. DWORD AMCC_ReadRegDWord (AMCCHANDLE hAmcc, DWORD dwReg);

Parameters

1. hAmcc - AMCC card handle.

2. dwReg - offset of register to read.

Return value

Data read from register.

AMCC_WriteRegByte(),AMCC_WriteRegWord(), AMCC_WriteRegDWord()

These functions write a Byte/Word/DWord to a register.

Prototypes

1. void AMCC_WriteRegByte (AMCCHANDLE hAmcc, DWORD dwReg, BYTE bData);

2. void AMCC_WriteRegWord (AMCCHANDLE hAmcc, DWORD dwReg, WORD wData);

3. void AMCC_WriteRegDWord (AMCCHANDLE hAmcc, DWORD dwReg, DWORD dwData);

Parameters

1. hAmcc - AMCC card handle.

2. data - the data to write to the register.

Return value

None

AMCC_ReadByte(), AMCC_ReadWord(), AMCC_ReadDWord()

These functions read a Byte/Word/DWord from an address space on the card.

Prototype

1. BYTE AMCC_ReadByte (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset);

2. WORD AMCC_ReadWord (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset);

3. DWORD AMCC_ReadDWord (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset);

Parameters

1. hAmcc - AMCC card handle.

2. addrSpace - Address space on card to read to / write from.

3. dwOffset - offset of address to read to / write from, in the address space

Return value

Data read from card.

AMCC_WriteByte(),AMCC_WriteWord(),AMCC_WriteDWord()

These functions write a Byte/Word/DWord to an address space on the card.

Prototype

1. void AMCC_WriteByte (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset, BYTE data);

2. void AMCC_WriteWord (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset, WORD data);

3. void AMCC_WriteDWord (AMCCHANDLE hAmcc, AMCC_ADDR addrSpace, DWORD dwOffset, DWORD data);

Parameters

1. hAmcc - AMCC card handle.

2. addrSpace - Address space on card to read to / write from

3. dwOffset - offset of address to read to / write from, in the address space

4. data - the data to write.

Return value

None

AMCC_ReadSpaceBlock(), AMCC_WriteSpaceBlock()

Read / Write a block of data from card to buffer.

Prototype

1. void AMCC_ReadSpaceBlock (AMCCHANDLE hAmcc, DWORD dwOffset, PVOID buf, DWORD dwBytes, AMCC_ADDR addrSpace);

2. void AMCC_WriteSpaceBlock (AMCCHANDLE hAmcc, DWORD dwOffset, PVOID buf, DWORD dwBytes, AMCC_ADDR addrSpace);

Parameters

1. hAmcc - AMCC card handle.

2. buf - buffer to read data into / from

3. addrSpace - Address space on card to read from / write to

4. dwLocalAddr - offset of address to read from / write to, in the address space

5. dwBytes - number of bytes to read / write

AMCC_IntIsEnabled()

Check whether interrupts are enabled or not.

Prototype

BOOL AMCC_IntIsEnabled(AMCCHANDLE hAmcc);

Parameters

hAmcc - AMCC card handle.

Return Value

TRUE if interrupts are already enabled (AMCC_IntEnable() was called).

AMCC_IntEnable()

Enable interrupt handler routines.

IMPORTANT NOTE: The AMCC S5933 uses level sensitive interrupts, therefore you must edit the implementation of this function (found in \windrvr\amcc\lib\amcclib.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 'WinDriver' manual,in the section regarding PCI interrupts implementation.

Prototype

BOOL AMCC_IntEnable(AMCCHANDLE hAmcc, AMCC_INT_HANDLER funcIntHandler);

Parameters

1. hAmcc - AMCC card handle.

2. FuncIntHandler - callback interrupt handler. For each incoming interrupt, this function will be called.

Return value

Returns TRUE if interrupt enable succeeds.

AMCC_IntDisable()

Disable interrupt handler routines.

Prototype

void AMCC_IntDisable(AMCCHANDLE hAmcc);

Parameters

hAmcc - AMCC card handle.

AMCC_ReadPCIReg(), AMCC_WritePCIReg()

Read from or write to the PCI configuration registers.

Prototype

1. DWORD AMCC_ReadPCIReg (AMCCHANDLE hAmcc, DWORD dwReg);

2. void AMCC_WritePCIReg (AMCCHANDLE hAmcc, DWORD dwReg, DWORD dwData);

Parameters

1. hAmcc - AMCC card handle.

2. dwReg - offset of PCI configuration register to read or write.

3. dwData - data to write (AMCC_WritePCIReg() only).

Return Value

Data read from configuration register (for AMCC_ReadPCIReg only).

PCI Configuration Registers Definitions

Use for AMCC_ReadReg() and AMCC_WriteReg() functions.

1. OMB1_ADDR = 0x00
2. OMB2_ADDR = 0x04
3. OMB3_ADDR = 0x08
4. OMB4_ADDR = 0x0c
5. IMB1_ADDR = 0x10
6. IMB2_ADDR = 0x14
7. IMB3_ADDR = 0x18
8. IMB4_ADDR = 0x1c
9. FIFO_ADDR = 0x20
10. MWAR_ADDR = 0x24
11. MWTC_ADDR = 0x28
12. MRAR_ADDR = 0x2c
13. MRTC_ADDR = 0x30
14. MBEF_ADDR = 0x34
15. INTCSR_ADDR = 0x38
16. BMCSR_ADDR = 0x3c

AMCC_ADDR

Used to choose the address space on the card.

1. AMCC_ADDR_REG = 0

2. AMCC_ADDR_SPACE0 = 1

3. AMCC_ADDR_SPACE1 = 2

4. AMCC_ADDR_SPACE2 = 3

5. AMCC_ADDR_SPACE3 = 4

AMCC_INT_RESULT

Used for the interrupt handler callback, given to AMCC_IntEnable() to receive information on the interrupt.

Members

TYPE	NAME	DESCRIPTION
DWORD	dwCounter	number of interrupts received
DWORD	dwLost	number of interrupts not yet dea
BOOL	fStopped	was interrupt disabled during wa
DWORD	dwIntStatusReg	value of status register when int occurred

AMCC_Open() fails.

The following may cause AMCC_Open() to fail:

Cause: WinDriver's kernel is not loaded.

Action: Run WDREG.EXE install (in the \windrvr\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.

Cause: The 30 day evaluation license is over.

Action: In this case, the WinDriver will inform you that your evaluation license is over, in a message box. Please contact sales@jungo.com to purchase WinDriver.

Cause: The VendorID / DeviceID requested in AMCC_Open() do not match that of the board. (In licensed versions).

Action: Run AMCCDIAG.EXE and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

Cause: The AMCC device is not installed or configured correctly.

Action: Run AMCCDIAG.EXE (found in your \windrvr\util directory), and choose PCI scan. Check that your AMCC device returns all the resources needed.

Cause: Your AMCC device is in use by another application.

Action: Close all other applications that might be using the AMCC device.

Introduction

The V3 PBC Diagnostics (PBC_DIAG.EXE) accesses the hardware using WinDriver, therefore WinDriver must be installed before being able to run PBC_DIAG. Once WinDriver is running, you may run PBC_DIAG.

Click 'Start / Programs / WinDriver / V3 PBC Diagnostics'. The application will first try to locate the card, with the default VendorID and DeviceID assigned by V3 PBC. If such a card is found you will get a message "V3 PBC 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 V3 PBC 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 V3 PBC 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.

V3 PBC local registers: This option is available only after choosing an active card. A list of the V3 PBC registers and their READ value are displayed. In order to WRITE to a V3 PBC register, enter the register number, and then enter the value to write to it.

Access local memory ranges on the board: This option 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 address regions. 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. The V3 PBC has two memory ranges to access a local region. There is no difference between them, therefore you need not change the active memory range.

- 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. PCI_MAP0 register is set with the base address.

Access memory spaces on the board: This option 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 address regions. To access Configuration, Aperture zero or one space, toggle the active memory space mode. To access the location as BYTE / WORD / DWORD, toggle the active mode option.

- To READ the location, choose Read from board. You will be asked for an offset into the space to read from.
- To WRITE to location, choose Write to board. You will for an offset into the space and the data to write.

Access EEPROM device: 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: 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.

V3PBC_CountCards()

Counts number of cards in the PCI bus whos VendorID / DeviceID is the same as the VendorID / DeviceID given as parameters.

Prototype

DWORD V3PBC_CountCards (DWORD dwVendorID, DWORD dwDeviceID);

Parameters

1. dwVendorID - Vendor ID of your V3 PBC card. If 0, detects cards from all vendors.

2. dwDeviceID - Device ID of your V3 PBC card. If 0, detects all devices.

Return value

Returns the number of matching PCI cards found.

V3PBC_Open()

Opens a handle to a V3PBC card. The handle will be used in future calls to the card.

Prototype

BOOL V3PBC_Open (V3PBCHANDLE *phV3, DWORD dwVendorID, DWORD dwDeviceID, DWORD nCardNum, DWORD dwOptions);

Parameters

1. dwVendorId - PCI Vendor ID of card to open. If 0, detects cards from all vendors.

2. dwDeviceId - PCI Device ID of card to open. If 0, detects all devices.

3. nCardNum - If there are a few cards with the given VendorID and DeviceID, this will choose what one of them to open. This number should be from 0 (the first board) up to the value returned by V3PBC_CountCards() minus one. Most programs that do not need to handle multi-boards, can set nCardNum to 0.

4. dwOptions - 0 if no interrupts needed, or V3PBC_OPEN_USE_INT for use of interrupts in session. You may use wildcards in the DeviceID / VendorID settings to detect a number of cards, and loop on V3PBC_Open() while incrementing nCardNum, to open all cards detected.

Return value

Returns TRUE if open card succeeds.

V3PBC_Close()

Must be called when ending session. Frees handle to card.

Prototype

void V3PBC_Close (V3PBCHANDLE hV3);

Parameters

hV3 - Handle to the V3 card obtained with V3PBC_Open.

V3PBC_IsAddrSpaceActive()

Returns TRUE if addrSpace exists on card.

Prototype

BOOL V3PBC_IsAddrSpaceActive(V3PBCHANDLE hV3, V3PBC_ADDR addrSpace);

Parameters

1. hV3 - V3 PBC card handle.

2. addrSpace - The address space to be checked.

Return value

Returns TRUE if address space is active.

V3PBC_GetRevision()

Returns the V3 PBC silicon revision (equates defined in V3PBC_REV).

Prototype

DWORD V3PBC_GetRevision(V3PBCHANDLE hV3);

Parameters

hV3 - V3 PBC card handle.

Return value

Returns the silicon revision.

V3PBC_ReadRegByte(), V3PBC_ReadRegWord(), V3PBC_ReadRegDWord()

These functions read a Byte/Word/DWord from a register.

Prototypes

1. BYTE V3PBC_ReadRegByte (V3PBCHANDLE hV3, DWORD dwReg);

2. WORD V3PBC_ReadRegWord (V3PBCHANDLE hV3, DWORD dwReg);

3. DWORD V3PBC_ReadRegDWord (V3PBCHANDLE hV3, DWORD dwReg);

Parameters

1. hV3 - V3 PBC card handle.

2. dwReg - offset of register to read.

Return value

Data read from register.

V3PBC_WriteRegByte(), V3PBC_WriteRegWord(), V3PBC_WriteRegDWord()

These functions write a Byte/Word/DWord to a register.

Prototypes

1. void V3PBC_WriteRegByte (V3PBCHANDLE hV3, DWORD dwReg, BYTE bData);

2. void V3PBC_WriteRegWord (V3PBCHANDLE hV3, DWORD dwReg, WORD wData);

3. void V3PBC_WriteRegDWord (V3PBCHANDLE hV3, DWORD dwReg, DWORD dwData);

Parameters

1. hV3 - V3 PBC card handle.

2. dwReg - offset of register to read.

3. data - the data to write.

Return value

None

V3PBC_ReadSpaceByte(), V3PBC_WriteSpaceByte(), V3PBC_ReadSpaceWord(), V3PBC_WriteSpaceWord(), V3PBC_ReadSpaceDWord(), V3PBC_WriteSpaceDWord()

These functions Read/Write a Byte/Word/DWord to/from an address space on the card.

Prototype

1. BYTE V3PBC_ReadSpaceByte (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset);
2. void V3PBC_WriteSpaceByte (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset, BYTE data);
3. WORD V3PBC_ReadSpaceWord (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset);
4. void V3PBC_WriteSpaceWord (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset, WORD data);
5. DWORD V3PBC_ReadSpaceDWord (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset);
6. void V3PBC_WriteSpaceDWord (V3PBCHANDLE hV3, V3PBC_ADDR addrSpace, DWORD dwOffset, DWORD data);

Parameters

1. hV3 - V3 PBC card handle.
2. addrSpace - Address space on card to read to / write from
3. dwOffset - offset of address to read to / write from, in the address space
4. data - the data to write.

Return value

For read functions, the return value is the data read.

V3PBC_ReadSpaceBlock(), V3PBC_WriteSpaceBlock()

Read / Write a block of data from card to buffer.

Prototype

1. void V3PBC_ReadSpaceBlock (V3PBCHANDLE hV3, DWORD dwOffset, PVOID buf, DWORD dwBytes, V3PBC_ADDR addrSpace);

2. void V3PBC_WriteSpaceBlock (V3PBCHANDLE hV3, DWORD dwOffset, PVOID buf, DWORD dwBytes, V3PBC_ADDR addrSpace);

Parameters

1. hV3 - V3 PBC card handle.
2. buf - buffer to write/read data into / from
3. addrSpace - Address space on card to read from / write to
4. dwLocalAddr - offset of address to read from / write to, in the address space
5. dwBytes - number of bytes to read / write

V3PBC_ReadBlock(), V3PBC_WriteBlock()

Reads or writes a block of data from card to or from a buffer.

Prototype

1. void V3PBC_ReadBlock (V3PBCHANDLE hV3, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, V3PBC_ADDR addrSpace);

2. void V3PBC_WriteBlock (V3PBCHANDLE hV3, DWORD dwLocalAddr, PVOID buf, DWORD dwBytes, V3PBC_ADDR addrSpace);

Parameters

1. hV3 - V3 PBC card handle.

2. buf - buffer to read or write data into.

3. addrSpace - Address space on card to read or write.

4. dwLocalAddr - offset of address to read or write, in the address space

5. dwBytes - number of bytes to read.

V3PBC_ReadByte(), V3PBC_ReadByte(), V3PBC_WriteByte(), V3PBC_ReadWord(), V3PBC_WriteWord(), V3PBC_ReadDWord(), V3PBC_WriteDWord()

These functions Read/Write a Byte/Word/DWord to/from a local address on the card.

Prototypes

1. BYTE V3PBC_ReadByte (V3PBCHANDLE hV3, DWORD dwLocalAddr);
2. void V3PBC_WriteByte (V3PBCHANDLE hV3, DWORD dwLocalAddr, BYTE data);
3. WORD V3PBC_ReadWord (V3PBCHANDLE hV3,DWORD dwLocalAddr);
4. void V3PBC_WriteWord (V3PBCHANDLE hV3, DWORD dwLocalAddr, WORD data);
5. DWORD V3PBC_ReadDWord (V3PBCHANDLE hV3, DWORD dwLocalAddr);
6. void V3PBC_WriteDWord (V3PBCHANDLE hV3, DWORD dwLocalAddr, DWORD data);

Parameters

1. hV3 - V3 PBC card handle.
2. dwLocalAddr - local address to read to / write from.
3. data - the data to write.

Return value

For read functions, the return value is the data read.

V3PBC_IntIsEnabled()

Check whether interrupts are enabled or not.

Prototype

BOOL V3PBC_IntIsEnabled(V3PBCHANDLE hV3);

Parameters

hV3 - V3 PBC card handle.

Return Value

TRUE if interrupts are already enabled (V3PBC_IntEnable() was called).

V3PBC_IntEnable()

Enable interrupt handler routines.

IMPORTANT NOTE*: The V3 PBC uses level sensitive interrupts, therefore you must edit the implementation of this function (found in \windrvr\v3\lib\pbclib.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 'WinDriver' manual, in the section regarding PCI interrupts implementation.*

Prototype

BOOL V3PBC_IntEnable(V3PBCHANDLE hV3, V3PBC_INT_HANDLER funcIntHandler);

Parameters

1. hV3 - V3 PBC card handle.

2. FuncIntHandler - callback interrupt handler. For each incoming interrupt, this function will be called.

Return value

Returns TRUE if interrupt enable succeeds.

V3PBC_IntDisable()

Disable interrupt handler routines.

Prototype

void V3PBC_IntDisable(V3PBCHANDLE hV3);

Parameters

hV3 - V3 PBC card handle.

V3PBC_ReadPCIReg(), V3PBC_WritePCIReg()

Read from or write to the PCI configuration registers.

Prototype

1. DWORD V3PBC_ReadPCIReg (V3PBCHANDLE hV3, DWORD dwReg);

2. void V3PBC_WritePCIReg (V3PBCHANDLE hV3, DWORD dwReg, DWORD dwData);

Parameters

1. hV3 - V3 PBC card handle.

2. dwReg - offset of PCI configuration register to read or write.

3. dwData - data to write (V3PBC_WritePCIReg() only).

Return Value

Data read from configuration register (for V3PBC_ReadPCIReg only).

V3PBC_DMAOpen()

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

Prototype

BOOL V3PBC_DMAOpen(V3PBCHANDLE hV3, WD_DMA *pDMA, DWORD dwBytes);

Parameters

1. hV3 - V3 PBC card handle.

2. pDMA - Pointer to a DMA structure that will be used during the DMA.

3. dwBytes - Size of buffer to allocate.

Return value

Returns TRUE if DMA buffer allocation succeeds.

V3PBC_DMAClose()

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

Prototype

void V3PBC_DMAClose(V3PBCHANDLE hV3, WD_DMA *pDMA);

Parameters

1. hV3 - V3 PBC card handle.

2. pDMA - pointer to the DMA buffer to release.

V3PBC_DMAStart()

Start DMA to/from card.

Prototype

BOOL V3PBC_DMAStart(V3PBCHANDLE hV3, V3PBC_DMA_CHANNEL dmaChannel, WD_DMA *pDMA, BOOL fRead, BOOL fBlocking, DWORD dwBytes, DWORD dwOffset, DWORD dwLocalAddr);

Parameters

1. hV3 - V3 PBC card handle.

2. dmaChannel - the channel to use for DMA transfer. Can be V3PBC_DMA_0 or V3PBC_DMA_1.

3. pDMA - initialised with V3PBC_DMAOpen

4. fRead - TRUE: read from card to buffer. FALSE: write from buffer to card

5. fBlocking - TRUE: wait until DMA is done

6. dwBytes - number of bytes to transfer (must be a multiple of 4)

7. dwOffset - offset in DMA buffer to read to / write from.

8. dwLocalAddr - local address on card to write to / read from

Return value

Returns TRUE if DMA transfer succeeds.

V3PBC_DMAIsDone()

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

Prototype

BOOL V3PBC_DMAIsDone(V3PBCHANDLE hV3, V3_DMA_CHANNEL dmaChannel);

Parameters

1. hV3 - V3 PBC card handle.

2. dmaChannel - the channel to test. Can be V3PBC_DMA_0 or V3PBC_DMA_1.

Return value

Returns TRUE if DMA transfer is completed.

V3PBC_PulseLocalReset()

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

Prototype

void V3PBC_PulseLocalReset(V3PBCHANDLE hV3, WORD wDelay);

Parameters

1. hV3 - V3 PBC card handle.

2. wDelay - Number of milliseconds to hold the reset down.

V3PBC_EEPROMInit()

Initialises I2C bus. Call this function once before calling V3PBC_EEPROMWrite() and V3PBC_EEPROMRead().

Prototype

void V3PBC_EEPROMInit(V3PBHANDLE hV3);

Parameters

hV3 - V3 PBC card handle.

V3PBC_EEPROMWrite()

Programs one byte to the EEPROM.

Prototype

BOOL V3PBC_EEPROMWrite(V3PBCHANDLE hV3, BYTE bSlaveAddr, BYTE bAddr, BYTE bData);

Parameters

1. hV3 - V3 PBC card handle.
2. bSlaveAddr - slave device to use
3. bAddr - address to write to
4. bData - data to write

Return value

Returns TRUE if EEPROM write succeeds.

V3PBC_EEPROMRead()

Reads one byte from the EEPROM.

Prototype

BOOL V3PBC_EEPROMRead(V3PBCHANDLE hV3, BYTE bSlaveAddr, BYTE bAddr, BYTE *bData);

Parameters

1. hV3 - V3 PBC card handle.

2. bSlaveAddr - slave device to use.

3. bAddr - address to read from

4. bData - data read.

Return value

Returns TRUE if EEPROM read succeeds.

V3PBC_ADDR

Used to choose the address space on the card.

1. V3PBC_ADDR_IO_BASE = 0
2. V3PBC_ADDR_BASE0 = 1
3. V3PBC_ADDR_BASE1 = 2
4. V3PBC_ADDR_ROM = 3

V3PBC_INT_RESULT

Used for the interrupt handler callback, given to V3PBC_IntEnable() to receive information on the interrupt.

Members

TYPE	NAME	DESCRIPTION
DWORD	dwCounter	number of interrupts received
DWORD	dwLost	number of interrupts not yet dealt with
BOOL	fStopped	was interrupt disabled during wait ?
DWORD	dwIntStatusReg	value of status register when interrupt occurred

V3PBC_Open() fails

The following may cause V3PBC_Open() to fail:

Cause: WinDriver's kernel is not loaded.

Action: Run WDREG.EXE install (in the \windrvr\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.

Cause: The 30 day evaluation license is over.

Action: In this case, the WinDriver will inform you that your evaluation license is over, in a message box. Please contact sales@jungo.com to purchase WinDriver.

Cause: The VendorID / DeviceID requested in V3PBC_Open() do not match that of the board. (In licensed versions).

Action: Run PBC_DIAG.EXE and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

Cause: The V3 device is not installed or configured correctly.

Action: Run PBC_DIAG.EXE (found in your \windrvr\util directory), and choose PCI scan. Check that your V3 device returns all the resources needed.

Cause: Your V3 device is in use by another application.

Action: Close all other applications that might be using the V3 device.

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:

- **\windriver\plx\9054\lib\p9054_lib.c**
  - **\windriver\plx\9080\lib\p9080_lib.c**
  - **\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

Contiguous Buffer DMA

More detailed examples can be found at:

- **windriver\v3\lib\pbclib.c**
    - **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,
  // 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,
   LocalAddr);

  // Wait for transfer to end

  while(!My_Dma_Done());

  // release the buffer

  WD_DMAUnlock(hWD,&dma);

}
```

General - Handling an Interrupt

- A thread that will handle incoming interrupts needs to be created.
  - The interrupt handler thread will run an infinite loop that waits for an interrupt to occur.
  - When an interrupt occurs, the driver's interrupt handler code is called.
  - When an 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 for an interrupt. Once an interrupt occurs, it is first handled by the WinDriver kernel, then the WD_IntWait() wakes up the interrupt handler thread and returns.

Since your interrupt thread runs in 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);

}
```

Simplified interrupt handling using *windrvr_int_thread.h*

From Version 4.3 onwards, a new header file *windrvr_int_thread.h* simplifies the code you need to write to handle interrupts. In this header file -- found under**windriver/include**, we provide the convienence functions InterruptThreadEnable [InterruptThreadEnable()] and InterruptThreadDisable[InterruptThreadDisable()]. These functions are implemented as static functions in the header file *windrvr_int_thread.h*. Please study the code in the header file to understand how this operates. We may rewrite the code from SectionGeneral - Handling an Interrupt as follows. This code was extracted from the sample program*int_io.c* which can be found under **windriver/samples/int_io**. Please refer to this file for the full listing.

```
// interrupt handler routine. you can use pData to pass
// information from InterruptThreadEnable()
VOID interrupt_handler (PVOID pData)
{
    WD_INTERRUPT * pIntrp = (WD_INTERRUPT *) pData;
    // do your interrupt routine here
    printf ("Got interrupt %d\n", pIntrp->dwCounter);
}

...

int main()
{
    HANDLE hWD;
    WD_CARD_REGISTER cardReg;
    // interrupt structure
    WD_INTERRUPT Intrp;
    HANDLE thread_handle;

    ...
    hWD = WD_Open();
    BZERO(cardReg);
    cardReg.Card.dwItems = 1;
    cardReg.Card.Item[0].item = ITEM_INTERRUPT;
    cardReg.Card.Item[0].fNotSharable = TRUE;
    cardReg.Card.Item[0].I.Int.dwInterrupt = MY_IRQ;
    cardReg.Card.Item[0].I.Int.dwOptions = 0;
    ...
    WD_CardRegister (hWD, &cardReg);
    ...
    PVOID  pData = NULL;
```

```
    BZERO(Intrp);
    Intrp.hInterrupt = cardReg.Card.Item[0].I.Int.hInterrupt;
    Intrp.Cmd = NULL;
    Intrp.dwCmds = 0;
    Intrp.dwOptions = 0;
    printf ("starting interrupt thread\n");
    // this calls WD_IntEnable() and creates an interrupt
    // handler thread which calls the function
    // interrupt_handler with the pointer pData as a parameter

    pData = &Intrp;
    ...
    if (!InterruptThreadEnable(&thread_handle, hWD, &Intrp,
         interrupt_handler, pData))
      {
         printf ("failed enabling interrupt\n");
      }
    else
      {
        // call your driver code here
        printf ("Press Enter to uninstall interrupt\n");
        fgets(line, sizeof(line), stdin);

        // this calls WD_IntDisable()
        InterruptThreadDisable(thread_handle);
      }
    WD_CardUnregister(hWD, &cardReg);
    ....
}
```

In the above code, the function *interrupt_handler* serves as our "interrupt handler", getting invoked once for every interrupt that occurs. In the simplified code for setting up the interrupt handling, we call InterruptThreadEnable[InterruptThreadEnable()] spawn a thread that calls the function *interrupt_handler* -- a pointer to this function is passed as the fourth parameter to InterruptThreadEnable -- each time an interrupt occurs, passing into this function, the data *pData* specified by the fifth parameter.

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 can be found under Section Computer hangs on interrupt that explains how to deal with the problem of the computer hanging 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 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;
}
```

If you study the implementation of the interrupt handling in *windrvr_int_thread.h*, you see that code similar to the above is used there.

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

attempt to initialize 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 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.

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

Supposing you have a PCI card in your X86 CEPC and the BIOS assigned IRQ9 to it. Since WinCE does not map this interrupt by default, you will not be able to receive interrupts from this card. In this case, you will need to insert a similar entry for IRQ 9.

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

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

For non-X86 machines like the hand-held PCs from HP and Sharp, the developer should use the logical interrupt ID which can be in 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 or Platform Builder 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.

USB Data Exchange

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

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

- Control exchange is used to configure a device when it is first attached, getting common configuration data, and can be also used for other device-specific purposes, including control of other pipes on the device. The control exchange is transferred via the control pipe (Pipe 00).

The control transfer consists of a setup stage (in which a setup packet is sent from the host to the device), an optional data stage and a status stage.

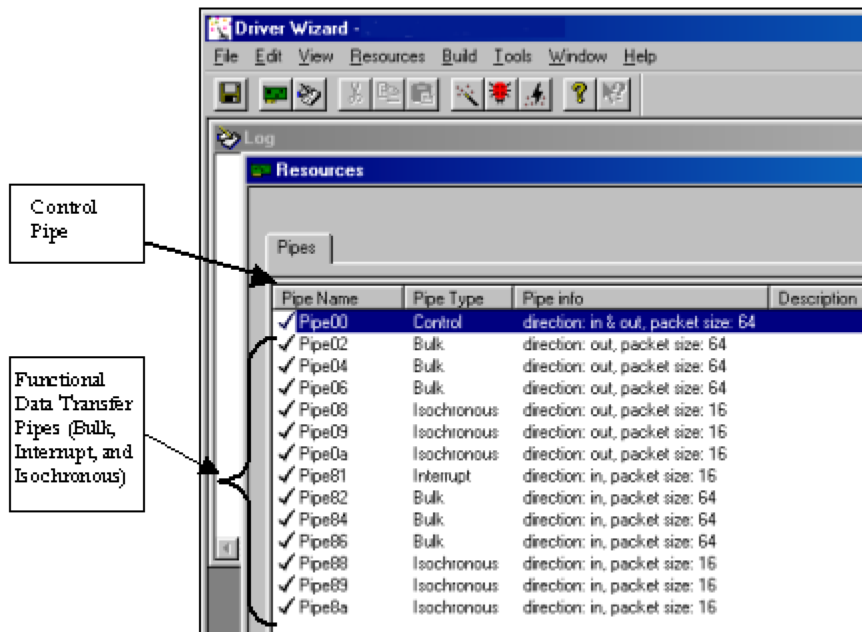


Figure 13.1: USB Data Exchange

More about the Control Transfer

The control transaction always begins with a setup stage. Then, it is followed by zero or more control data transactions (data stage) that carry the specific information for the requested operation, and finally, a Status transaction completes the control transfer by returning the status to the host.

During the setup stage, a SETUP Packet is used to transmit information to the control endpoint of the device. The Setup Packet consists of eight bytes, and its format is specified in the USB specification.

A control transfer can be a read transaction or a write transaction. In a read transaction, the Setup Packet indicates the characteristics and amount of data to be read from the device. In a write transaction, the Setup Packet contains the command sent (written) to the device and the number of control Data bytes, associated with this transaction, that are sent to the device in the data stage.

The following figure shows the sequence of read and write transactions (the figure is taken from the USB specification). 'In' means the data flows from the device to the host. 'Out' means the data flows from the host to the device.

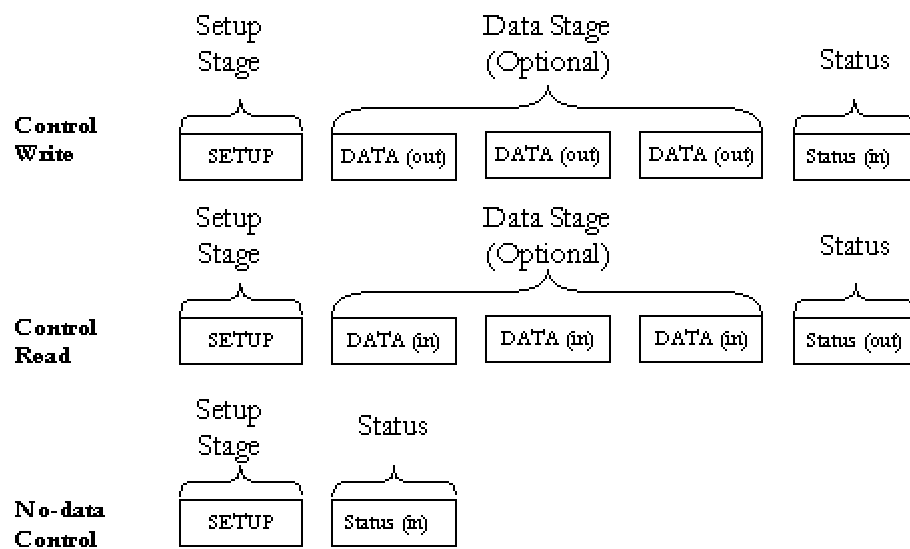


Figure 13.2: USB Read and Write

The Setup Packet

The Setup Packets (combined with the control data stage and the status stage) are used to configure and send commands to the device. Chapter 9 of the USB specification defines standard device requests. USB requests such as these are sent from the host to the device, using Setup Packets. The USB device is required to respond properly to these requests. In addition, each vendor may define device-specific Setup Packets, to perform device specific operations. The standard Setup Packets (standard USB device requests) are detailed below. The vendor's device-specific Setup Packets are detailed in the vendor's specific data book for each USB device.

USB Setup Packet Format

The table below shows the format of the USB setup packet. (For further information please refer to the USB specification at `http://www.usb.org`.)

Byte	Field	Description
0	bmRequest Type	Bit 7: Request direction (0=Host - out, 1=Device to host - in) Bits 5..6:Request type (0=standard, 2=vendor, 3=reserved) Bits 0..4 (0=device, 1=interface, 2=endpoint,3=other)
1	bRequest	The actual request (see next tab
2	wValueL	A word-size value that varies ac the request (for example in the CLEAR_FEATURE request, the used to select the feature, in the GET_DESCRIPTOR request, th indicates the descriptor type, in SET_ADDRESS request, the va contains the device address)
3	wValueH	The upper byte of the Value wor
4	wIndexL	A word size value that varies ac the request. The index is genera specify an endpoint or an interfa
5	wIndexH	The upper byte of the Index wor
6	wLengthL	Word size value, indicates the n bytes to be transferred if there is stage.
7	wLengthH	The upper byte of the Length wo

Standard device requests codes

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Setup Packet example

This is an example of a standard USB device request to illustrate the Setup Packet format and its different fields. The Setup packet is in Hex format.

The following Setup Packet is a 'Control Read' transaction that retrieves the 'Device descriptor' from the USB device. The 'Device descriptor' includes information such as USB standard revision, the vendor ID and the device product ID.

GET_DESCRIPTOR (device) Setup Packet

80	06	00

Setup Packet meaning:

Byte	**Field**	**Value**
0	BmRequest Type	80
1	bRequest	06
2	wValueL	00
3	wValueH	01
4	wIndexL	00
5	wIndexH	00
6	wLengthL	12
7	wLengthH	00

In response, the device sends the 'Device Descriptor' data. For example, this is a 'Device Descriptor' of 'Cypress EZ-USB Integrated Circuit':

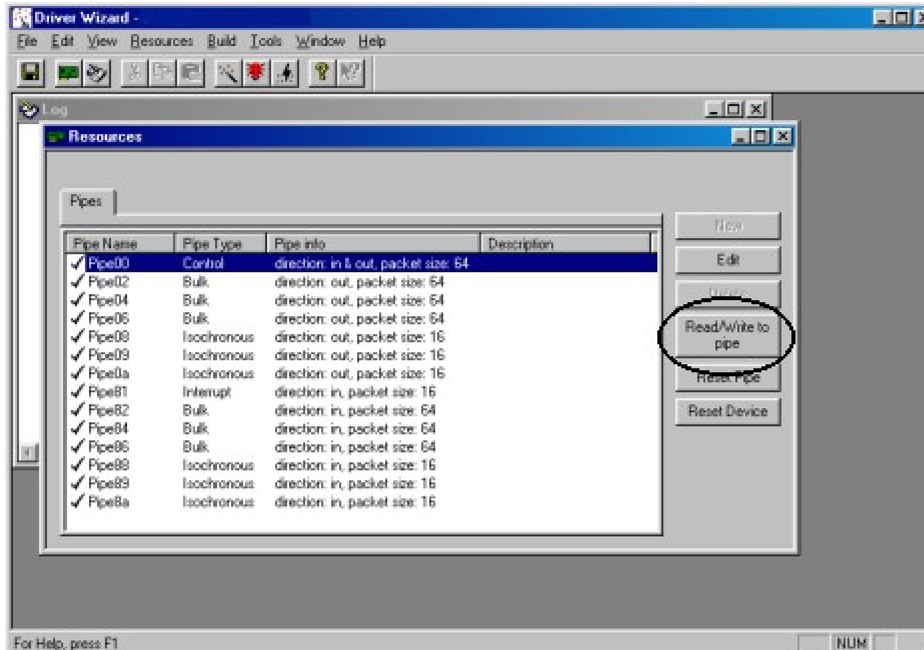
Byte No.	0	1	2	3
Content	12	01	00	01

Byte No.	11	12
Content	00	01

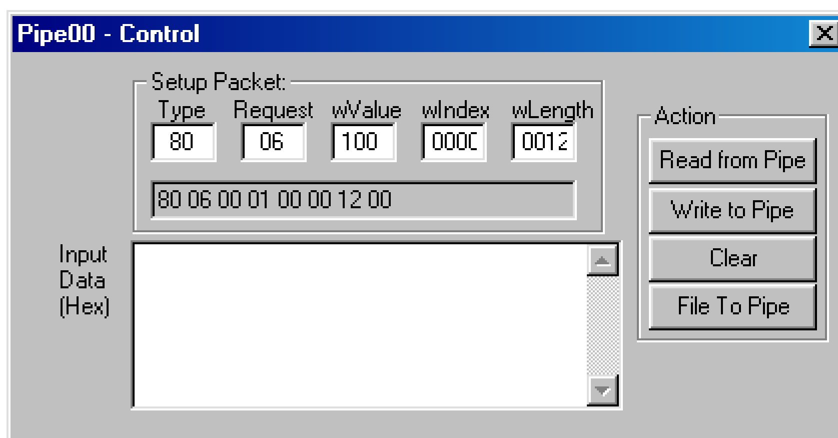
As defined in the USB specification, byte 0 indicates the length of the descriptor, bytes 2..3 contains the USB specification release number, byte 7 is the maximum packet size for endpoint 00, bytes 8..9 are the Vendor ID, bytes 10..11 are the Product ID, etc.

Control Transfers with DriverWizard

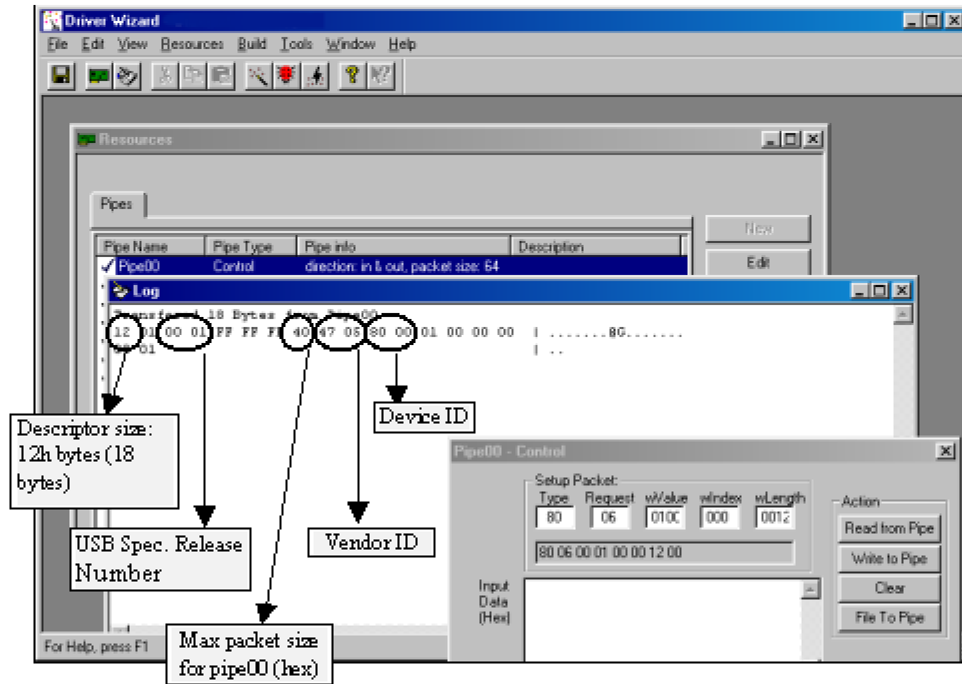
1. Choose Pipe00 and click on 'Read/Write to pipe'



2. Enter the required Setup Packet. For a 'Write' transaction that includes a data stage, enter the data in the 'Input Data' field. Click 'Read from Pipe' or 'Write to Pipe' according to the required transaction.



3. The 'Device Descriptor' data retrieved from the device can be seen in DriverWizard log screen: (See the following image)



Control Transfers with WinDriver API

To perform a read or write transaction on the control pipe, you can either use the API generated by DriverWizard for your hardware, or directly call WinDriver WD_UsbTransfer function within your application.

DriverWizard generates the functions below(the functions can be found in the MyDevice_lib.c source file). Fill the setup packet in the BYTE SetupPacket[8] array (an element in the WD_USB_TRANSFER structure) and call these functions to send setup packets on Pipe00 and to retrieve control and status data from the device:

```
DWORD MY_DEVICE_ReadPipe00(MY_DEVICE_HANDLE hMY_DEVICE,
        PVOID pBuffer, DWORD dwSize, CHAR setupPacket[8])
{
    WD_USB_TRANSFER transfer;
    DWORD i;
    BZERO(transfer);
    transfer.dwPipe = 0x00;
    transfer.dwBytes = dwSize;
    transfer.fRead = TRUE;
    for (i=0; i<8; i++)
        transfer.SetupPacket[i] = setupPacket[i];
    transfer.pBuffer = pBuffer;
    transfer.hDevice = hMY_DEVICE->hDevice;
    WD_UsbTransfer(hMY_DEVICE->hWD, &transfer);
    if (transfer.fOK)
        return transfer.dwBytesTransfered;
    return 0xffffffff;
}



DWORD MY_DEVICE_WritePipe00(MY_DEVICE_HANDLE hMY_DEVICE,
       PVOID pBuffer, DWORD dwSize, CHAR setupPacket[8])

    WD_USB_TRANSFER transfer;
    DWORD i;
    BZERO(transfer);
    transfer.dwPipe = 0x00;
    transfer.dwBytes = dwSize;
    for (i=0; i<8; i++)
        transfer.SetupPacket[i] = setupPacket[i];
    transfer.pBuffer = pBuffer;
```

```
    transfer.hDevice = hMY_DEVICE->hDevice;
    WD_UsbTransfer(hMY_DEVICE->hWD,&transfer);
    if (transfer.fOK)
        return transfer.dwBytesTransfered;
    return 0xffffffff;
```

For further information regarding WD_UsbTransfer, please refer to Chapter WinDriver Function Reference that illustrates the function reference for WinDriver in the WinDriver Manual.

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

The 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 - ReadPort 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 3 to port 0x301 etc (**WP_BYTE** - Write Port Byte).

A quick way to save 50% of the function call overhead would be to call **WD_Transfer()**with a **WP_SBYTE** (Write Port String Byte), with two bytes at a time. 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.

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.

Write your KP_INIT() function

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 library WD_KP.LIB
    CHAR cDriverName[9];      //  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 PlugIn. (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;
}
```

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()[KP_Close()]	Called when the User Mode application calls the WD_KernelPlugInClose() [WD_KernelPlugInClose()] Function
KP_Call()[KP_Call()]	Called when the User Mode application calls the WD_KernelPlugInCall() [WD_KernelPlugInCall()] function. This function is a message handler for your utility functions.
KP_IntEnable()[KP_IntEnable()]	Called when the User Mode application calls the WD_IntEnable()[WD_IntEnable()] function. This function should contain any initialization needed for your Kernel PlugIn interrupt handling
KP_IntDisable()[KP_IntDisable()]	Called when the User Mode application calls the WD_IntDisable() [WD_IntDisable()] function. This function should free any memory which was allocated in the KP_IntEnable() [KP_IntEnable()] callback.
KP_IntAtIrql()[KP_IntAtIrql()]	Called when WinDriver receives an interrupt. This is the function that will handle your interrupt in the Kernel Mode.
KP_IntAtDpc()[KP_IntAtDpc()]	Called if the KP_IntAtIrql()[KP_IntAtIrql()] callback has requested deferred 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_Kernel PlugInClose()***), 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;
}
```

Write the remaining PlugIn call-backs

Add your specific code inside the call backs routines.

Interrupt handling in 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:

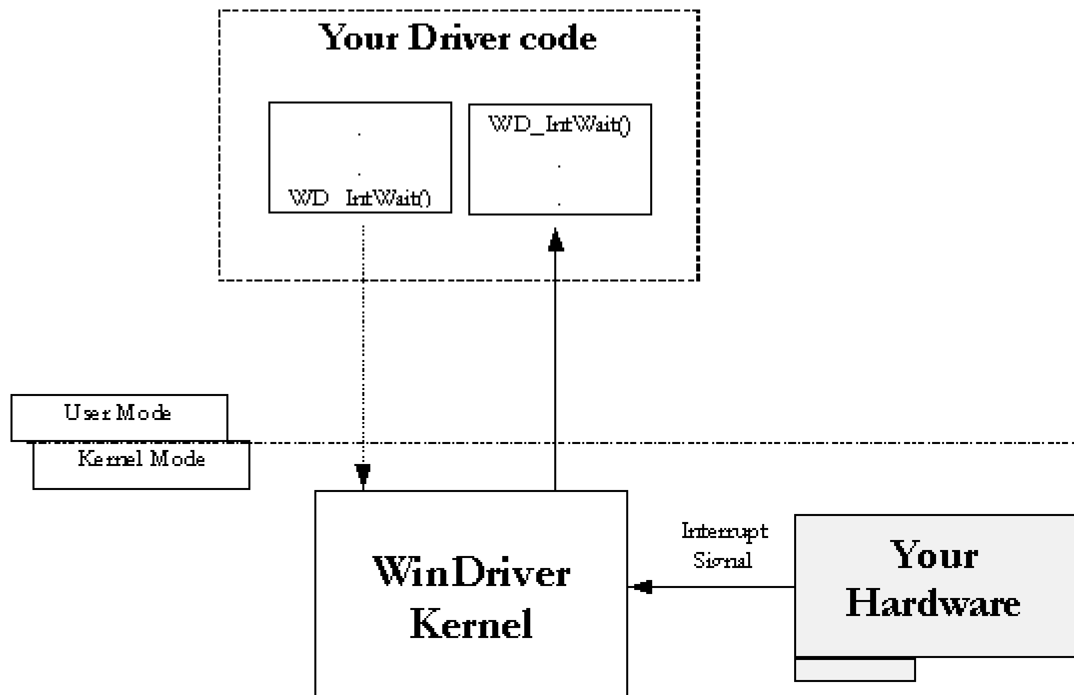


Figure 17.1: Interrupt Handling without Kernel PlugIn

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:

- You may only access non pageable memory.
  - You may only call the following functions:
    1. WD_Transfer()
    2. Specific DDK functions which are allowed to be called from an IRQL.
- You may not call malloc(), free(), or any WD_xxx command (other than WD_Transfer()).

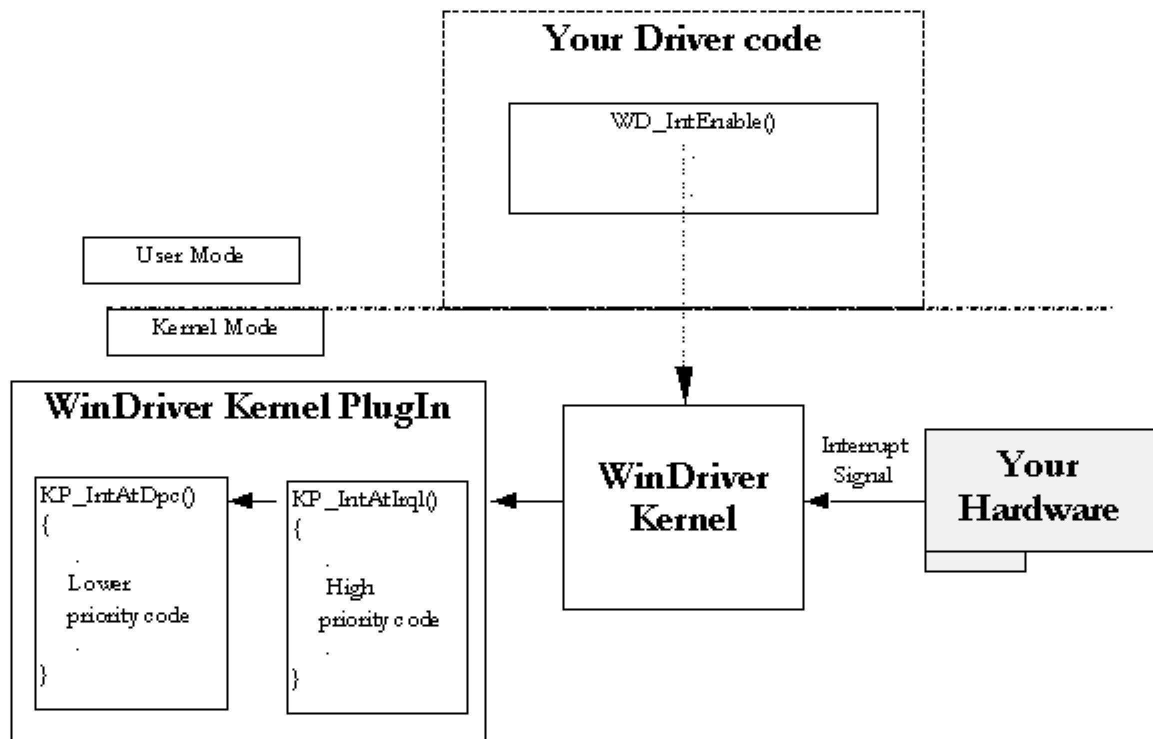


Figure 17.2: Interrupt Handling with the Kernel PlugIn

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.

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/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 all 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)

textbfkpCall- information on Kernel PlugIn to install as interrupt handler.

textbfkpCall.hKernelPlugIn- handle of Kernel PlugIn. if zero, then no Kernel PlugIn interrupt handler is installed.

textbfkpCall.dwMessage- message ID to pass to KP_IntEnable() callback.

textbfkpCall.pData- pointer to data to pass to KP_IntEnable() callback.

textbfkpCall.dwResult- value set by KP_IntEnable() callback.

For information about all other parameters of WD_IntEnable(), see the documentation of *WD_IntEnable()* in Chapter WinDriver Function Reference explaining the WinDriver functions.

RETURN VALUE

None

EXAMPLE

```
WD_INTERRUPT Intrp;

BZERO(Intrp); // from WD_CardRegister()

Intrp.hInterrupt = hInterrupt;

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");
```

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 with which KP_Close(), KP_Call() and KP_IntEnable() functions will be called. Use this to keep driver specific information.

RETURN VALUE

TRUE if successful. If FALSE, then the call to WD_KernelPlugInOpen() 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 memory here
    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 the memory allocated for pDrvContext here
}
```

KP_Call()

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

PROTOTYPE

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

PARAMETERS

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

kpCall - structure with information from WD_Kernel PlugInCall().

kpCall.dwMessage - message ID passed from WD_Kernel PlugInCall().

kpCall.pData - pointer to data passed from WD_Kernel PlugInCall().

kpCall.dwResult - value to return to WD_Kernel PlugInCall().

fIsKernelMode - this parameter is passed by the Windriver kernel.

RETURN VALUE

None

EXAMPLE

```
void _cdecl KP_Call(PVOID pDrvContext,
   WD_KERNEL_PLUGIN_CALL *kpCall, BOOL fIsKernelMode)
{
    kpCall->dwResult = MY_DRV_OK;
    switch ( kpCall->dwMessage )
    {
      // in this sample we implement a GetVersion message
    case MY_DRV_MSG_VERSION:
    {
      MY_DRV_VERSION *ver = (MY_DRV_VERSION *)
         kpCall->pData;
      COPY_TO_USER_OR_KERNEL(&ver->dwVer, &dwVer,
          sizeof(DWORD), fIsKernelMode);
      COPY_TO_USER_OR_KERNEL(ver->cVer, "My Driver V1.00",
          sizeof("My Driver V1.00")+1, fIsKernelMode);
      kpCall->dwResult = MY_DRV_OK;
```

```
    }
    break;
    // you can implement other messages here
    default:
      kpCall->dwResult = MY_DRV_NO_IMPL_MESSAGE;
    }
}
```

NOTES

The **fIsKernelMode** parameter is passed by the Windriver kernel to the KP_Call routine. The user need not do anything about this parameter. But notice how this parameter is passed to the macro COPY_TO_USER_OR_KERNEL -- this is required for the macro to function correctly. You may see the implementation of the macro COPY_TO_USER_OR_KERNEL in the header file **kpstdlib.h**, found under the directory *WinDriver\Include* of your WinDriver installation.

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 that handle 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_Int Disable(), KP_IntAtIrql() and KP_IntAtDpc() functions will be called with. Use this to keep interrupt specific information.

RETURN VALUE

Returns TRUE if enable is successful.

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_Int Enable().

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 DPC function is needed for execution.

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, then 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;

}
```

WD_KERNEL_PLUGIN

Defines a Kernel PlugIn open command.

Used by WD_KernelPlugInOpen()[WD_KernelPlugInOpen()] and WD_KernelPlugInClose() [WD_KernelPlugInClose()].

MEMBERS:

TYPE	NAME	DESCRIPTION
DWORD	hKernelPlug In	Handle to Kernel PlugIn

TYPE	NAME	DESCRIPTION
PCHAR	pcDriverName	Name of Kernel PlugIn driver. S no longer than 8 letters. Should include the VXD or SYS extensi
PCHAR	pcDriverPath	The directory and file name in w look for the KP driver. If NULL, t driver will be searched for in the windows system directory, unde supplied in pcDriverName, with added for Windows-95, or SYS Windows-NT
PVOID	pOpenData	Data to pass to KP_Open() callb Kernel PlugIn

WD_INTERRUPT

Used to describe an interrupt.

Used by the following functions: WD_IntEnable()[WD_IntEnable()], WD_Int Disable()[WD_IntDisable()], WD_IntWait()[WD_IntWait()], WD_IntCount()[WD_IntCount()].

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_KERNEL_ PLUGIN_CALL [WD_KERNEL_PLUGIN_CALL]	kpCall	The kpCall structure contains th to the Kernel PlugIn and other in which should be passed to the K mode interrupt handler when ins the handle is zero, then interrup installed without a Kernel PlugIn handler.

For information about all other members of WD_INTERRUPT, see the documentation of this structure in Chapter WinDriver Structure Reference that explains the WinDriver structures.

WD_KERNEL_PLUGIN_CALL

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

Used by WD_KernelPlugInCall()[WD_KernelPlugInCall()] and WD_IntEnable()[WD_IntEnable()].

MEMBERS:

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

KP_INIT

The KP_INIT structure is used by your KP_Init()[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()[WD_KernelPlugInOpen()].

MEMBERS:

TYPE	NAME	DESCRIPTION
DWORD	dwVerWD	Version of WinDriver library WD
CHAR	cDriver Name[9]	The device driver name, upto 8
KP_FUNC_OPEN	funcOpen	The KP_Open() Kernel Mode fu which WinDriver should call whe application calls WD_KernelPlu

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 the application calls the WD_Kernel PlugInCall 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_Kernel PlugInIntEnable(). 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.

TYPE	NAME	DESCRIPTION
KP_FUNC_CLOSE	funcClose	Name of your KP_Close() functi kernel.
KP_FUNC_CALL	funcCall	Name of your KP_Call() functior kernel.
KP_FUNC_INT_ENABLE	funcIntEnable	Name of your KP_IntEnable() fu the kernel.
KP_FUNC_INT_DISABLE	funcIntDisable	Name of your KP_IntDisable() fu the kernel.
KP_FUNC_INT_AT_IRQL	funcIntAtIrql	Name of your KP_IntAtIrql() func kernel.
KP_FUNC_INT_AT_DPC	FuncIntAtDpc	Name of your KP_IntAtDpc() fur the kernel.

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 is found in ***\windriver\util\WDREG.EXE***.

USAGE: WDREG [-vxd] [-name <driver name>] [-file <driver file name>]
[[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 the 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:

INSTALL - Creates and starts your driver (same as using WDREG CREATE START).

REMOVE - Unloads your driver from memory, and removes it from the 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:

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.

Dynamically loading your driver in your installation application:

Add the WDREG source code to your installation application.

The full source code for WDREG is found in ***\windriver\samples\wdreg***

For Linux and Solaris, use the command **wdreg <driver name>** to load and unload your driver. The parameters are the same as that of the Windows version.

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.

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/ME without an INF file.

3. In some cases the OS doesn't assign physical 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?

1. When no driver exists for your hardware, use DriverWizard to generate an INF file for your card / device (the INF file includes your device **VID/PID**and loads **WDPNP.SYS** as your device driver).

2. 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 Chapter The DriverWizard that explains the DriverWizard .

3. Go to: **Start | Settings | Control Panel |System**.

4. 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 2000
Windows 95/98/ME

WinDriver Kernel Module

Since Windrvr.o is a kernel module, it requires recompilation for every kernel version that it must be loaded on. To faciliate this, we supply the following components to insulate the windriver kernel module from the Linux kernel:

- **windrvr.a:** This is the compiled object code for the windriver kernel module
- **linux_wrappers.c/h:** These are the wrapper library source code that binds the windriver kernel module to the Linux kernel.

You need to distribute these components along with your driver source code or object code. We suggest that you adapt our makefile from the windriver/redist directory to compile and insert the module windrvr.o into the kernel. Note that this makefile calls the wdreg utility shell script that we supply under windriver/util. You should understand how this works and adapt it for your own needs.

Your User Mode Driver

Since the user mode driver does not have to matched against the kernel version number, you are free to distribute it as binary code (in case you wish to protect your source code from unauthorized copying), or as source code.

Kernel Plugin Modules

Since the kernel plugin module is a kernel module, it also needs to be matched against the active kernel's version number. This means recompilation for the target system. It is advisable to supply the kernel plugin module source code to clients so that they may recompile it. You may also use the same makefile, to build and insert any kernel plugin modules that you distribute with Linux, that you use to recompile and install the WinDriver kernel module.

Installation script

We suggest that you supply an installation shell script that copies your driver executables to the correct places (perhaps /usr/local/bin), then invoke make or gmake to build and install the windriver kernel module and any kernel plugin modules.

Installation script

We suggest that you supply an installation shell script that copies your driver executables to the correct places (perhaps /usr/local/bin), then install the windriver kernel and any kernel plugin modules. You may adapt the utility scripts wdreg and install_windrvr, that we supply under the directory windriver/util for your purpose.

Layered Drivers

Layered drivers are device drivers that are part of a "stack" of device drivers, that together process an I/O request. An example of a layered driver is a driver that intercepts calls to the disk, and encrypts / decrypts all data being 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.

Layered drivers are sometimes also known as filter drivers. These are also supported on Windows 95/98/ME.

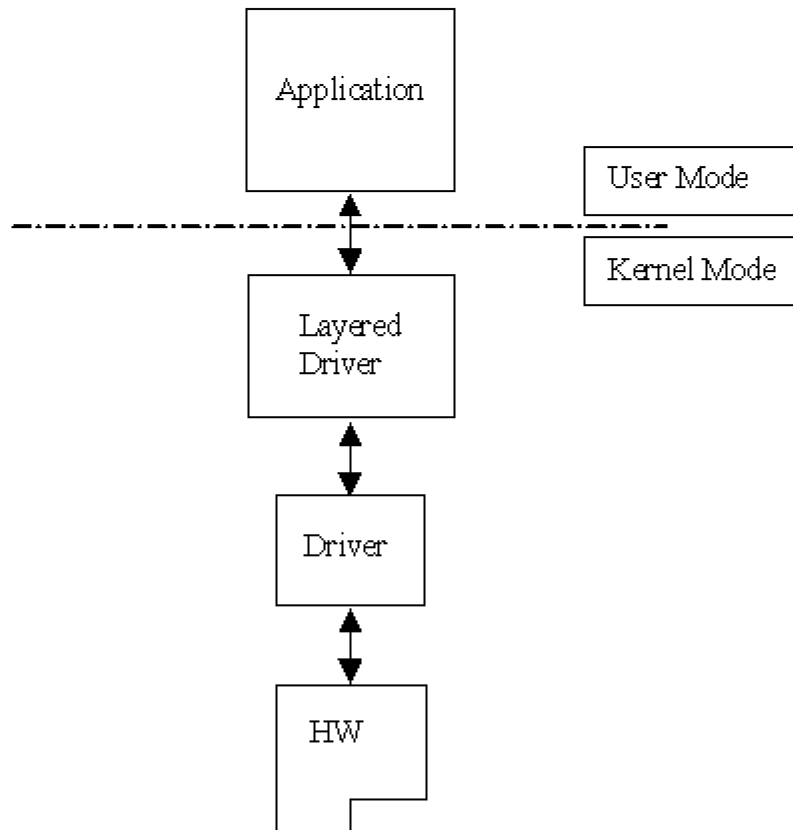


Figure 1.3: Layered Drivers

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.

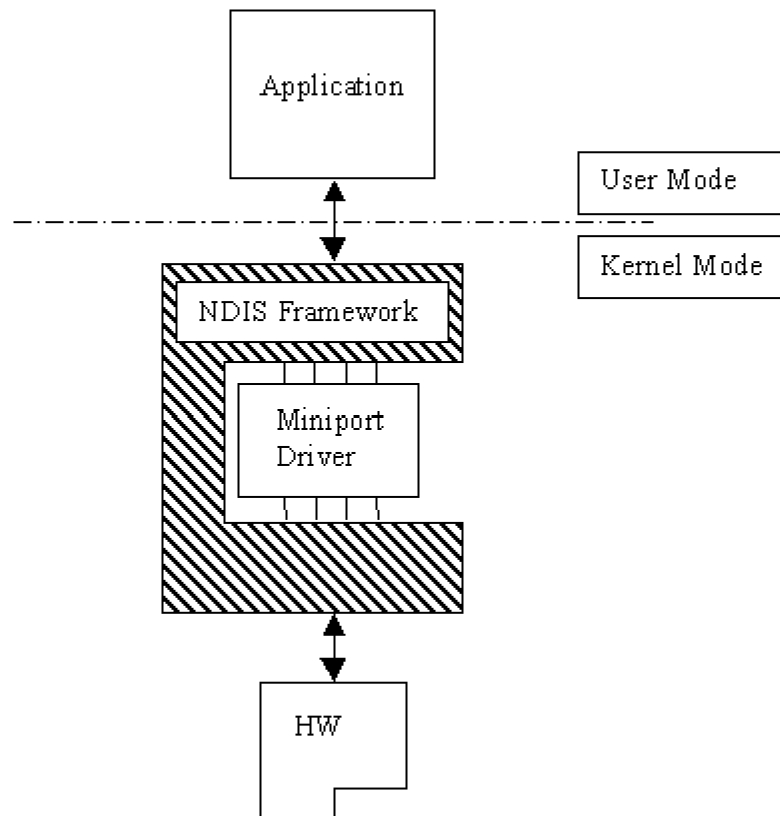


Figure 1.4: Miniport Drivers

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

An example of Miniport drivers is the "NDIS" miniport driver. The NDIS miniport framework is used to create network drivers that hook up to 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.

Installing WinDriver CE for a hand held PC

Insert the WinDriver CD into your NT machine CD drive.

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).

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

Use the Windows CE Remote Registry Editor tool(ceregedt.exe) or the Pocket Registry Editor(pregedt.exe) 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.

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.

Compile and run the sample programs (see Section Checking Your Installation that describes how to check your installation) to make sure that WinDriver CE is loaded and is functioning correctly.

Using WinDriver CE with ETK/Platform Builder

- 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:

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

- 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.

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

- 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.

- 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.

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

- Compile and run the sample programs (see Section Checking Your Installation that describes how to check your installation) to make sure that WinDriver CE is loaded and is functioning correctly.

Emulation on Windows NT

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

2. Select the X86EMU target and then compile and run one of the sample programs to make sure that it works correctly.

Restricting hardware access on Linux

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

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

Restricting hardware access on Solaris

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

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

Solaris Platform Specific Issues

WinDriver for Solaris supports version 2.6, 7.0 and 8.0 on Intel X86 and Sparc. The same WinDriver based hardware access code will run on both platforms after recompilation.

WinDriver does not support Solaris 7 64 bit kernel. To switch from a 64 bit kernel to a 32 bit kernel follow these simple steps:

1. Reboot the computer (as super user)-#reboot

2. When the computer resets, Break into the boot prompt by pressing **STOP+A**.

3. At the prompt enter the following: (*boot kernel/unix*)

4. To make the 32 bit kernel to be the default one, enter the following at the boot prompt: (*setenv boot-file kernel/unix*)

DebugMonitor Graphical Mode

Applicable for Windows 95, 98, ME, NT, 2000. You may also use DebugMonitor to debug your CE driver code running on CE emulation on Windows NT. For Linux, Solaris, VxWorks and CE targets use the console mode DebugMonitor.

- Start DebugMonitor from the **Start | Programs |WinDriver | MonitorDebugMessages** menu.

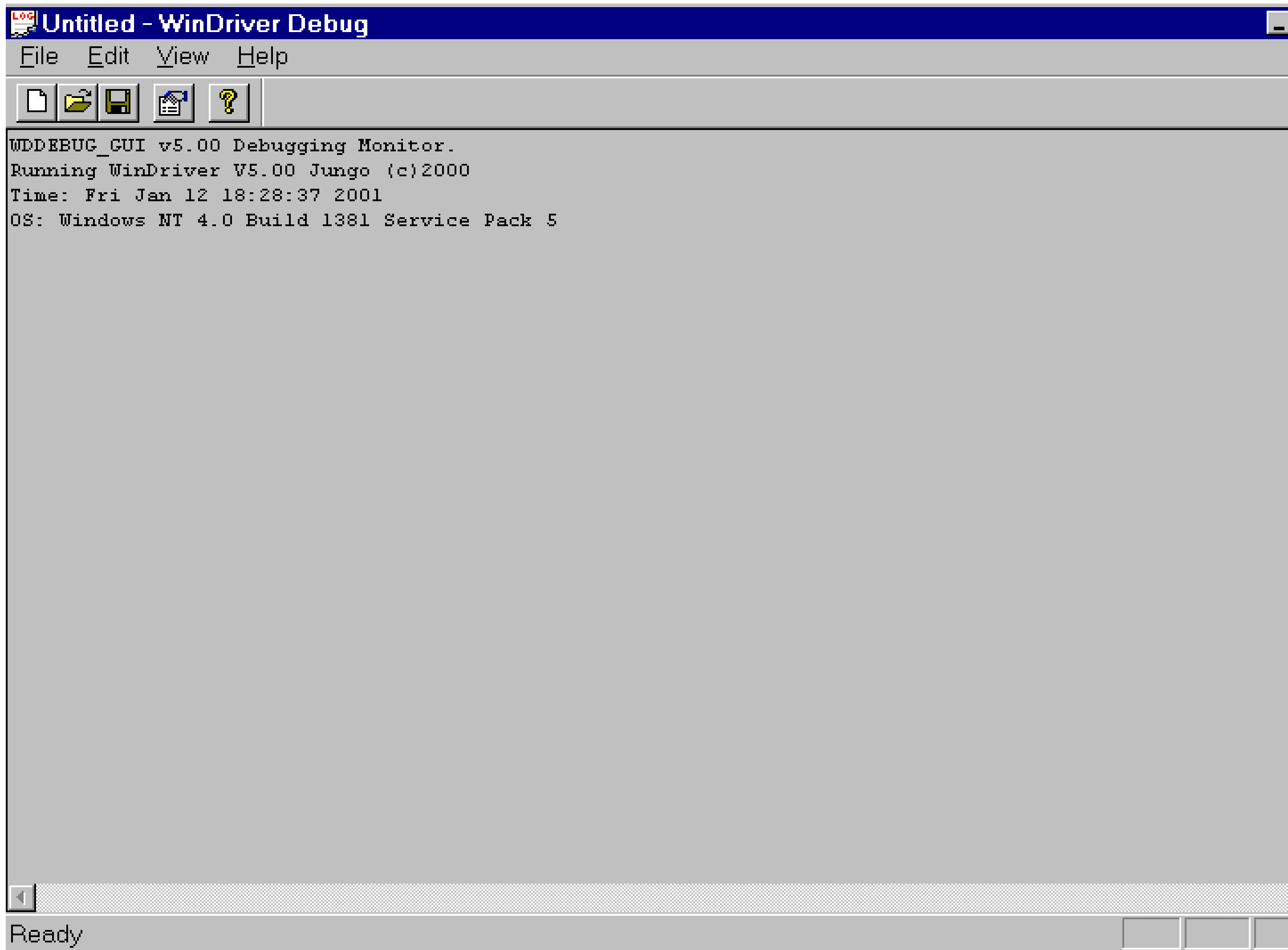


Figure 6.1: Start DebugMonitor

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

Figure 6.2: Set Trace Options

Status - Set trace on or off.

Section - Choose what part of the WinDriver API you 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.

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 run "**wddebug**" in 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 appears, describing all the different options for this command.

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

DebugMonitor on Linux and Solaris

On Linux and Solaris, DebugMonitor is only available in console mode. Its usage is therefore as described in Section DebugMonitor - Console Mode. However, you can also start it from DriverWizard GUI using the menu selection *Tools* | *Debug Monitor*. This starts up seperate Xterm window with the command line verison of *wddebug* running inside it.

DebugMonitor on Windows CE

On Windows CE, DebugMonitor is only available in console mode. Its usage is therefore as described in Section DebugMonitor - Console Mode.   You first need to start a Windows CE command window (CMD.EXE) on the Windows CE target computer and then run the program WDDEBUG.EXE inside this shell.

DebugMonitor on VxWorks

On VxWorks, DebugMonitor is only available in console mode. Its usage is therefore as described in Section DebugMonitor - Console Mode. However, because of the special syntax of the Tornado WindShell, we show a sample session with Tornado II IDE below, where we first load the debug monitor, then set the options and then run it to capture information.

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

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

Please note the following:

- The DebugMonitor object binary module is called wddebug.out.
- The main program entry point is called wddebug_main
- The arguments are enclosed in double quotations, and seperated by commas. This syntax is required by *WindShell*

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);
}
```

Windows 2000

1. Use DriverWizard to generate an INF file for your card / device (the INF file includes your device VID/PID and loads WDPNP.SYS as your device driver).

2. 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 Chapter The DriverWizard that explains the DriverWizard.

3. Go to: **Start | Settings | Control Panel |System**.

4. Click the '**Device Manager**' option in the "**Hardware**" tab.

5. Now select the **View Devices by connection** option.

6. For PCI cards, navigate to **Standard PC | PCI bus |<your_card>.**

7. For USB devices: navigate to **Standard PC | PCI bus |PCI to USB Universal Host Controller | USB Root Hub |<the current driver for the device>**.

8. Select **Action | Properties**.

9. Click the '**Update Driver**' button in the "**Driver**" tab.

10. In DriverWizard: click on the '**Next**' button.

11. Select "**Search for a suitable driver for my device**" option by clicking on the appropriate radio button.

12. Click the "**Next**" button

13. Check the "**only Specify a location**" check-box

14. Click the "**Next**" button

15. Enter the path: **c:\temp\<your .INF file>**in the input box that is displayed.

16. Click the "**OK**" button

17. Click the "**Next**" button

18. Click the "**Finish**" button

19. Reboot

Windows 95/98/ME

1. Use DriverWizard to generate an INF file for your card / device (the INF file includes your device VID/PID and loads WDPNP.SYS as your device driver).
2. 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 Chapter The DriverWizard that explains the DriverWizard.
3. Go to: **Start | Settings | Control Panel |System**.
4. Now select the **View Devices by connection** option.
5. For PCI cards, navigate to **Standard PC | PCI bus |<your_card>.**
6. For USB devices: navigate to **Standard PC | PCI bus |PCI to USB Universal Host Controller | USB Root Hub |<the current driver for the device>**.
7. Select **Properties**.
8. Click the '**Update Driver**' button in the "**Driver**" tab.
9. Click the "**Next**" button
10. Check the "**only Specify a location**" check-box
11. Click the "**Next**" button
12. Enter the path: **c:\temp\<your .INF file>**in the input box that is displayed.
13. Click the "**OK**" button
14. Click the "**Next**" button
15. Click the "**Finish**" button
16. Reboot