KernelDriver V5 Developer's Guide

COPYRIGHT

Overview

In this chapter you will learn the basics of kernel mode driver development, and learn the basic steps of creating your 95 / 98 / NT / 2000 and Linux drivers.

Introduction
Device Driver Overview
Matching the right tool for your driver
How does KernelDriver make it easy?
What is included in the package?
Before you begin

Installing KernelDriver

System Requirements
Installing KernelDriver
Testing the installation
How do I uninstall KernelDriver?

KernelDriver development process

Developing a driver: DriverWizard vs. Samples
How to Write a Driver from Scratch
Building Your Driver
WDREG - Dynamically loading and unloading your driver
KernelDriver USB

The DriverWizard

DriverWizard - An Overview
DriverWizard Walkthrough
DriverWizard Notes

KernelDriver Class Model

Read through this chapter to gain an understanding of the classes that make up KernelDriver, and how they relate to each other.

Overview
Where it all begins - The DriverEntry() Function
Communicating with Drivers - IRPs and IOCTLs
Classes in brief

Elements of a basic driver

This chapter takes you through the structural elements of a device driver. After reading this chapter, you will know how to build a simple device driver.

KdDriver, KdDevice and KdIrp

Classes for Accessing Device Memory

This section shows you how to access your device's memory and I/O spaces. After reading this chapter, you will know how to read and write from your hardware.

Part of the CPU's address space is allocated for external devices. This address space is shared between several buses. Some of these are the PCI bus, ISA bus, VME bus, etc. On any particular bus, devices have an address which is relative to the bus they are connected to. The bus bridges (see figure below) translate the bus-relative addresses to the CPU's address space.

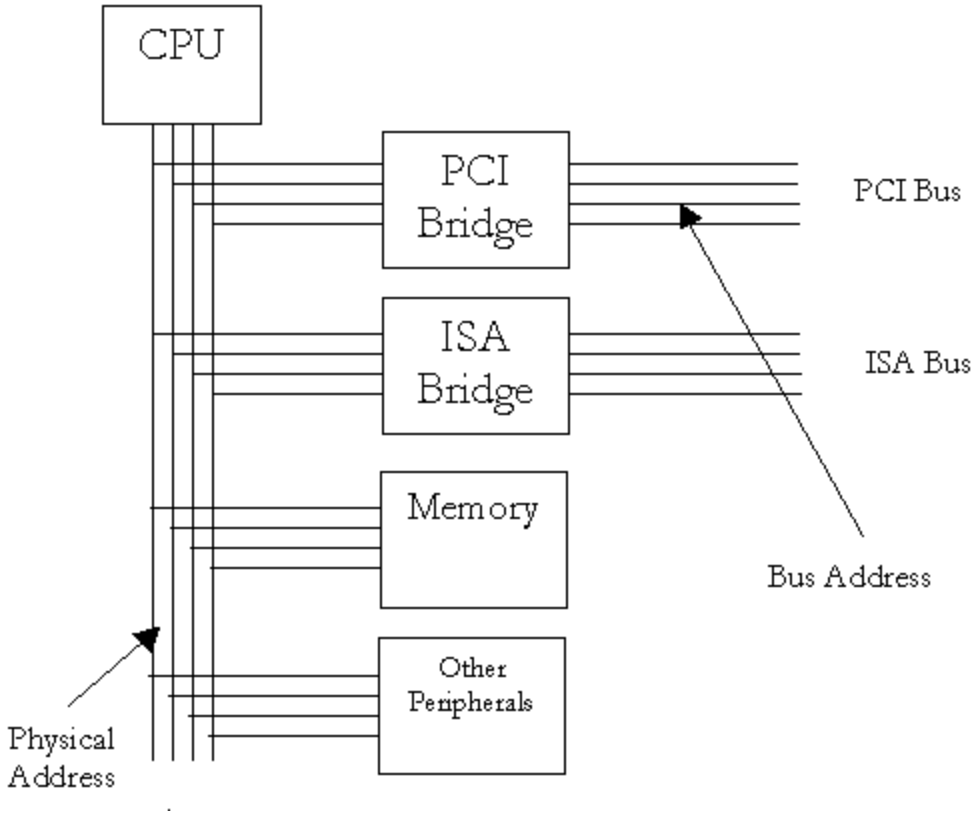

Figure 7.1: Translating bus adresses to CPU addresses

As an example, consider two different devices on two different busses. Both devices can own the same memory space on their respective busses. Their bus bridges will map the devices to different CPU address locations, thereby allowing the CPU to access each device separately.

Plug-n-Play
I/O space Vs. Memory space
Class KdBusAddress
Class KdMapKernel
Class KdMapProcess
Quick Sample ... How to Map and Report Bus Addresses

Classes for Handling Interrupts

This chapter explains how interrupts are handled by the NT operating system, and shows you how to write the code which handles your hardware's interrupts. After reading this chapter, you will know how to handle interrupts by creating interrupt service routines (ISRs) and a deferred procedure calls (DPCs).

Interrupt Handling
Implementing an Interrupt
Class KdIrq
Class KdDpc
Quick Sample ... How to Handle Interrupts

Classes for Registry and Resource Reporting

These classes enable you to easily read from / write to the registry, and report driver resources.

Windows NT Registry
Class KdRegistry
Class KdResources
Quick Sample ... How to Read and Register Resources

Classes for Synchronization

This section defines the synchronization utilities that KernelDriver provides.

When a data buffer is shared between two threads, there is a potentially dangerous situation of data integrity. KernelDriver offers several synchronization objects for solving these issues.

Class KdSyncObject
Class KdEvent
Class KdSpinLock
Class KdMutex
Class KdSemaphore
Class KdTimer
Class KdTimedCallback
Class KdThread

Utility Classes

This section lists the different utilities that KernelDriver provides. The utilities encapsulate activities that are common in device drivers.

Class KdString
Class KdList
Class KdIrpList
Class KdFifo
Class KdFile
Class KdPhysAddr
Class KdMem
DebugDump utility
Memory Compare Utilities
Memory Allocation Utilities

Classes for Layered Drivers

This section explains what layering devices are, and how KernelDriver enables you to create layered drivers. This section also shows how to use KernelDriver objects for communicating between drivers.

Layered drivers (or Filter 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 which intercepts calls to the disk, and encrypts / decrypts all data being written / read from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption / decryption.

Layered drivers use IRPs (I/O request packets) to communicate with the other drivers in the stack, and may communicate with the application using IOCTLs.

KernelDriver provides theKdFilterDevice class for building the filter (top layer) driver, and the KdLowerDevice class for representing the driver which is connected to the filter driver on its 'bottom edge'.

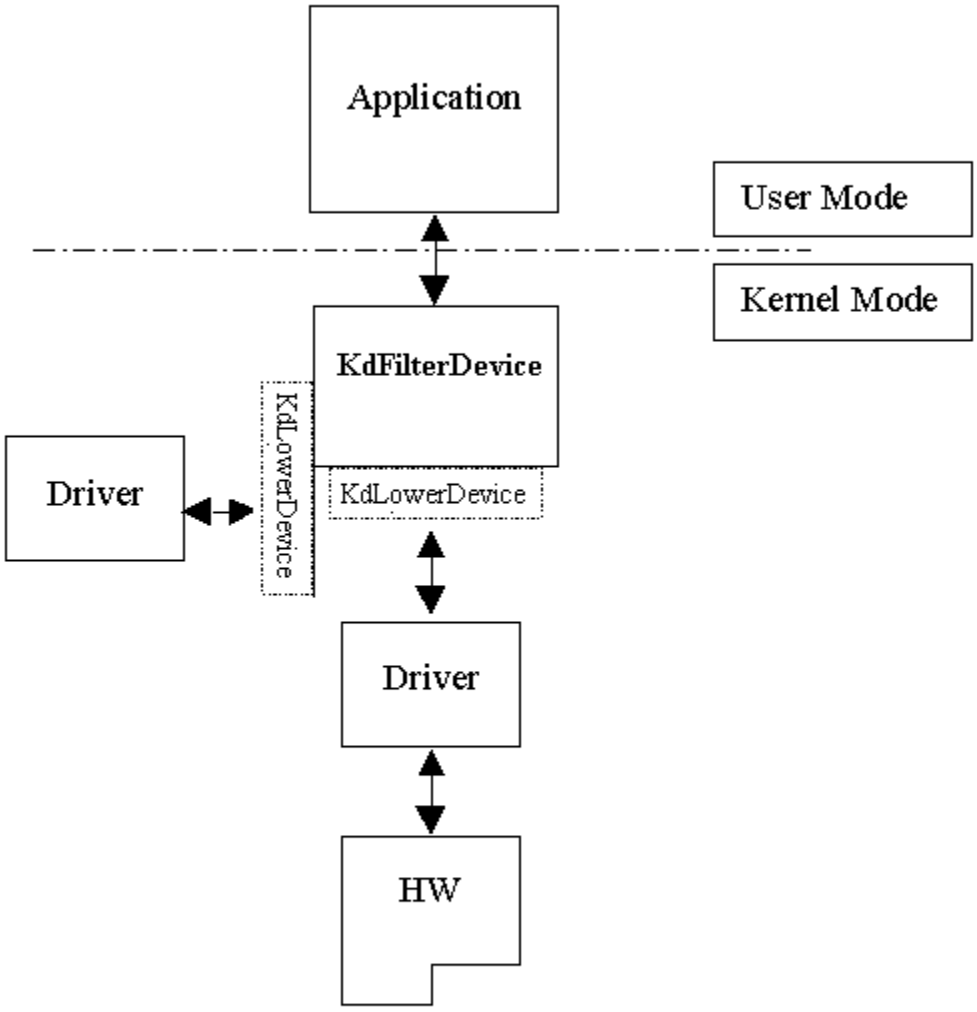

Figure 12.1: Classes for Layered Drivers

Drivers can also communicate with other existing drivers by forwarding an IRP to them, or by creating a new IRP and sending it to the other driver. This can be done using the KdLower object as well.

Class KdFilterDevice
Class KdLowerDevice

Classes for NDIS Drivers

This chapter provides an overview of the architecture of NDIS Miniport drivers, and shows the KernelDriver objects which encapsulate the NDIS functionality.

Overview
The KdNdisMiniport class
The KdNdisAdapter class
The KdNdisConfiguration class

Debugging your driver

This chapter helps you prepare your environment for debugging and introduces you to the driver debugging process.

Overview
Using KernelTracer
KernelTracer - Graphical mode
KernelTracer - Console Mode
Using WinDbg
Establishing a WinDbg Debugging Session
WinDbg Command-Line Options
Debugging a Crash Dump

KernelDriver Class Architecture

Inheritance Chart
Dependency Chart

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

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.

License Agreement

SOFTWARE LICENSE AGREEMENT OF KernelDriver V5.0

Jungo © 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, nonexclusive "one-user" license to use the Software 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 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 one copy of the Software on a single computer 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 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 the Software allows you for royalty free distribution of WINDRVR.SYS and WINDRVR.VXD files only when complying with sections **5a, 5b, 5c** and **5d** of this agreement.

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 WINDRVR.SYS or WINDRVR.VXD files.

5d. The Software may not be used to develop a development product, 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 distributed with the Software provided that they be used only by developers 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 THE 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 Licensor for any reason, please write:

Jungo © 2001

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)

Address:

Jungo Ltd,

P.O.Box 8493,

Netanya 42504,

ISRAEL.

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 subparagraphs (c)(1)(ii) and (2) of Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable.

Technical Support

Please feel free to contact our technical team with any questions you may have. Our technical support personnel are generally the same developers who wrote the product, so you can expect to receive excellent support from engineers who are highly qualified in device driver development.

Contact Jungo support via phone, email or fax at:

Phone: 1-877-514-0537(USA) +972-9-8859365 (Worldwide)

Fax: 1-877-514-0538(USA) +972-9-8859366 (Worldwide)

Email: :support@jungo.com

Using a Serial Cable for Kernel Debugging (Null Modem Serial Cable)

To create a cable for kernel debugging, connect RXD to TXD, TXD to RXD, and GND to GND. The following table lists the pin numbers of these signals for 9-pin and 25-pin connectors.

Signal	Pin Numbers for 9-pin connector	Pin Numbers for 25-pin connector
RXD (Receive Data)	2	3
TXD (Transmit Data)	3	2
GND (Signal Ground)	5	7

Once a connection has been made using a cable as described above, the connection should then be tested using an available serial communications package (e.g. terminal.exe). The communication application should be started on both the target and the host. Type a few keystrokes from both machines to make sure that the data is received on both ends.

Other resources

For an updated list of kernel device driver development resources, please check
http://www.jungo.com/windrv/resources.html

Introduction

Developing kernel mode drivers is a time consuming task with a steep learning curve. It requires you to master your operating systems' architecture, learn the relevant device driver development environment, and debug your software in the kernel mode, where new debugging software and techniques must be learned.

Jungo provides driver developers with tools that dramatically ease the development process, providing much faster time to market and code which is faster and cleaner.

KernelDriver is a generic cross-platform kernel mode development toolkit. In addition, it has special features that apply only to Windows NT. The C++ class library of KernelDriver can currently only be used with Windows NT/2000. The source code of these libraries are also available, but are only guaranteed to compile and work correctly with Windows NT and Windows 2000. Thus, KernelDriver Version 5.0 fully supports the development of Windows NT style (.SYS) drivers, which can be used on Windows NT and Windows 2000. KernelDriver version 5.0 does not as yet, support WDM driver development.

The first chapter of this manual provides an overview on kernel mode device drivers and on how to build them with KernelDriver. The later chapters will go into the details of every step of the development process - from creating your driver code with DriverWizard, through debugging your code, to how to package and ship your driver.

This product is designed to help you quickly create the quality driver you need. If you run into problems at any time, we urge you to contact our technical support at `:support@jungo.com`, or (USA Toll Free) **1-877-514-0537**(World wide) **972-9-8859365**, or to contact your nearest Jungo reseller (see the complete reseller list at `http://www.jungo.com/resellers.html`)

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.

How does KernelDriver make it easy?

One of the problems often encountered by driver developers, is that the hardware they receive has not been debugged. **DriverWizard** (included), helps you verify the hardware by providing you with a friendly graphical interface for 'peeking' and 'poking' your hardware. DriverWizard automatically detects your hardware and its resources. You can add descriptions of your hardware (e.g. registers), and use these to diagnose your hardware. After making sure that the hardware is working as expected, DriverWizard will automatically generate the driver code for you, along with a sample application that communicates with this driver to talk to your hardware. After compiling and running this driver and application, you can modify them to your needs.

KernelDriver generates driver frameworks for Linux (kernel modules), Windows (VxD drivers) and Windows NT model drivers (SYS drivers). It also provides a C++ class library for Windows NT model drivers that eases your driver development process by abstracting many of the driver details, and providing classes for common device driver tasks. These classes encapsulate thousands of lines of kernel code, enabling you to focus on your driver's added-value functionality and not on Windows NT internals.

KernelDriver also includes many samples, written in portable C and C++ (the C++ samples use the KernelDriver class library) to implement different types of drivers. You can jump-start your driver development by finding a sample that implements the main functionality of your driver, and using it as the starting point for your driver.

KernelDriver includes the **Kernel Tracer** tool, which helps you in collecting debugging information about your driver in real time

What is included in the package?

The CD that is included with the KernelDriver package includes all of Jungo's driver tools. All tools are available on the CD for a free 30 day evaluation. Along with the CD, you will find your license string. This software key 'unlocks' the products that you have purchased. After evaluating the other Jungo products, you may order additional licenses from Jungo or from your nearest Jungo reseller `http://www.jungo.com/resellers.html`

The KernelDriver package includes the following:

- **KernelDriver classes and libraries:** These are the classes with which you will build your driver. The source code of the class library is also available.

- **DriverWizard:** The tool you will use to diagnose your hardware, and to generate the source code of your driver.

- **Samples:** The 'Samples' directory includes many different compilable working drivers, all written with the KernelDriver classes. Use the relevant driver sample as a starting point for your driver.

- **Driver Tracer:** Use this tool to collect runtime information about your driver.

- **Technical Support:** Jungo's technical support will help you complete your project in record time. We urge you to contact Jungo's technical support with any questions you may have at `:support@jungo.com`, or via our toll free numbers).

- **This manual.**

Before you begin

The KernelDriver toolkit assumes that the developer has a fair understanding of C++.

Reading the KernelDriver manual will provide enough information to create, compile and debug your driver. In certain cases, deeper knowledge of NT internals may be needed. In this case, check out Jungo's site `http://www.jungo.com` for information, and a list of recommended NT Kernel books.

System Requirements

You must have the following software installed in order to use KernelDriver to create your driver:
Windows
Linux

Installing KernelDriver

This section covers the installation of KernelDriver on all supported operating systems.

Installing KernelDriver On Windows 95/98/ME/NT/2000
Installing KernelDriver on Linux

Testing the installation

It is important to check that all of the needed elements are properly installed. To check the installation, follow the instructions below. They will guide you in compiling and running a simple sample driver:

Testing the installation under Windows NT/2000

Testing the Installation under Windows 95/98/ME

Troubleshooting on Windows

Testing the installation under Linux

Troubleshooting on Linux

How do I uninstall KernelDriver?

If for some reason, you need to uninstall the evaluation or registered version of KernelDriver, please follow the instructions in this section.

Uninstalling KernelDriver from Windows 95/98/ME/NT/2000

Uninstalling KernelDriver from Linux

Developing a driver: DriverWizard vs. Samples

The two recommended ways of writing your device drivers are by either using DriverWizard to generate your driver code, or to use an existing sample as the basis for your device driver.

- If you are writing a monolithic driver that accesses hardware - Use DriverWizard to diagnose your hardware and generate your code. DriverWizard can generate frameworks and sample test applications for Windows VxDs, Windows NT model drivers (SYS), and Linux kernel modules.

- If you are writing NT model drivers, and the generated code does not fit your model, then choose a sample which most closely resembles the driver you need to write, and modify it to suit your needs.

Using DriverWizard to create your driver

Using a Sample to create your driver

How to Write a Driver from Scratch

Another alternative method of writing your driver is to write it from scratch. Although this is not the recommended way to work (it is faster to take a sample from ***kerneldriver*****samples** or to use DriverWizard to generate the skeletal code), this overview will provide a good insight on how device drivers should be written with KernelDriver.

Writing a Windows NT Model Driver

Writing drivers for other operating systems

Writing a Windows NT Model driver without using WinDriver

Building Your Driver

This section covers the process of building your driver under all supported operating systems.

Compile the Driver

Installing, Loading and Registering Your Driver

Running Your Driver

WDREG - Dynamically loading and unloading your driver

KernelDriver provides a utility for dynamically loading and unloading your driver called ***WDREG.EXE***

Usage under Windows
Usage under Linux
Using WDREG from within your application

KernelDriver USB

Introduction
Advantages
Building a Kernel Mode USB Driver

DriverWizard - An Overview

DriverWizard (included in the KernelDriver 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 an API for accessing all your hardware resources including custom defined registers. This API is implemented by DriverWizard by calling the WinDriver API.

For example, WinDriver's API contains a function called **WD_Transfer()**for exchanging data with your hardware. DriverWizard might generate a more specific function such as **MyCard_ReadStatusRegister()** (where 'status register' is a register you have defined on your hardware).

DriverWizard can generate user mode code (choose the '**WinDriver**' option when generating the code), or kernel mode code (choose the '**KernelDriver**' option when generating the code). In either case, DriverWizard will generate an API specific to your hardware (either user mode API that you can call from your user mode application or kernel mode API that you can call from your kernel mode application).

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

1. After the hardware has been built, insert the hardware into the PCI/PCMCIA/ISA/ ISA PnP/EISA/CompactPCI bus, and use DriverWizard to check that the hardware is performing as expected.

2. 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:

For User Mode Drivers

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

2. A Win32 diagnostics program, in console mode with which you can diagnose your card. This application utilizes the special library functions, (described above), which were created for your card by DriverWizard. Use this diagnostics program as your skeletal device driver.

3. A project workspace which you can use to automatically load all of the above project information into your Microsoft Developer's Studio (Version 5.0 and up).

4. The source code generated is portable between all platforms that WinDriver supports, including Windows 95, 98, ME, 2000, CE and Linux.

For Kernel Mode Drivers

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

2. Kernel Mode device driver. Use this source code as your skeletal device driver.

3. A Win32 diagnostics program, in console mode, with which you can diagnose your card. This application calls the kernel mode device driver that DriverWizard created.

4. A project workspace which you can use to automatically load all of the above project information into your Microsoft Developer's Studio (Version 5.0 and up).

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/USB).

2. Run DriverWizard.

    - Click *Start* | *WinDriver* | *DriverWizard*from the start menu or double click 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).

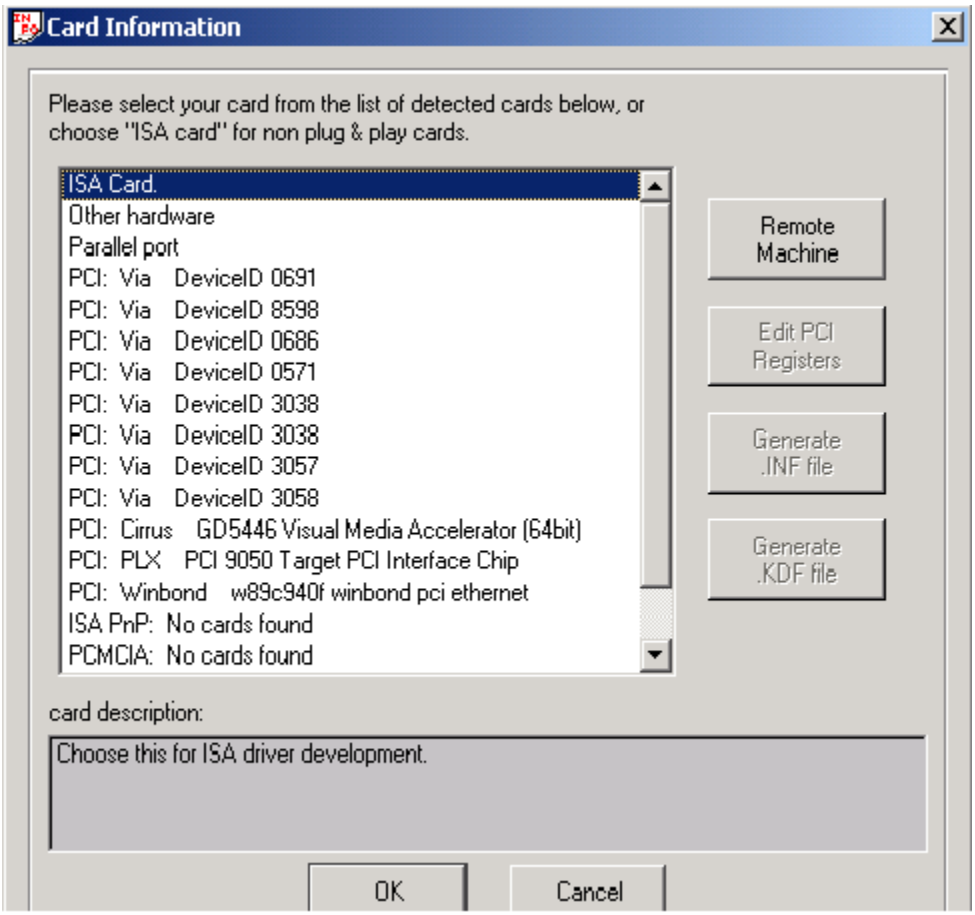

Figure 4.1: Selection of PnP Device

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: The wizard reads all the interface and alternate settings of the supported devices and displays them. For USB devices that have only one interface configured, the wizard automatically selects the detected interface and the 'interface selection' screen will not be displayed.

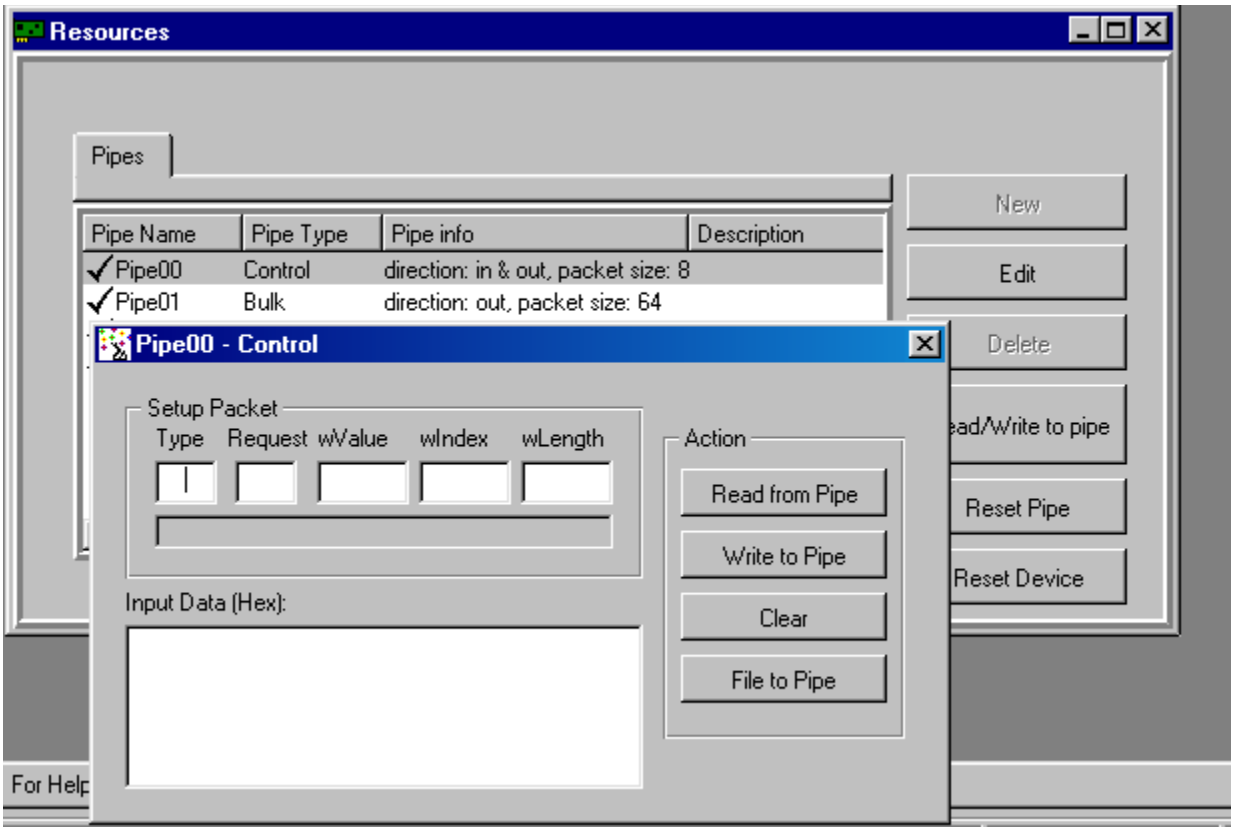

Figure 4.2: KernelDriver USB Device Configuration

4. Diagnose your device

   - Test your card's I/O, memory ranges, registers and interrupts.
   - Test the pipes of the USB device.
   - 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.

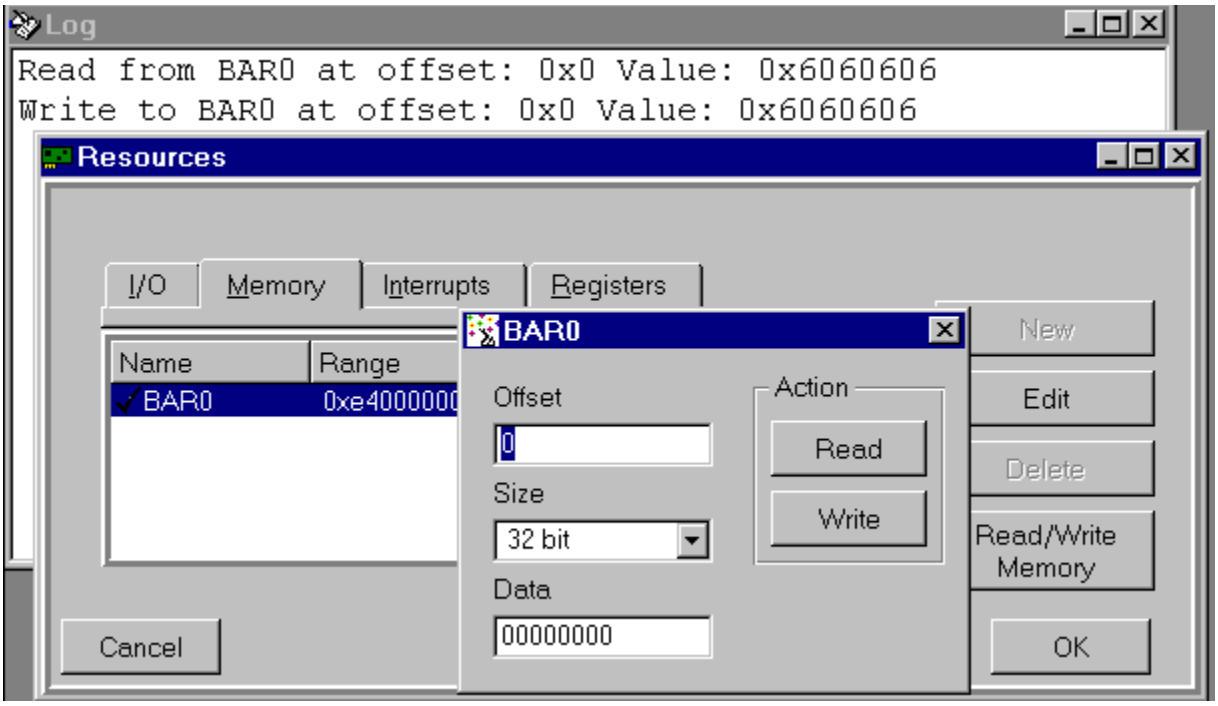

Figure 4.3: A PCI Diagnostics Screen

- 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 and here 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 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 that the 'Transfer failed'.

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

- For output pipe (host to device) - click '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.

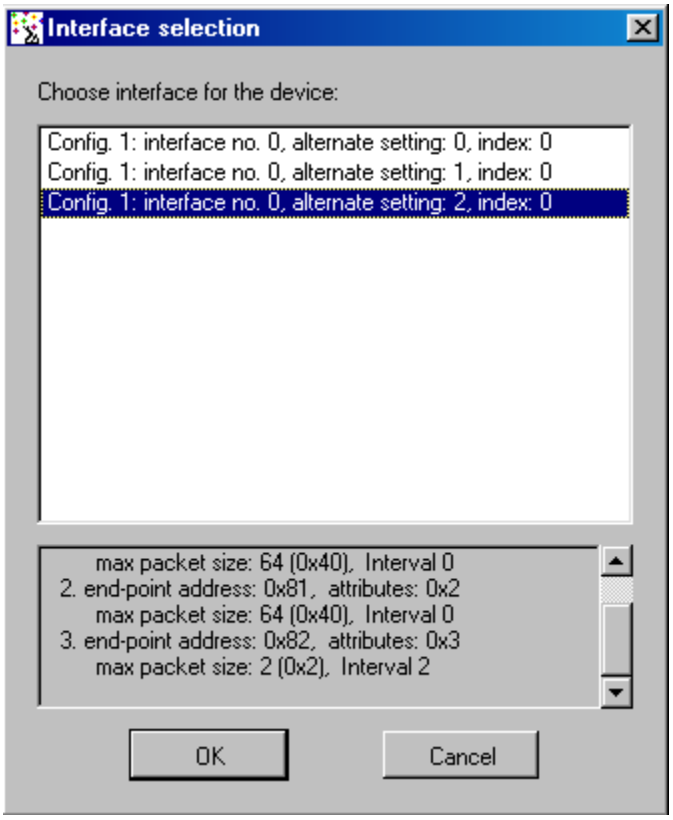

Figure 4.4: USB Diagnostics Screen

5. Let DriverWizard generate skeletal code for you.

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

Figure 4.5: Generate Code Options

- Select **KernelDriver** from the 'Choose type of driver' screen to generate full kernel mode drivers. Selecting the KernelDriver option will generate source code for user mode drivers. Click *Next* to continue.

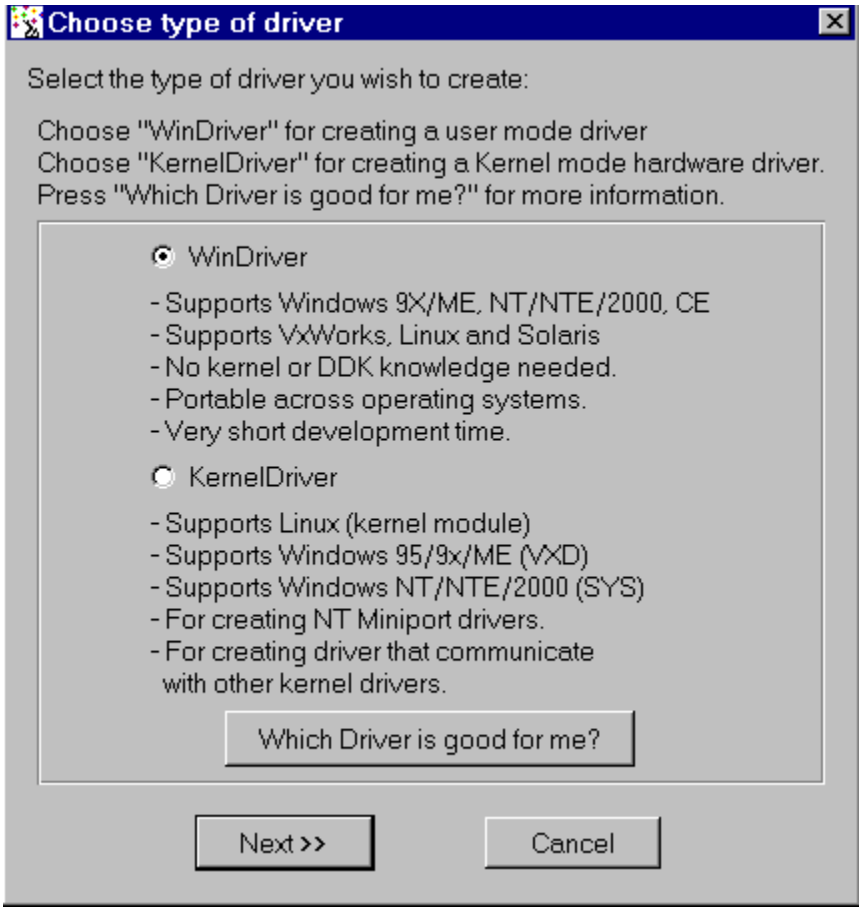

Figure 4.6: Generate Code dialog

- In the following dialog, choose your desired target operating systems.

Figure 4.7: Choosing target OS for Kernel Mode Driver Project

- Note the following:

1. You can choose between a Windows VxD driver (for 95/98/ME), a Windows NT SYS driver (for Windows NT/NTE/2000) and a Linux kernel module.

2. You can choose whether your NT model driver (SYS) is generated in C or C++. C drivers resemble the VxD and Linux drivers at the source code level; they call the WinDriver API as well. The C++ drivers use the KernelDriver C++ Kernel mode class library for Windows NT Model drivers.

3. DriverWizard generates a KDF file (required by Window NT Embedded) if you select the*WinNT Embedded* option. You should also have checked either the *WinNT C* or*WinNT C++* options.

- If the KernelDriver base installation directory shown in the dialog does not reflect the correct value, or you have a different installation you want to use, you should edit this field. Another reason to edit this field is that you are using the Windows GUI to generate code for Linux. Since Linux uses different path syntax, you should enter the correct path string for Linux here, for example */usr/local/KernelDriver*

- Now choose as many options as you like and click the **Generate Code** button at the bottom of the screen. Since DriverWizard runs on Linux, it is better to generate Linux code on Linux itself; transferring files from Windows to Linux sometimes requires CR-LF translation, to work correctly with Unix based compilers and tools.

6. Compile and run the generated code.

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

- The kernel mode source code that DriverWizard created should be compiled with MS Visual C++ compiler on Windows, and the GNU gcc compiler on Linux.

- For more information on building your drivers, see Chapter KernelDriver development process that explains the KernelDriver Development Process.

DriverWizard Notes

Sharing a Resource
Disabling a Resource
DriverWizard Logger
Automatic Code Generation

Overview

KernelDriver provides an object-oriented view of the NT driver object model. KernelDriver is modeled closely to the NT architecture, while simplifying it by grouping functionalities and performing many common driver tasks.

This chapter will walk you through the KernelDriver classes, and provide a good insight into the NT kernel architecture. Read through this chapter to understand which classes are available, how they interact, and what the KernelDriver object model is. For a full function reference, turn to Appendix WinDriver Function Reference that explains the WinDriver functions later in this manual.

Driver Vs. Device

Where it all begins - The DriverEntry() Function

Every device driver must have one starting point (like the main() function in a C console application), called **DriverEntry()**. When the operating system loads the device driver, this **DriverEntry()** function is called.

The DriverEntry function may perform start-up initializations. It also registers which driver callbacks (Dispatch routines) will be called by the OS for performing different driver activities. The OS can initiate calling any of the registered driver routines, whenever the corresponding action takes place.

Loading the driver

Communicating with Drivers - IRPs and IOCTLs

Applications can initiate communications by opening a handle to the driver (using the **CreateFile()** call with the name of the device as the file name), and then reading and writing from the device, or sending a request, using the**ReadFile()**, **WriteFile()** and **DeviceIoControl()** calls. In the **DeviceIoControl()** call, the application specifies which device the call is made to (by providing the device's handle), and an IOCTL code that describes which function this device should perform. The IOCTL code is a number which the driver and the requester agree upon for a common task.

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

Classes in brief

The following is a list of the classes that KernelDriver provides, with a short description of each of the classes. After this list you will find a specific section for each family of classes with an in-depth description of each of them.

Basic Classes

Classes for Accessing Device Memory

Classes for Handling Interrupts

Classes for Registry and Resource Reporting

Classes for Synchronization

Utility Classes

Classes for Layered Drivers

Classes for NDIS Drivers

KdDriver, KdDevice and KdIrp

The two basic objects in the device driver architecture are the Driver object (**KdDriver** explained in Section Class KdDriver - The Driver Class), and the Device object (**KdDevice** explained in Section Class KdDevice - The Device Class). A device is an entity which describes a logical I/O device. The Driver object controls one or more devices which are logically grouped together. The driver points to a linked list of devices which it controls. Any interaction with a device is always done through the driver object -- the OS will forward all device calls to the associated driver. The driver will then (by default) forward these calls to the device.

The communication with drivers is done through an I/O Request Packet (IRP) and this is also discussed in Chapter Class KdIrp - The I/O Request Packet that explains the **KdIrp** class.

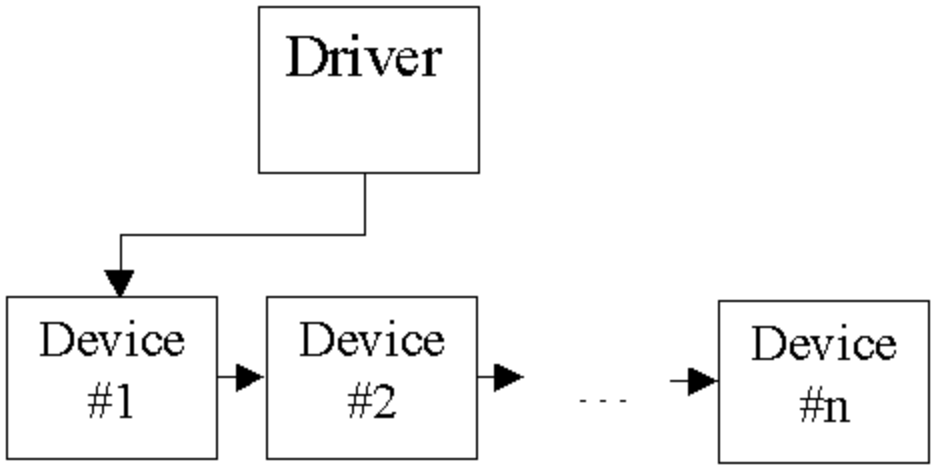

Figure 6.1: Elements of a Basic Driver

Class KdDriver - The Driver Class

Class KdDevice - The Device Class

Class KdIrp - The I/O Request Packet

Calling the driver from an application

Communication between devices

Quick Sample ... How to create a simple driver

Plug-n-Play

Plug-n-Play devices (such as PCI or ISA PnP devices) get their physical bus address only after the initial configuration cycle is complete (as opposed to non plug-n-play devices in which addresses are fixed). Therefore, it is necessary to query the PnP device to know its address range before reading or writing to it.

As an example, consider a PnP card that needs a 16KB block of memory. This card will request memory from the PnP bus controller in the initial configuration cycle. The PnP bus controller will assign this card a physical bus address. This is the address you will provide to the **KdBusAddress** class (which does memory address translations).

I/O space Vs. Memory space

The CPU has an additional address space called an I/O space. To read or write from the I/O space, your driver will have to use I/O specific Read/Write functions. When using the KernelDriver classes, there is no need to use different functions for the I/O space. Your card can declare its memory to be 'I/O space' or 'Memory space'. The I/O space is now a legacy.

Class KdBusAddress

Description

Class KdMapKernel

Description

Class KdMapProcess

Description

Quick Sample ... How to Map and Report Bus Addresses

Sample #1: Map local memory
Sample #2: Map I/O space.

Interrupt Handling

To handle a specific interrupt, a device must register its ISR (Interrupt Service Routine) with the operating system. When a device interrupts, the ISR which is registered on that device is called. The ISR runs at raised Interrupt Request Level (IRQL) and blocks out any other interrupts and system events which are of lower priority. Therefore, the time spent in the ISR must be minimal, and should usually include only the interrupt acknowledgement to the hardware. The rest of the interrupt code should reside in a deferred procedure call (DPC), which runs as soon as the processor is at DISPATCH_LEVEL. The ISR should be defined as a member function of your device's class. The DPC is handled by the**KdDpc** class.

Typically, an ISR will identify the interrupt (i.e. if the interrupt is shared with another driver, the ISR will identify whether this interrupt should be handled by this ISR), acknowledge the interrupt (usually by writing to or reading from a specified register), and schedule a DPC if more processing is needed. If the ISR requires a DPC, it will queue it for execution at a later stage. An ISR might also need to handle time critical operations.

NOTES:

- Several device drivers may handle the same interrupt, as long as they all define the interrupt as sharable.

- Most of the operations cannot be performed at ISR due to the high IRQL, and therefore should be performed at the DPC level.

- The DPC will only occur when one of the processors is available below the DISPATCH_LEVEL. Therefore, the DPC will run only after the ISR is completed (on a single CPU machine).

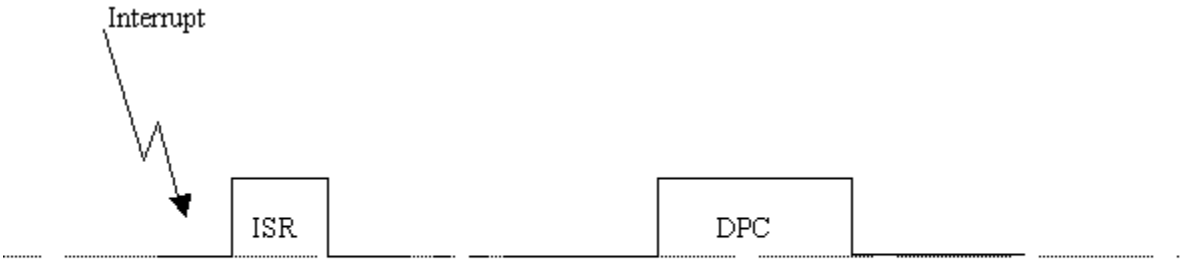

Figure 8.1: Interrupt handling

Implementing an Interrupt

1. Define the interrupt you want to handle by using the **KdIrq** class.

2. The ISR should be defined as a member function of your device's class. The KdIrq object contains a pointer to the ISR.

3. Define the DPC by using the **KdDpc** class.

4. When an interrupt occurs, the ISR is called. If necessary, the DPC is called afterwards.

Class KdIrq

Description

Class KdDpc

Description

Quick Sample ... How to Handle Interrupts

```
#include "..\..\include\ kd.h"

#include "int.h"

class KdStatDriver : public KdDriver
{

public:
    KdStatDriver( NTSTATUS &Status, PDRIVER_OBJECT pDriverObject,
        PUNICODE_STRING puniRegistryPath);

};
class KdStatDevice : public KdDevice
{
public:

    KdStatDevice();
    ~KdStatDevice();

virtual NTSTATUS DispatchDeviceControl (KdIrp &Irp);
virtual void StartIo (KdIrp &Irp);

// This is the DPC
VOID DpcRoutine (PKdDpc Dpc,
        PVOID SystemArgument1,
        PVOID SystemArgument2);

// This is the ISR
BOOLEAN InterruptServiceRoutine(KdIrq *Irq, PVOID Context);

BOOLEAN DoIo();

KdIrq            m_KdInterrupt;
KdBusAddress    *m_pBusAddress;
KdDpc            m_KdDpc;
ULONG            m_InterruptCount;
```

```
};
// Driver entry point
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{

NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
new KdStatDriver(Status, DriverObject, RegistryPath);

if (!NT_SUCCESS(Status) && KdDriver::Driver())
    KdDriver::Driver()->Unload();
return Status;

}

KdStatDriver::KdStatDriver(NTSTATUS &Status, PDRIVER_OBJECT
                           pDriverObject PUNICODE_STRING
                           puniRegistryPath) :
         KdDriver(Status, pDriverObject, puniRegistryPath)
{
           ...
    KdStatDevice{*} pDevice = new KdStatDevice;

    // Initialize the device
    Status = pDevice->Init(FILE_DEVICE_UNKNOWN, L"Int");

           ...

   // This call will map our IRQ to an interrupt.
   pDevice->m_KdInterrupt.Init(
       (KDIRQ_CALLBACK) pDevice->InterruptServiceRoutine,
       pDevice,
       ISA,          // The bus of the device
       0,            // The number of the bus
       STAT_IRQ,     // The interrupt number we are waiting on
       STAT_IRQ,     // The interrupt vector
       Latched       // Type of interrupt (Latched or Level Sensitive)
```

```
       );

      ...

// Register the IRQ with the device
    Status = pDevice->m_KdInterrupt.Connect();
    ...
}

// This will initialize the device's DPC to be `DpcRoutine()'
KdStatDevice::KdStatDevice() :
m_KdDpc ((KDDPC_CALLBACK)DpcRoutine, this)
{
}

KdStatDevice::~KdStatDevice()
{
    ...

}

// The IOCTL dispatch function

NTSTATUS KdStatDevice::DispatchDeviceControl(IN KdIrp &Irp)
{

    NTSTATUS status;
    switch (Irp.IoctlCode())
    {
       case IOCTL_STAT_GET_INTERRUPT_COUNT:
            ...
            *(PULONG)Irp.SystemBuffer() = m_InterruptCount;
            m_InterruptCount = 0;
            ...
    }
    return STATUS_INVALID_PARAMETER;
}

void KdStatDevice::StartIo(KdIrp &Irp)
{
```

```
    // This synchronizes execution between the ISR and the `DoIo()' routine
    // Only one of these routines can execute at a time.

    m_KdInterrupt.SynchronizeExecution((KDIRQ_SYNC_CALLBACK)DoIo);
       ...
}

{BOOLEAN KdStatDevice::DoIo()
{
    ...
}

// This is the ISR. It dismisses the interrupt (by writing to
// the device), and schedules a DPC
BOOLEAN KdStatDevice::InterruptServiceRoutine(KdIrq *Irq,
                                              PVOID Context)
{
    // device specific I/O to dismiss the interrupt
    m_pBusAddress->WriteByte(CONTROL_PORT, ARM+TF_T3);

    // Schedule a DPC by queuing it
    m_KdDpc.InsertQueue();

    return TRUE;
}

// This is the DPC that gets queued by the ISR to
// finish any interrupt relate processing
VOID KdStatDevice::DpcRoutine(
    IN PKdDpc Dpc,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2)
{
    KdIrp Irp(m_pDeviceObject->CurrentIrp);

    m_InterruptCount++;

    if ((PIRP)Irp)
    {
        // need to fill in this field to get the I/O manager to copy the data
```

```
        // back to user address space
        Irp.Information() = sizeof(IOCTL_INFORMATION);
        Irp.Status() = STATUS_SUCCESS;
        StartNextIrp();
        Irp.Complete();
    }

    return;
}
```

Windows NT Registry

The Windows NT registry holds system information, which may include information about the driver you write. You may use the registry to store information about your driver that you may want to reuse the next time your driver starts up.

The device driver's Memory, I/O, Interrupts and DMA channel are called 'Resources'. It is a good practice to report the driver's resources to the NT registry, so that:

1. Your driver ensures that the resources are valid.

2. Determine that the resources that are exclusively requested by the driver are not claimed by another driver.

3. Enable other applications (including user-mode applications) to view the resources that are used by your driver.

Class KdRegistry

Description
Sample

Class KdResources

Description

Quick Sample ... How to Read and Register Resources

The following sample reads the I/O range base address (**PortBase**) and its size (**PortCount**) from the registry (where this configuration data for the driver was set), and reports this resource to the OS through the KdResources class.

```
// The Driver
class KdLocalDriver : public KdDriver
{

public:
    KdLocalDriver(NTSTATUS &Status, PDRIVER_OBJECT pDriverObject,
                  PUNICODE_STRING puniRegistryPath);



    KdResources m_Resources;       // The Driver's resources object
};

KdLocalDriver::KdLocalDriver(NTSTATUS &Status, PDRIVER_OBJECT
                             pDriverObject,PUNICODE_STRING
                             puniRegistryPath):
       KdDriver(Status, pDriverObject, puniRegistryPath)
};

          ...

       ULONG PortBase;         // Port location, in NT's address form.
       ULONG PortCount;        // Count of contiguous I/O ports
       KdPhysAddr PortAddress;

       // Try to retrieve base I/O port and range from the
       // Parameters key of our  entry in the Registry.
       // If there isn't anything specified then use the values
       //compiled into  this driver.

       KdRegistry Registry;

       if  (!NT_SUCCESS(Registry.Init()))
       {
           PortBase = BASE_PORT;
           PortCount = NUMBER_PORTS;
```

```
        }
        else
        {

            Registry.QueryDWORD(L"IoPortAddress", &PortBase, BASE_PORT);
            Registry.QueryDWORD(L"IoPortCount", &PortCount, NUMBER_PORTS);
}

            PortAddress = PortBase;

           // Register resource usage (ports)
           //
           // This ensures that there isn't a conflict between
           // this driver and  a previously loaded one
           // or a future loaded one.
           m_Resources.Init(Isa);

           m_Resources.AddIo(PortAddress, PortCount);

           // Report it's resources. We do this now because we are just
           // about to touch the resources for the first time.

           Status = m_Resources.Report();

           if (NT_SUCCESS(Status) && m_Resources.GetIsConflictDetected())
              Status = STATUS_DEVICE_CONFIGURATION_ERROR;

             ...

}
```

Class KdSyncObject

Description

Class KdEvent

The basic synchronization object

Description

Class KdSpinLock

Description
Sample

Class KdMutex

Description

Class KdSemaphore

Description

Class KdTimer

Description

Class KdTimedCallback

Description

Class KdThread

Description

Class KdString

Description
Sample

Class KdList

Description
Sample

Class KdIrpList

Description
Sample

Class KdFifo

Description

Class KdFile

Description
Sample

Class KdPhysAddr

Description

Class KdMem

Description

DebugDump utility

Description
Sample

Memory Compare Utilities

Description
Sample

Memory Allocation Utilities

Description
Sample

Class KdFilterDevice

Description

Class KdLowerDevice

Description

Overview

The NDIS Miniport framework is used to create network device 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 which 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.

KernelDriver provides NDIS Miniport classes, which contain code that is common to many NDIS Miniport, drivers, such as NDIS registry handling, initialization code, etc. The NDIS classes also provide a convenient framework, which enable the user to jump-start the NDIS Miniport driver development.

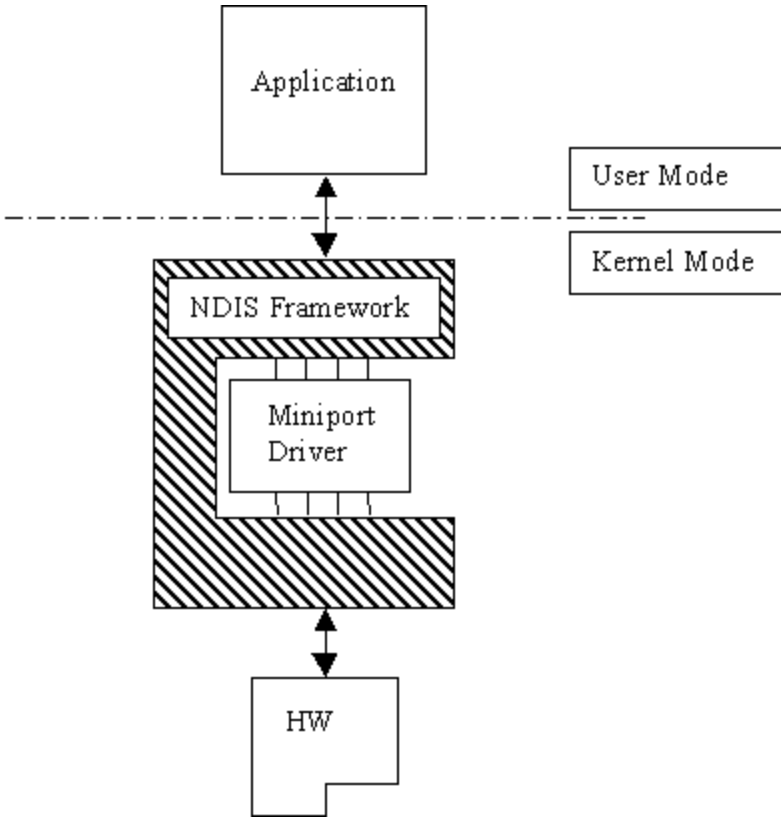

Figure 13.1: NDIS Architecture

There are two major classes involved in order to implement a complete NDIS Miniport driver:

- **An Adapter class** - which represents a network interface card (Network Interface Cards - NICs).
- **A Miniport class** - which is the container object of several adapter classes.

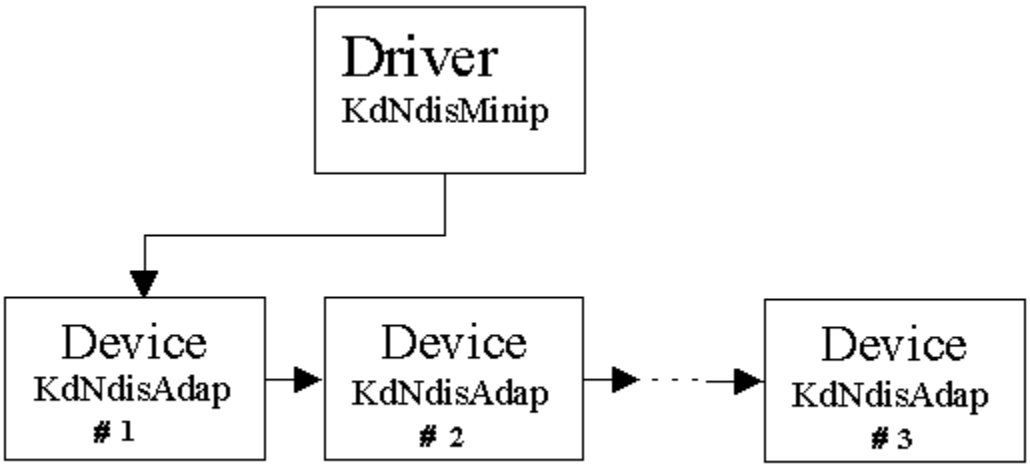

Figure 13.2: NDIS Miniport Driver

The relation between these two classes can be compared to the relation between KdDriver and KdDevice classes in the way that the Miniport class may own several Adapter classes.

A NDIS miniport driver can support one or several NICs. The list of the adapter classes which the driver controls is found under the Linkage key of that driver: (*HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Service\<drivername>\Linkage*). A miniport driver is implemented as a derived class from KdNdisMiniport. For each NIC supported by the miniport, it creates an adapter class that is derived from an adapter base class called KdNdisAdapter. When the miniport driver is loaded, it registers itself with NDIS and NDIS in turn will check the registry and have the miniport create an adapter and initialize it for each NIC the driver supports.

The miniport operates through a set of callback functions which are registered and then called by NDIS. The functionality of these callbacks are specific for each NIC, and are thus implemented in the adapter classes.

The KdNdisMiniport class

Description
Initialization
Behind the scenes

The KdNdisAdapter class

Description
Initialization

The KdNdisConfiguration class

During the initialization process the adapter must query the registry for NDIS specific configuration information regarding the NIC. This class enables an easier way to retrieve that information, while conforming to the NDIS specifications of accessing the registry.

Quick Sample ... NE2000 NDIS Miniport Driver

Overview

During your driver development process, you will use Microsoft's Developer Studio to write and compile your drivers. However, you cannot use the MSDEV's debugger to debug your kernel mode driver.

Debugging device drivers is a challenging feat. It demands that you properly set up a debugging environment, and learn to use new debugging tools. This chapter will show you how to set up your debugging environment, and how to use **KernelTracer**and **WinDbg** to debug your driver.

Using KernelTracer

KernelTracer 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***). Using this tool you can monitor how each WinDriver command (WD_XXX) sent to the kernel is executed. This pertains to all WinDriver functions (WD_XXX) that you use in your code. KernelTracer can also intercept the Debug data which you send to it from your driver code via the **DebugDump()**function. For other debugging facilities, see Section Using WinDbg that explains how to debug with WinDbg .

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

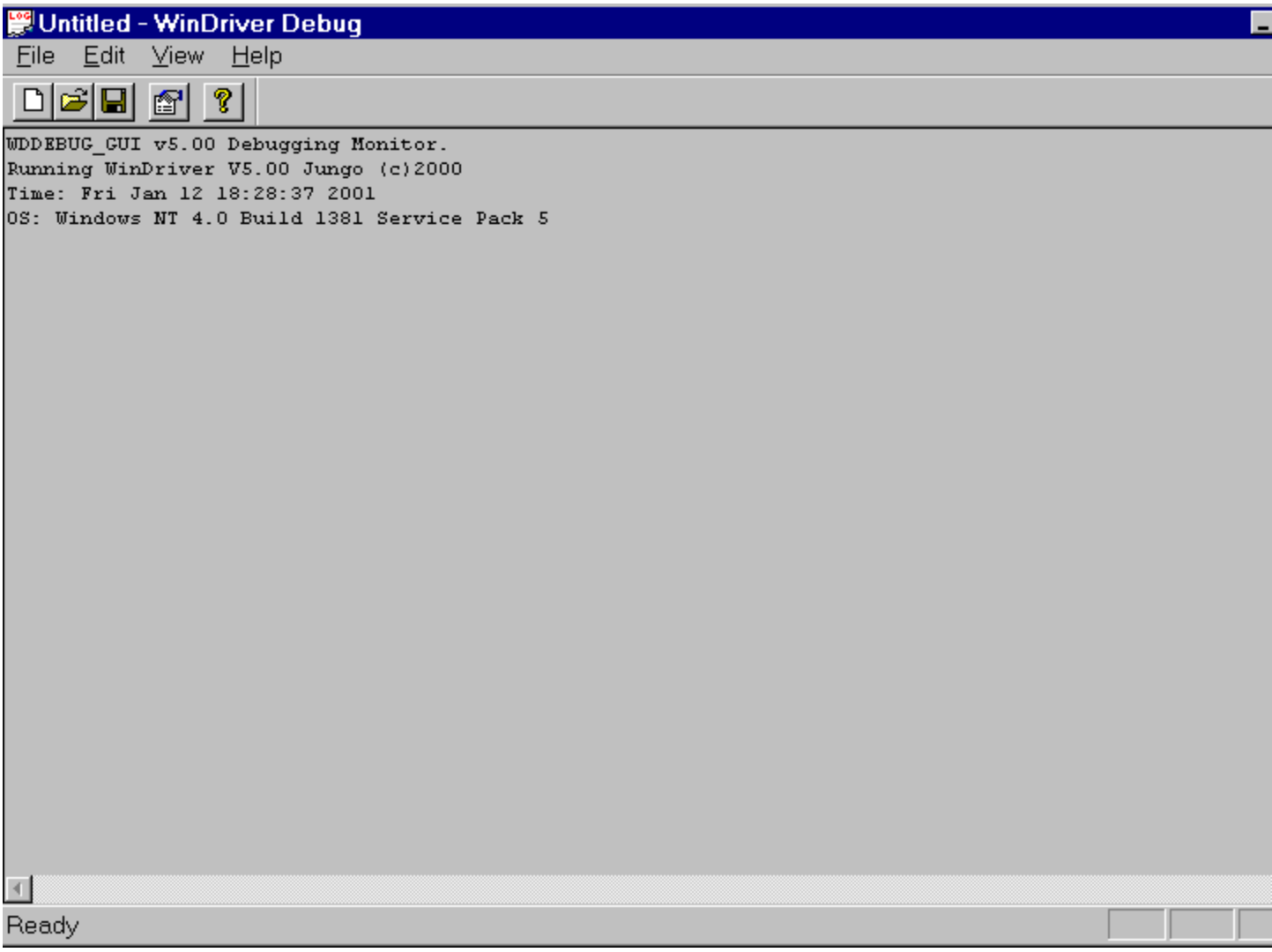

Figure 14.1: KernelTracer - Graphical Mode

KernelTracer - Graphical mode

- Start KernelTracer from ***Start*** **|*Programs*** **|*KernelDriver*** **|*Monitor Debug Messages***.

- Activate and set the trace level you are interested in from the ***View*** **|** ***Debug Options*** menu or using the Change Status button.

Figure 14.2: Set trace Level

1. **Status** - Set trace on or off.

2. **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.

3. **Ker_drv option** - This is for KernelDriver users, monitoring communication between their custom Kernel mode drivers (developed using KernelDriver) and the WinDriver kernel.

4. **Level** - Choose the level of messages you are interested to see for the resources defined. **Error** is the lowest level of tracing, 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.

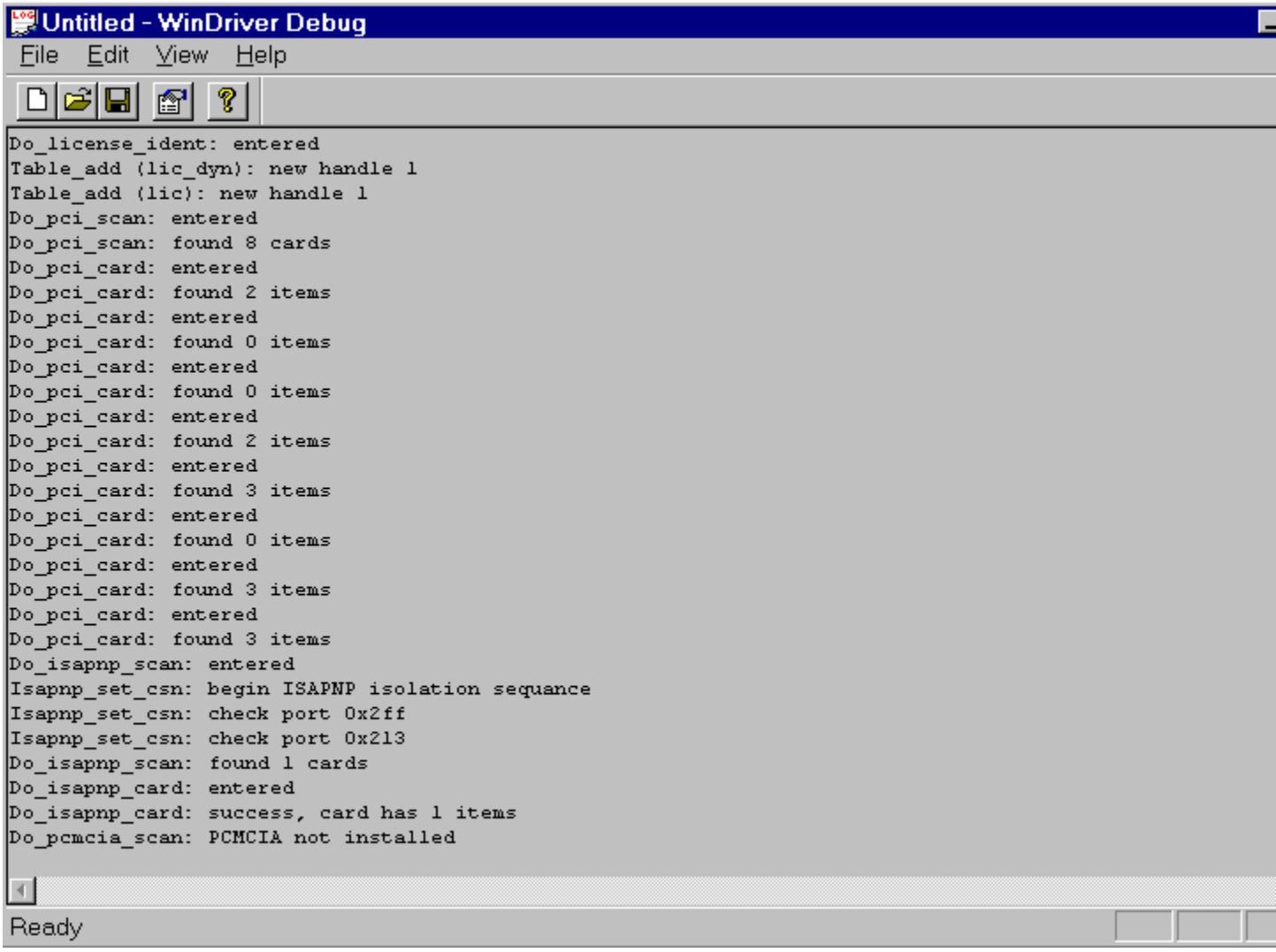

Figure 14.3: Debug Monitor

KernelTracer - Console Mode

This tool is available in all operating systems supported including Linux. To use it, just type ***wddebug*** from the ***KernelDriver\util***directory with the appropriate switches. For a list of switches available with KernelTracer in console mode just type ***wddebug***and a help screen is displayed, describing all the different options for this command.

To see activity logged with KernelTracer simply type ***wddebug dump***.

Using WinDbg

Overview

Establishing a WinDbg Debugging Session

Connect the Host and Target Machines
On the Target Machine
On the Host Machine

WinDbg Command-Line Options

The following options are available from the command line:

windbg [-a] [-g] [-h] [-i] [-k [platform, port, speed]] [-l[text]] [-m] [-p id [-e event]] [-s[pipe]][-v] [-w name] [-y path] [-z crashfile] [filename[.ext][arguments]]

Options

Debugging a Crash Dump

To configure the target to generate a crash dump, select ***Start Menu*** **|** ***Settings*** **|** ***Control Panel*** **|** ***System***. Click on the ***Startup/Shutdown*** tab, and then select the '**Write Debugging Information To:**' option. The file specified in this option is the crash dump file. Its default name is **memory.dmp**.

Specify the name of the crash dump file in the **Kernel Debugger Options**dialog box or use the **-z** command-line option to specify the name of the crash dump file.

Two utilities, DUMPCHK and DUMPREF shipped with the DDK are useful in debugging crash dumps:

DUMPCHK [options] CrashDumpFile

DUMPCHK is used to check the validity of a crash dump file.

DUMPCHK Options

DUMPREF Options

Comments

Inheritance Chart

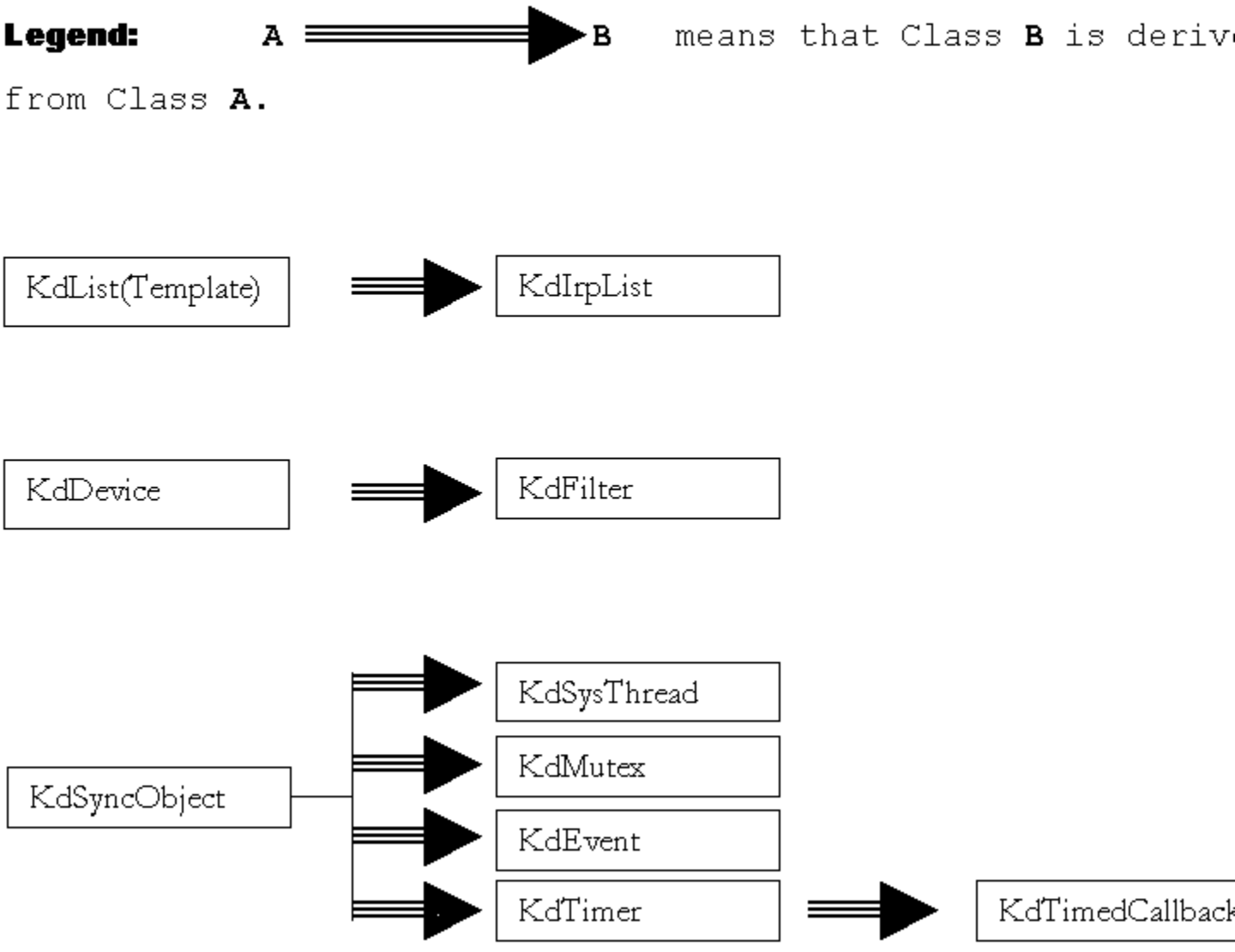

Figure 15.1: Inheritance Chart

Dependency Chart

Legend: **A** ⟹ **B** means that Class **B** contains Class **A** as a member.

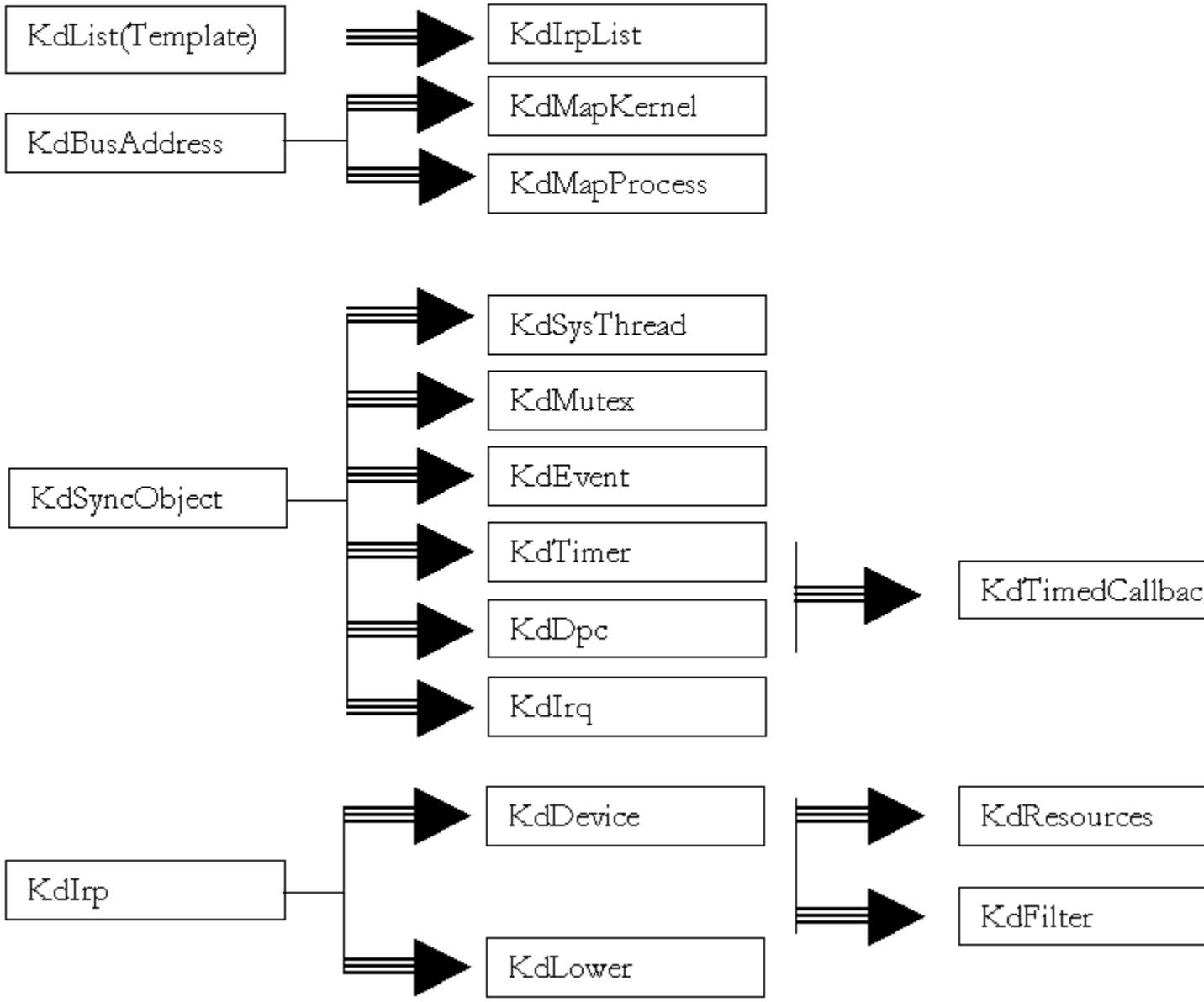

Figure 15.2: Dependency Chart

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_SENSITIV 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 card
DWORD	dwSlot	PCI physical slot number of card
DWORD	dwFunction	PCI function on card

WD_PCI_ID

Defines the identity of a PCI card.

MEMBERS:

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

WD_PCI_SCAN_CARDS

Receives information on cards detected on the PCI bus.

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

MEMBERS:

TYPE	NAME	DESCRIPTION
WD_PCI_ID	searchId	If searchId.dwVendorId==0, then 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/written
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 th
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 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 c 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_HALT 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 actu 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 interfac 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

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.

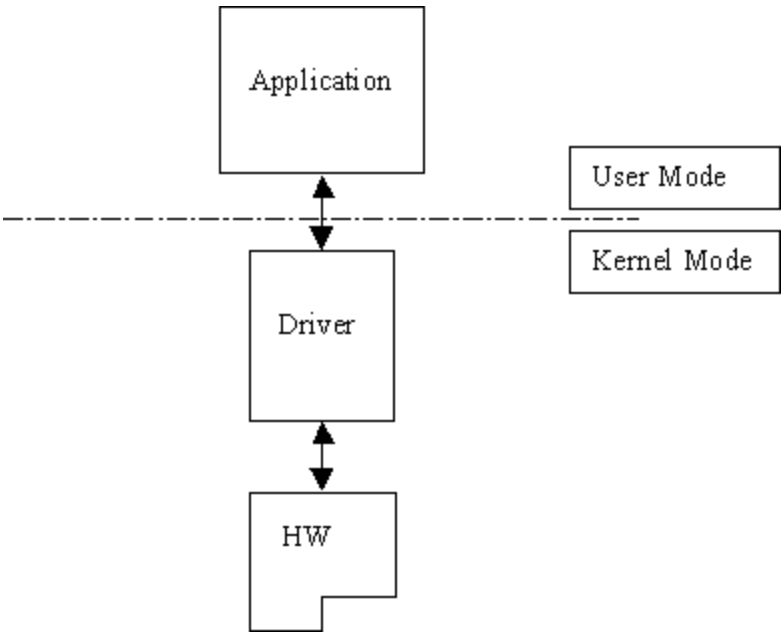

Figure 1.1: Monolithic Drivers

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.

Windows

- NT 4.0 DDK and Windows 2000 (WDM) DDK - Device driver kit. You need this if you are creating Windows NT or Windows 2000 drivers. The DDKs are available in the MSDN subscription from Microsoft. You can also download free of charge from
  `http://www.microsoft.com/hwdev/ddk`

- Windows 95 DDK - Device driver kit. You need this if you are creating Windows VxDs.

NOTE: The SYS files created using DriverWizard, or from the KernelDriver samples will not work on Windows 98/ME. They are meant to be used only on NT and 2000. For 95/98/ME, please use the KernelDriver VxDs.

Linux

- Linux kernel sources and header files (Version 2.0 and Version 2.2 series only)
- GNU GCC compiler (Versions 2.6.2 and above are recommended)
- Linux kernel modutils package

Installing KernelDriver On Windows 95/98/ME/NT/2000

- Install the NT or WDM DDK if you wish to create drivers for Windows NT/2000.

- The installation of the DDK should automatically point your **BASEDIR** environment variable to the directory in which the DDK is installed. Make sure that this has been set by typing in from the command prompt: ***set BASEDIR***. If you are not using a COMMAND shell to build your drivers, then use the Control Panel (under Windows NT/2000) to add this variable to the environment, or AUTOEXEC.BAT under Windows 95/98/ME.

- Install the Windows 95 DDK if you wish to create VxD drivers for Windows 95/98/ME

- When compiling Windows 95/98/ME VxD drivers, the **BASEDIR** environment variable should be set to point to the directory in which the DDK is installed. Make sure that this has been set by typing in from the command prompt: ***set BASEDIR***. If you are not using a COMMAND shell to build your drivers, then use the Control Panel (under Windows NT/2000) to add this variable to the environment, or AUTOEXEC.BAT under Windows 95/98/ME.

- Install KernelDriver by double clicking the KernelDriver icon on your CD (KD500.EXE)

Installing KernelDriver on Linux

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

- Insert the kernelDriver CD into the CD drive of your Linux Machine

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

- Extract the file kdxxxln.tgz (where xxx is the version number) (**/usr/local>tar xvzf /mnt/cdrom/LINUX/kdxxxln.tgz**)

- If you downloaded the installation file from the WEB site, please extract the file kdxxxln.tgz from wherever you saved it. (**/usr/local>tar xvzf /usr/local/kdxxxln.tgz**)

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

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

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

- 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.

The following steps are for Registered Users only

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

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

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

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

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

6. To install the registered version run the command **make install IS_REGISTERED=1**

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

Restricting hardware access via /dev/windrvr

Testing the installation under Windows NT/2000

- Open MSDEV (version 5 and above).

- Open the ***\kerneldriver\samples\simple\simple.dsw*** project file - This will open up two projects:

1. simple: which is the sample kernel mode driver.

2. gethndl: which is the sample application that uses the kernel mode driver.

- Compile both projects (user mode and kernel mode).

- Install the driver created by:

1. Copying the created **simple.sys** from ***\kerneldriver\samples\simple\sys\free*** to ***\winnt\system32\drivers***.

2. From the command line type ***c:\kerneldriver\util\wdreg.exe name simple-startup manual create*** (this registers your driver in the NT registry).

3. From the command line, type ***net start simple*** (this runs your driver).

- Run your sample application by running ***\kerneldriver\samples\simple\exe\win32\gethndl.exe***.

- Make sure that the application notifies you that it succeeded in getting the handle to the driver by noting the messages displayed on the screen.

Testing the Installation under Windows 95/98/ME

- Use DriverWizard to generate a simple kernel mode hardware access VxD for your parallel port (the parallel port is displayed as an ISA card by DriverWizard)
    - Compile the sample to create a VxD. To do this:
        - Open a COMMAND prompt window
    - Change directory to the path where you generated source code for the VxD project (say under C:\myvxd)
    - CD C:\myvxd\vxd\
    - Build the vxd (NMAKE /F myvxd.mak)
    - If you don't enjoy using nmake on the command line, Use Visual Studio to open the file myvxd_driver.dsp file under the directory C:\myvxd\vxd\msdev5 and build the myvxd.vxd driver using the Build Menu.

- Use Visual Studio to open the file myvxd_diag.dsp under the directory C:\myvxd\usr_mode\ msdev5 directory and build the sample usermode application myvxd_diag.exe using the Build Menu.

- Load the VxD driver under Windows. To do this:
    - Copy the file myvxd.vxd to the directory C:\WINDOWS\SYSTEM\VMM32
        - Install the VxD using the command *C:\KernelDriver\util\wdreg -vxd -name myvxd install*

- Launch the test program myvxd_diag.exe in a command window. This program allows you to work with the parallel port.

- Observe the output of the program. If the program fails to connect to the VxD driver, it will issue an error message.

Troubleshooting on Windows

- Make sure that the **DDK BASEDIR** environment variable has been set to the directory that your DDK resides in.
    - Evaluation users - Make sure that your 30 day license has not expired.

- Use the Kernel Tracer (GUI Debug Monitor) to see what the problem is. This utility is found under WinDriver\util by the name wddebug_gui.exe.

Testing the installation under Linux

- Use DriverWizard to generate a simple kernel mode hardware access kernel module for your parallel port (the parallel port is displayed as an ISA card by DriverWizard)

- Compile the sample to create a Linux kernel module.

  For an in-depth explanation on Compiling under Linux, please see Section Compiling under Linux that explains how to compile under Linux.

- Insert the module kdprj.o into the kernel.

  For an explanation on how to do this, please see Section Installing under Linux that explains how to install under Linux.

- Launch the test program kdprj_diag. This program allows you to work with the parallel port.

- Observe the output of the program. If the program fails to connect to the kernel module driver, it will issue an error message.

Troubleshooting on Linux

- Make sure you have the kernel modutils package loaded -- this gives you the commands lsmod, rmmod and insmod -- otherwise wdreg will fail when attempting to load your kernel module.

- Make sure your kernel module is correctly loaded by issuing the command /sbin/lsmod. This should show an entry for your kernel module.

- Make sure that windrvr.o is already loaded. Your kernel module will communicate with windrvr.o for hardware access and licensing.

Uninstalling KernelDriver from Windows 95/98/ME/NT/2000

1. Uninstall the WinDriver service (which is shared between WinDriver and KernelDriver) using the command *windriver\util\wdreg remove*. Do not do this if you have WinDriver running on your system and you wish to continue using it.

2. Use the "Add/Remove Programs" applet from the Control Panel, select KernelDriver 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

4. If the KernelDriver installation directory did not get removed by the automated installation, then remove it manually using Windows Explorer.

Uninstalling KernelDriver 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 KernelDriver module is in use by any application or by other modules
   - Make sure that no programs are using KernelDriver
   - if any application or module is using KernelDriver, close all applications and do a /sbin/rmmod to remove any module using KernelDriver
   - Run the command "/sbin/rmmod windrvr"
   - rm -rf /dev/windrvr (Remove the old device node in the /dev directory)
   - Remove the file .kerneldriver.rc in the /etc directory.
     Run the command rm -rf /etc/.kerneldriver.rc to do this.
   - Remove the file .kerneldriver.rc in $HOME.
     Run the command rm -rf $HOME/.kerneldriver.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/KernelDriver"

Using DriverWizard to create your driver

The following is a quick walk-through of developing your driver using DriverWizard. For a full description of DriverWizard, see Chapter The DriverWizard that explains DriverWizard.

- Use DriverWizard to diagnose your hardware.
- Generate the code - This creates two project files:

1. A driver project (which creates a device driver).
2. An application project (which creates the source code of an application that accesses your device).

- The Driver project includes a file with specific APIs for accessing your hardware. You can use these APIs from within any other driver you write.
- Install your compiled driver by copying it to the proper directory and starting it (see Section Installing, Loading and Registering Your Driver that explains how to install your driver).
- Run your application.
- Modify the generated source code - see Chapter KernelDriver Class Model that explains the KernelDriver classes for more information.

Using a Sample to create your driver

Find the sample which is closest to the driver you need to create.

Open the sample's workspace (e.g. MyDriver.dsw) in MSDEV 5.0 and above. It is very convenient to use the sample's project file to quickly load all project information and files into your IDE. On Linux, you need to use the make command to build your project. All the samples have makefile suitable for use on Linux.

Compile the sample code and install your compiled driver. All samples include a companion user mode application to exercise the driver.

List of KernelDriver samples

Writing a Windows NT Model Driver

In this section, we cover a Windows NT Model Driver in detail.

Directory Structure
Writing the Driver

Writing drivers for other operating systems

As we discussed before KernelDriver has special support for Windows NT model drivers. However, it can also be used to write almost any kind of driver for other Windows operating systems and Linux. The code that KernelDriver generates can be equally compiled as a Windows VxD, a Windows NT SYS file, or a Linux kernel module. Since kernel mode code is generally not portable, once you modify the generated code and insert into it, operating system specific kernel mode code, the source code is not portable anymore. If portability is your top priority, consider using WinDriver instead.

However, some measure of portability is also possible with KernelDriver. The full WinDriver API is available and can be accessed from kernel mode from drivers written by KernelDriver. The following subsection illustrates this. The steps for all the OSes are similar on the source code level. The makefile or the build project files may differ, of course, since this is platform and compiler dependent.

Example on WinDriver API access from kernel mode

Writing a Windows NT Model driver without using WinDriver

In section Writing drivers for other operating systems, we walked through the source code of a Linux kernel mode driver which calls the WinDriver WD_XXX API from the kernel mode. If you study the equivalent 9054 samples for Windows NT (under the **9054\sys\cpp** directory), you see the sort of code we walked through in Section Writing a Windows NT Model Driver in the file *p9054_driver.cpp*, and calls to the WinDriver API in the file **plx\9054\lib\9054\p9054_lib.c**.

Since KernelDriver has a C++ class library that wraps the Windows NT DDK functionality for Windows NT and Windows 2000, sometimes you may wish to write a NT kernel mode driver that uses this class library without any dependence on the Windriver API. Specifically, you may not want to redistribute the file *windrvr.sys* with your driver at all. If you intend to develop drivers in this manner, we strongly recommend that you acquire the source code of the KernelDriver class library (please contact sales@jungo.com for more information).

Recall the simple code fragment from Section Writing the Driver. We created a driver object by instantiating the **KdDriver** class. The constructor for this class is declared in **kddriver.h** as follows:

```
class KdDriver
{
public:
    KdDriver(NTSTATUS &Status,
        PDRIVER_OBJECT pDriverObject,
        PUNICODE_STRING puniRegistryPath,
        BOOLEAN fImplementUnload = TRUE,
        BOOLEAN fLoadWinDriver = FALSE);
```

If you do not wish to connect to WinDriver automatically when your kernel mode driver is instantiated, you should call the **KdDriver** constructor in your**DriverEntry** procedure, with the *fLoadWinDriver* argument set to FALSE:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath)
{
    new KdDriver(Status, DriverObject, RegistryPath, TRUE, FALSE);
    ....
}
```

As you might have already noticed, the default value for the *fLoadWinDriver*argument is FALSE, so you may just skip specifying it, if you do not want **KdDriver** to attempt to connect to WinDriver. To have the **KdDriver** class automatically connects to WinDriver at startup, you have to pass the parameter TRUE for the *fLoadWinDriver* argument.

Compile the Driver

Compiling under Windows
Compiling under Linux

Installing, Loading and Registering Your Driver

Installing under Windows
Installing under Linux

Running Your Driver

Running Your Driver under Windows
Running Your Driver under Linux

Usage under Windows

WDREG [-vxd] [-name <vxd_name>] [-file <file_name>] [[CREATE] [START] [STOP] [DELETE] [INSTALL] [REMOVE]]

WDREG.EXE has 4 basic operations:

- **CREATE** - Instructs Windows to load your driver the next time it boots, by adding your driver to registry.
- **START** - Dynamically loads your driver into memory for use. On Windows NT, you must first 'CREATE' your driver before 'START'ing it.
- **STOP** - Dynamically unloads your driver from memory.
- **DELETE** - Removes your driver from the registry , so that it does not load on next boot.

For example,

To Load your driver in the next boot, use:

WDREG -name MyDriver CREATE

To reload your driver use:

WDREG -name MyDriver 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 registry so that it does not load on next boot. (same as using ***WDREG STOP DELETE***).

For example,

To create and start your driver you can just use:

WDREG -name MyDriver INSTALL

WDREG enables 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 (file) name] install*

Example: To install the sample.sys driver under the name MySample in the registry, use:

WDREG -name MySample -file sample install

The -VXD option tells WDREG that you are loading a VXD rather than a SYS file. On Windows 98, the default is to load the SYS file. To force WDREG to load a VXD, please supply the -VXD option.

Example:

WDREG -VXD -name MySample -file sample install

Usage under Linux

Under Linux, wdreg is a shell script that first tries to unload an older instance of your module. It then removes any stale device nodes that referred to the old module. This step in necessary because under Linux, the module major numbers are dynamically assigned and may change after each insertion. The script then loads the new module and creates a corresponding device node under /dev.

Advanced users may look at the shell script under WinDriver/util to see how it works.

USAGE: WDREG <module_name>

Using WDREG from within your application

Its possible for an application to launch a driver dynamically, and then proceed to access the driver.

Under Windows
Under Linux

Introduction

KernelDriver USB is a kernel mode device driver development tool for writing kernel mode USB drivers. The code generated by Kernel Driver USB can currently be compiled and run on Windows 2000 only. Please contact *marketing@jungo.com* or check our website: *http://www.jungo.com/kdusb.html* for the latest information on this product.

It is recommended to use KernelDriver USB when developing 'Standard' USB drivers, such as USB NDIS drivers, USB to Serial Port drivers, or for writing USB drivers that need to communicate with other kernel drivers.

Advantages

KernelDriver USB uses the same API as WinDriver. This implies that you can access your USB hardware using the same source code from both the User Mode and the Kernel Mode (SYS). Both products use the same hardware and code generation wizard and thus offer a complete driver development environment.

Building a Kernel Mode USB Driver

The following steps illustrate the process of building a Kernel Mode USB driver using Kernel Driver USB.

1. Generate the USB driver code using DriverWizard. For details on using DriverWizard with USB, please see Chapter The DriverWizard that explains DriverWizard.

2. Open the generated dsw file using MSDEV 5.0 or 6.0.

3. Modify the generated code to suit your needs.

4. Build the SYS driver and the user mode application program using MSDEV.

5. Generate an INF file using DriverWizard.

Installing a Kernel Mode USB Driver

To install the Kernel Mode USB driver that you built as described in the previous section, follow the steps outlined below:

1. Copy the SYS file built in the previous section to *WINNT\SYSTEM32\DRIVERS*.

2. Use the Device Manager Hardware Wizard to install a PnP driver for your device, using the INF file generated above.

   (This is required for WDM compliance)

3. Install the SYS file by using the command *wdreg -name <SYS FILE> install*, where <SYS FILE> refers to the name of the SYS FILE that was built in the previous section.

4. You can now start the user mode application to access your USB device.

NOTE: The <SYS FILE> uses the WinDriver Kernel (WINDRVR.SYS) to access the USB hardware. Therefore, the WinDriver service should be started prior to accessing the USB device. You can manually install WINDRVR.SYS by doing the following:

- Copy it to *WINNT\SYSTEM32\DRIVERS*

- Run the command *WDREG INSTALL*

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.

Choose **Generate Code** from the **Build** menu. Choose **'KernelDriver'** in the next menu, to generate kernel mode code.

DriverWizard generates the source code for your driver, and for a sample application that accesses your driver, and place it under the same directory of the project file.

NT The driver files are placed under the sys directory, under subdirectories called *C* or *C*++. The application files are placed under the usr_mode directory.

95/98/ME The driver files are placed under the vxd directory. The application files are placed under the usr_mode directory.

Linux The driver files are placed under the linux directory. The application files are placed under the usr_mode directory.

Test The user mode application files are placed under the usr_mode directory. There are subdirectories under usr_mode for each operating system / development environment (Linux, MSDEV5, MSDEV6 ...').

In the driver source code directory you now have a new **'xxxlib.h'** file that 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. Kernel Mode library functions for accessing each element of your card's resources (Memory ranges, I/O ranges, registers and interrupts).

2. Kernel Mode device driver.

3. A 32-bit diagnostics program, in console mode with which you can diagnose your card. This application calls the kernel mode device driver that DriverWizard created.

4. A project workspace or makefile that you can use to build your application.

You may now modify the driver and the sample application so that the functionality fits your needs.

Driver Vs. Device

The two basic objects in the device driver architecture are the Driver object, and the Device object. A device is an entity which describes a logical I/O device. The Driver object controls one or more devices which are logically grouped together.

Loading the driver

When the OS loads the driver, it calls it's **DriverEntry()** routine, and registers it's dispatch functions. A device driver may be loaded automatically (by the OS), or manually (by using the command line ***net start***, or by starting it from the control panel).

A driver may be assigned one of the following loading modes:

Boot - Driver is started at the NT bootstrap process.

System - Driver is started up along with the system drivers.

Automatic - Driver is started along with the system and application start-up.

Manual - Driver is loaded only upon user request.

Disabled - Driver cannot be started.

Basic Classes

Class KdDriver
Class KdDevice
Class KdIrp

Classes for Accessing Device Memory

Class KdBusAddress
Class KdMapProcess

Classes for Handling Interrupts

Class KdDpc
Class KdIrq

Classes for Registry and Resource Reporting

Class KdRegistry
Class KdResources

Classes for Synchronization

Class KdSyncObject
Class KdSpinLock
Class KdEvent
Class KdMutex
Class KdSemaphore
Class KdTimer
Class KdTimedCallback

Utility Classes

Class KdMem
Class KdString
Class KdFile
Class KdList
Class KdIrpList
Class KdFifo
Class KdPhysAddr

Classes for Layered Drivers

Class KdLowerDevice
Class KdFilterDevice

Classes for NDIS Drivers

Class KdNdisMiniport
Class KdNdisAdapter

Class KdDriver - The Driver Class

This section describes the Driver Class

Description
Initialization
Handling IRPs

Class KdDevice - The Device Class

Description
Symbolic links

Class KdIrp - The I/O Request Packet

Description

Calling the driver from an application

When an application calls a driver via **ReadFile()**, **WriteFile()**and **DeviceIOControl()**, the I/O manager encapsulates this call in an IRP. This IRP is passed to the relevant driver which passes it on to the called device.

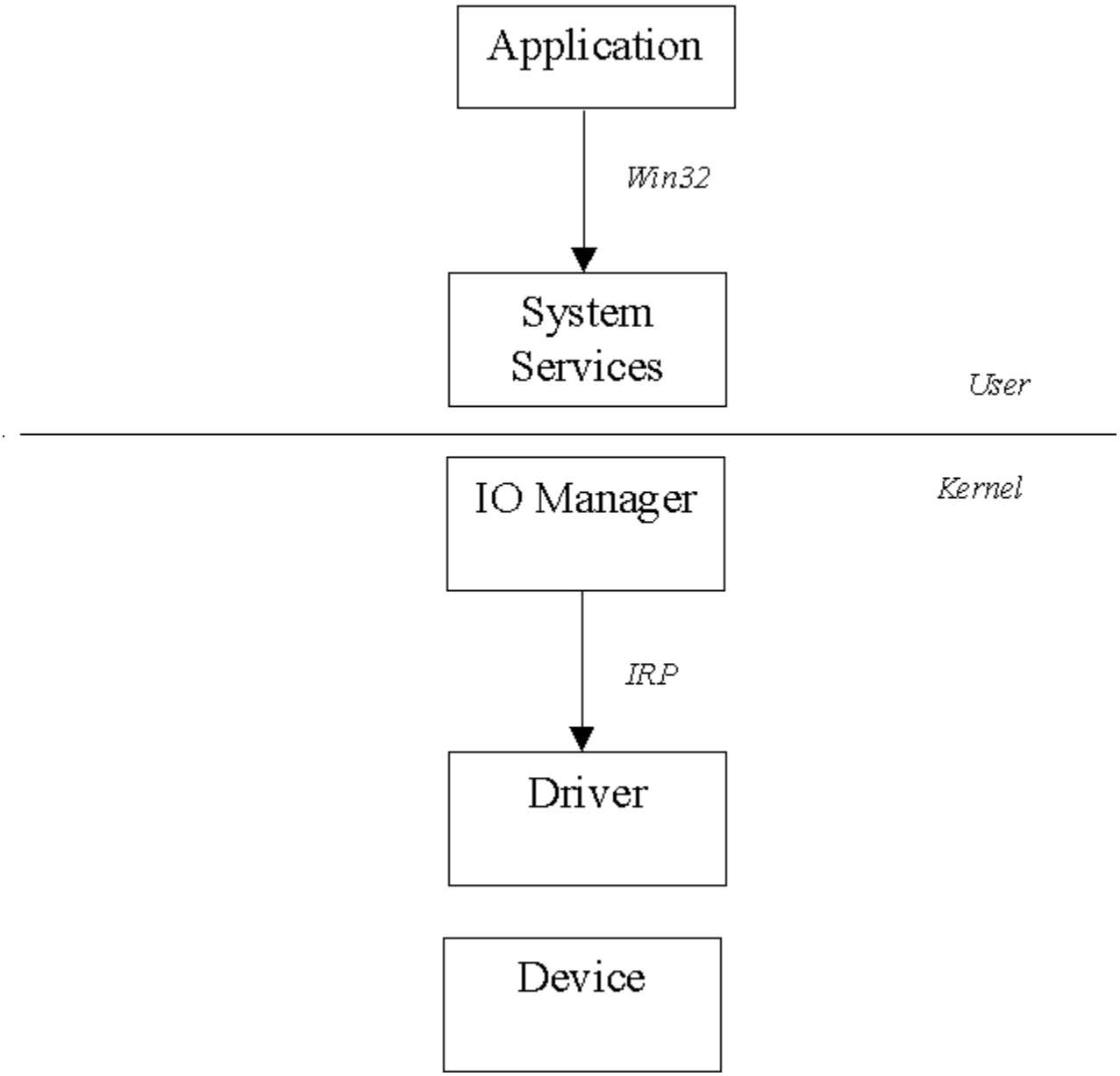

Figure 6.2: Calling the driver from an application

Communication between devices

When a device needs to send a message to another device, it creates a new IRP, and passes it on to the device it needs to communicate with. Note that the devices do not need to belong to the same driver.

Quick Sample ... How to create a simple driver

'Simple' sample - In the following sample, you will see how **KdDriver** explained in Section Class KdDriver - The Driver Class and **KdDevice** explained in Section Class KdDevice - The Device Class are used. The sample uses these objects to create a very simple (but operational) kernel mode device driver:

```
// This driver demonstrates building a simple driver, and
//dynamically loading and unloading it using Kernel Driver

#include "..\..\..\include\kd.h"

// Create a new driver class that inherits KdDriver
class KdSimpleDriver : public KdDriver
{
public:
    KdSimpleDriver(NTSTATUS &Status, PDRIVER_OBJECT pDriverObject,
         PUNICODE_STRING puniRegistryPath);
};

// Create a new driver class that inherits KdDriver
class KdSimpleDevice : public KdDevice
{
};

// This is the main entry point where the driver is created
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
                     PUNICODE_STRING RegistryPath)
{
        NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;

        new KdSimpleDriver(Status, DriverObject, RegistryPath);

        if (!NT_SUCCESS(Status) && KdDriver::Driver())
                KdDriver::Driver()->Unload();
        return Status; }
}

KdSimpleDriver::KdSimpleDriver(NTSTATUS &Status, PDRIVER_OBJECT
                pDriverObject, PUNICODE_STRING puniRegistryPath):
        KdDriver(Status, pDriverObject, puniRegistryPath)
{
```

```
        // Check status returned from the base class constructor
        if (!NT_SUCCESS(Status))
        {
             DebugDump(DBG_ERRORS , ("ERROR KdDriver::KdDriver()
                                      FAILED!\n"));
             return;
        }
        //Write debug information to the KernelDriver debug output
        DebugDump(DBG_DIAG1 , ("Entered the Simple driver!\n"));
        // Create a new device and connect it to this driver
        KdSimpleDevice *pDevice = new KdSimpleDevice;
        Status = pDevice->Init(FILE_DEVICE_UNKNOWN, L"Simple");

        if (NT_SUCCESS(Status))
            DebugDump(DBG_DIAG1 ,("All initialized - !\n"));
        else
            DebugDump(DBG_ERRORS ,("Couldn't create the device\n"));
}
```

Description

Translates the given address on a given bus to a CPU physical address.

As a result of the address space architecture described above, a physical address on a hardware peripheral must be mapped to the CPU's address space (the physical address space) before being used by the driver.

Description

Translates physical addresses to virtual addresses in the kernel mode space.

The CPU instruction set uses virtual addresses to access memory. The CPU translates the virtual address to physical addresses using the page table of the current process. Therefore, entries need to be added to the page table to point to the physical address to be able to access the memory from the driver. KdMapKernel adds the page table entries to the kernel page table.

When mapping an address with KdMapKernel, the type of bus and bus address are given. The address returned is a virtual kernel mode address.

Description

Maps local device memory to the user mode virtual memory of the subject process. (i.e. to the virtual memory of the process that called the driver).

Sample #1: Map local memory

```
//Map local device memory to the subject process's user mode
//virtual memory(i.e. to the virtual memory of the process that
//called the driver).
KdMapProcess MyMapProcess(InterfaceType,        // Bus Type (PCI, ISA,..)
                          BusNumber,            // Bus Number
                          BusAddress,           // Bus Local Address
                          Length);              // Memory Range Length

// Get the virtual address for the user mode process that was
// mapped in the constructor (in the previous function).
ProcAddress = MyMapProcess.GetProcAddress();
```

Sample #2: Map I/O space.

```
KdMapKernel     *m_pKdAddress;

// Translate the base port address to a system mapped address.
// This will be saved in the device extension after IoCreateDevice,
// because we use the translated port address to access the ports.

pDevice->m_pKdAddress = new KdMapKernel(Isa,       // Bus type
                                        0,         // Bus number
                                        0x250,     // Local address
                                        0x4,       // Size of addr space
                                        IO_RANGE); // From the IO range

// Check if resources (ports and interrupt) are available
m_Resources.Init(Isa);
m_Resources.AddIo((ULONGLONG)pDevice->m_pKdAddress->GetSystemAddress(),
                  0x4));
m_Resources.AddInterrupt(STAT_IRQ);

// Report the resources to the operating system
Status = m_Resources.Report();

// Initialize the device
pDevice->m_pKdAddress->WriteByte(CONTROL_PORT, RESET);
```

Description

The **KdIrq** class describes the interrupt resource of a device. For every interrupt that a device has, a **KdIrq** class will be defined.

Description

Defines the DPC routine for an interrupt.

Description

This class enables you to read from / write to the Windows NT registry.

By default, the KdRegistry class looks for these entries under the ***\Registry \Machine\System\CurrentControlSet\Services\<drivername>*** branch of the registry. This default lookup point can be changed in the constructor or the **Init()** function. (See Chapter KernelDriver Class Model that explains the class references for more details).

Sample

The following KernelDriver sample reads two entries (**PortBase** and **PortCount**) from the NT registry.

```
KdRegistry Registry;

Registry.QueryDWORD(L"IoPortAddress", &PortBase, BASE_PORT);
Registry.QueryDWORD(L"IoPortCount", &PortCount,NUMBER_PORTS);
```

Description

Determining your driver's resources is done by either reading them from a pre-determined place in the registry, or by hard coding these resources in your code, or in Plug-n-Play cards, by the standard PnP configuration cycle.

After determining which resources your driver needs, it is a good practice to report the driver's resources to the NT registry. This is done with the **KdResources**class.

Description

The **KdSyncObject** class is the base synchronization class from which most synchronization objects in KernelDriver are derived.

Description

The **KdEvent** class defines events which can sync two or more objects by creating an event which can be set, waited on, queried or cleared.

Description

The **KdSpinLock** class is used to maintain the integrity of a data structure when running in separate threads (including multi-processor environments). The thread that first accesses the data structure acquires a lock on the spin lock, and raises the IRQL so that the scheduler will not interrupt it. In a single processor environment, this will automatically give this thread the assurance that it is the only one accessing this piece of code. On a multi-processor environment, a different processor may be at a lower IRQL and may want to access this code. Having a spin lock before the code, will lock out the other processor. The other thread that reaches the same data structure will wait (spin) until the first thread releases the lock.

Spin locks can cause serious problems if not used judiciously. In particular, no deadlock protection is performed and dispatching is disabled while the spin lock is held.

Therefore:

- The code within a critical region guarded by a spin lock must neither be page-able nor make any references to page-able data.
- The code within a critical region guarded by a spin lock can neither call any external function that might access page-able data nor raise an exception, nor can it generate any exceptions.
- The caller should release the spin lock as quickly as possible.

Sample

The following KernelDriver sample acquires a spin lock, does some data manipulation, and releases the lock so that other threads will be able to access this data.

```
{

KdSpinLock spin;

DebugDump(DBG_DIAG1, ("Entered AsyncCancel.\n");

// Acquire the spin lock for the wait queue.
DebugDump(DBG_DIAG1, ("Getting spin lock for wait queue.\n"));
spin.Lock();

// Now remove the irp from the queue completely.
DebugDump(DBG_DIAG1,("Removing Irp from queue.\n"));
RemoveEntryList(&Irp->Tail.Overlay.ListEntry) ;

// Release the spin lock.
DebugDump(DBG_DIAG1, ("Releasing spin lock.\n");
spin.Release();

...

}
```

Description

KdMutex provides a mutual exclusive lock, which can protect a sensitive code section, so that only one thread accesses this section at any time. A mutex is acquired by an **Acquire()**, and is released with a **Release()**.

Description

A semaphore is differentiated from the other synchronization methods in that each semaphore has a 'count' to it. When a semaphore is created, its count is zero. When a semaphore is signalled, the count is incremented. When a semaphore is waited upon, it waits only if the count is zero, otherwise it decrements the count and passes through.

When the code reaches a wait on a semaphore, it either passes through the wait and continues execution (when the semaphore count is above zero), or it will stop (when the semaphore count is zero) and let the scheduler pass the execution to a different thread. When other threads signal the semaphore, the execution will return to the original thread waiting on the semaphore, and the wait will pass through.

Description

The **KdTimer** class enables you to specify the absolute or relative time at which the timer is to expire. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes. You may query the status of the timer. It is signalled when the timer expires.

Description

The **KdTimedCallback** is derived from the **KdTimer** class. It enables you to specify the absolute or relative time at which the callback function you have specified to it will be called as soon as conditions permit.

```
if (UseATimer)

{

      m_KdTimer_TimerDpc.Set(TotalTime);

     ...

}
```

Description

The **KdThread** object is used by drivers to create a new execution thread.

```
class KdMyDevice : public KdDevice
{
public:
    KdMyDevice();
    KdThread m_Thread;                    // The thread object
    VOID Thread_Function( PVOID pArgs );  // The thread execution
                                          //function
};

KdMyDevice::KdMyDevice()
{
         // Create the thread, and set the function that will
         // run in that thread
         m_Thread.Create((KD_THREAD_CALLBACK)Thread_Function, this);

}

        VOID KdMyDevice::Thread_Function( PVOID pArgs )
        {
             LARGE_INTEGER LI;
             LI.QuadPart = -10000000; // 1 Second. The Minus
                                      // indicates relative time.
             DebugDump(DBG_DIAG2, ("Thread_Function A\n"));
             for ( int I = 0 ; I < 10 ; I++ )
             {
                 KeDelayExecutionThread( KernelMode, FALSE, &LI );
                 DebugDump(DBG_DIAG2, ("."));
             }
             DebugDump(DBG_DIAG2, ("Thread_Function B\n"));

             // Terminate this thread
             m_Thread.Terminate( STATUS_INSUFFICIENT_RESOURCES );
             DebugDump(DBG_DIAG2, ("Thread_Function C -
                                    NOT SUPPOSED TO BE CALLED\n"));
}
```

```
void KdMyDriver::Unload()
{
    NTSTATUS s = ((KdMyDevice *)GetFirstDevice())->m_Thread.TerminateStatus();
    DebugDump(DBG_DIAG2,("Thread_Function D = 0x%x \n", s));

    KdDriver::Unload();

}
```

Description

In the NT DDK, all string manipulation is performed using the UniCode strings (UNICODE_STRING). UniCode strings are constructs which describe the string's current size, maximum size, and the string buffer itself. This makes working with strings extremely tedious and time consuming.

KdString is a class which provides 'CString'-like string handling. The **KdString**class enables you to use the printf() string formatting methods (%s, %d, ...), concatenating strings, automatically allocating and freeing memory for strings, type casting strings between the KdString types and the UniCode types and to PCWSTR (pointer to const wide string) and more.

Sample

The following KernelDriver sample initializes and formats a string. (Code taken from the serial driver sample). As you can see, this is very similar to using the CString in Win32 MFC.

```
KdString sNameString;


sNameString.Format(L"\\Device\\ \%s",(PCWSTR)ConfigData->sNtNameForPort);

...
```

The code below demonstrates what it is like to achieve the same results using the NT DDK: (Code taken from the Serial sample in NT's DDK)

```
...
RtlInitUnicodeString(
        &uniNameString,
        NULL
        );

   uniNameString.MaximumLength = sizeof(L"\\Device\\")+
        //ConfigData->NtNameForPort.Length+sizeof(WCHAR);
   uniNameString.Buffer = ExAllocatePool(
                              PagedPool,
                              uniNameString.MaximumLength
                              );

  //
  // The only reason the above could have failed is if
  // there wasn't enough system memory to form the UNICODE
  // string. //

if (!uniNameString.Buffer) (

     SerialDump(
         SERERRORS,
         ("SERIAL: Could not form Unicode name string for %wZ\n",
            &ConfigData->NtNameForPort)
          );
     SerialLogError(
     DriverObject,
      NULL,
      ConfigData->Controller,
```

```
       SerialPhysicalZero,
        0,
        0,
        0,
        4,
        STATUS_SUCCESS,
        SERIAL_INSUFFICIENT_RESOURCES,
        0,
        NULL,
        0,
        NULL);

    ...

    return STATUS_INSUFFICIENT_RESOURCES;

}

//
// Actually form the Name.
//

RtlZeroMemory(
    uniNameString.Buffer,
    uniNameString.  MaximumLength
    );

RtlAppendUnicodeToString(
    &uniNameString,
    L"\\Device\\"
    );

RtlAppendUnicodeStringToString(
     &uniNameString,
     &ConfigData->NtNameForPort
     );
...
```

Description

Many device drivers use linked lists for managing custom queues, and FIFOs. textbfKdList provides a clean interface to creating custom linked lists.

The **KdList** class is a template class, so that it can provide linked lists to any types of data entries.

Sample

The following sample uses textbfKdList to create a list which contains custom entries (Struct ENTRY). Another example use of KdList is in the Serial sample.

```
// Define the Entry
struct ENTRY {
    LIST_ENTRY listEntry;     // Must have this field
    PVOID Data;                // This is the data you need to list
} ENTRY;

// Create the class of the `ENTRY' list
class KdEntryList : public KdList<ENTRY>
{

       // Define the offset of the LIST_ENTRY in the ENTRY structure
       // (Not needed in this case, since the default is 0)
       KdEntryList () : KdList<ENTRY> (0) {}

};

{

       KdEntryList MyList;
       ENTRY ListHead, Entry1;

        ...

        // Add an entry
        MyList.AddTail(Entry1) ;

        ...

        // Remove the list's head
        ListHead = MyList.RemoveHead();

         ...

}
```

Description

The **KdIrpList** is a linked list, specific for IRP queuing. An IRP queue is available also by the OS (The operating system's IRP list is accessible through some KdDevice member functions - see the class reference for details). However, drivers might need to create custom IRP queues, where different priorities or rules may apply, or when several IRP queues need to be implemented.

Sample

From the 'Async' sample. In this sample, a 'Standby' IRP list is created, and later dumped back into an existing IRP list (m_WaitQueue).

```
KdIrpList StandbyList;
DebugDump(DBG_DIAG1, ("Entered AsyncCleanup.\ n")) ;

   // Enter a loop to empty out the current queue.
   // We will check the file
   // objects to see if they match. If they do,
   // then we will cancel the irp. If not, then we will put
   // the irp in the standby queue and when
   //done with the device extension queue, we will requeue
   //the standby queue back into the main queue.

DebugDump(DBG_DIAG1,("Entering WaitQueue loop.\ n"));
while (!m_WaitQueue.IsEmpty())
{

    // Reconstruct the Irp.
    DebugDump(DBG_DIAG1, ("Reconstructing IRP.\ n")) ;
    rcIrp = m_WaitQueue.RemoveHead();

    // Check to see if this is an IRP we need to cancel.

    DebugDump(DBG_DIAG1, ("Checking for match on file object.\ n"));
    if (rcIrp.FileObject() == Irp.FileObject())
    {

       ...

     }

     else
     {

       // Not one of ours. Put it in the standby queue.
       DebugDump(DBG_DIAG1, ("No match. Standby queue.\ n")) ;
       StandbyList.AddTail(rcIrp);
```

```
        // Continue on our merry way.

      }

}

// Ok, the WaitQueue is empty. Now re-queue
//anything we did not cancel.
DebugDump(DBG_DIAG1,("Requeuing irps not cancelled.\ n"));
while (!StandbyList.IsEmpty())

{

      // Take the head irp, reconstruct it, and put
      // it back in the wait queue.

      DebugDump(DBG_DIAG1, ("Irp transiting to WaitQueue.\ n")) ;
      m_WaitQueue.AddTail( StandbyList.RemoveHead() );

}
```

Description

The **KdFifo** class is derived from KdList, and provides a clean FIFO interface.

Description

The **KdFile** class simplifies the access to files from the kernel mode in Windows NT. It provides a user mode - like API for accessing the files, yet can provide all of the low level functionality you would expect. The **KdFile** provides a simple prototype for every file access command, and provides an additional robust prototype for each command which includes all of the available file access options. The simple prototype uses intelligent defaults to achieve the desired access mode.

Sample

```
class KdMyDevice : public KdDevice

{

public:
   KdMyDevice();
      KdFile m_File;
   NTSTATUS DumpFile();

};

NTSTATUS KdMyDevice::DumpFile()
{

    NTSTATUS Status;
    UCHAR ST[200];

    Status = m_File.OpenRead(L"\\??\\c:\\kdtest.txt");

    if (!NT_SUCCESS(Status)) { DebugDump(DBG_ERRORS,
        ("Error in OpenRead(), 0x\%x\n",
        Status)); return Status; }

    ULONG BytesRead = 0;

    Status = m_File.Read( ST, 180, & BytesRead );

    if (!NT_SUCCESS(Status)) { DebugDump(DBG_ERRORS,("Error in Read(),
       0x\%x\n",Status)); return Status; }

    KdPrint( ("BytesRead = %d, ", BytesRead ));

    if ( BytesRead > 0 && BytesRead < 190 )
    {
    ST[ BytesRead ] = 0;

    DebugDump(DBG_DIAG1, ("Read String = %s\n",ST ));
```

```
    }
    Status = m_File.Close();
    if (!NT_SUCCESS(Status))
    {
         DebugDump(DBG_ERRORS, ("Error in Close(),
                   0x%x\n", Status));
         return Status;
    }

return Status;

}
```

Description

KdPhysAddr is a conversion utility which can convert between physical addresses, ULONGLONG (64 bit integer), LARGE_INTEGER(struct that contain 64 bit integer) and PLARGE_INTEGER (pointer to LARGE_INTEGER struct). For more information, please see Chapter KernelDriver Class Model that explains the class references.

Description

Each contiguous buffer in the virtual memory is actually composed of physical memory pages which are not necessarily contiguous. This class wraps the MDL structure.

The **KdMem** class is used to describe and provide services to a virtual buffer. It contains a pointer to the buffer in the virtual memory, and the list of non-contiguous physical memory pages.

Description

WinDbg (Microsoft's Kernel Debugger), enables you to view debug messages which are sent from kernel mode drivers. **DebugDump()** enables you to easily send messages from your driver, which will be displayed by WinDbg.

Use DebugDump to throw debug messages from within your driver source code. By setting the DriverDebugLevel parameter to different values, DebugDump will send only the messages that correspond to that debug level. It is recommended that instead of hardcoding the DriverDebugLevel parameter in your code, you read it from a registry value. This will enable different levels of verbosity in debugging your driver.

DebugDump sends additional information for debugging (the file name and line number of the line that called DebugDump()). The KernelDriver source code itself contains many debug messages in different levels (trace, info, warning and errors). It helps to view KernelDriver's debug messages when debugging your driver. This is possible by setting the DriverDebugLevel to the appropriate levels described.

Sample

```
DriverDebugLevel = DBG_DIAG1;
{
      ...
      DebugDump (DBG_DIAG1, "This is a debug message number %d\n", i);
      ...

}
```

Description

The memory compare utilities include **MemCompare** which enables you to compare two memory ranges and determine if they are equal, overlapping or disjoint. Other memory utilities available in KernelDriver are **IsPhysicalZero**and **IsPhysicalMax**. These enable the developer to check whether a given physical address is zero, and whether the buffer exceeds the maximum size of the physical memory (see details in Chapter KernelDriver Class Model that explains the class references).

Sample

```
if (MemCompare(
               New->Controller,             // Buffer #1 Address
               New->SpanOfController,       // Buffer #1 Length
               OldConfig->Controller,       // Buffer #2 Address
               OldConfig->SpanOfController  // Buffer #2 Length
               ) != ADDRESSES_DISJOINT)

{

       ...

}
```

Description

In the kernel mode, there are several memory 'pools' from which the operating system allocates memory:

- **PagedPool** - Allocates memory which may be paged out to disk. For a temporary buffer, used by the **DriverEntry** or **Reinitialize**routines to contain objects, data, or resources necessary for initialization, if the buffer is released before the caller returns, or for a storage area that will be accessed only by one or more driver-created threads.

- **NonPagedPool** - Allocates memory which will not be paged. For any objects or resources not stored in a device extension or controller extension that the driver might access while running at IRQL > APC_LEVEL.

- **NonPagedPoolCacheAligned** - For a permanent I/O buffer the driver uses, such as a SCSI class driver's buffer for request-sense data.

- **NonPagedPoolCacheAlignedMustS** - For a temporary, but critically important, I/O buffer, such as a buffer containing initialization data for a physical device that must be present for the system to boot.

- **NonPagedPoolMustSucceed** - For a temporary, but critically important, storage area, that the driver will release as soon as possible, such as memory that a driver needs to correct an error condition because it would corrupt the system otherwise.

- **PagedPoolCacheAligned** - For a file system driver's I/O buffer that it locks down and passes in an IRP requesting a DMA transfer by an underlying mass-storage device driver.

of the memory allocations will require non-paged memory. The reason for this is as follows:

When you attempt to access paged-memory from your driver, the OS will interrupt, signaling that a page-fault has occurred. If this happens while your driver is in an IRQL which is higher than the page-fault IRQL, this interrupt will not be processed, and the page fault will not be able to bring the desired memory. Performance is also a consideration for using non-paged memory - when paged memory is accessed, a page fault will occur. The page fault is a considerable hindrance to performance.

KernelDriver provides memory allocation and freeing utilities, which were designed to be as similar as possible to their user mode equivalents.

malloc() and **free()** can be used exactly as they are used in the user mode. This provides for portability for some of your user mode code. The malloc() function allocates memory from the non-paged memory pool.

new() and **delete()** may also be called as in the user mode. However, the developer must be aware of the following differences:

1. By default, **new()** will allocate memory from the non-paged memory pool. It is also possible to use new to allocate paged memory.

2. Memory allocation cannot be performed during an ISR.

Sample

From the "Async" sample code.

```
...

// Create an IRP Queue
KdIrpList m_WaitQueue;

....

// Now insert in the Irp into our queue.
DebugDump(DBG_DIAG1, ("Inserting Irp into queue.\n")) ;
m_WaitQueue.AddTail(Irp) ;

....
// Enter a loop to empty out the current queue.
DebugDump(DBG_DIAG1, ("Entering WaitQueue loop.\n"));
while (!m_WaitQueue.IsEmpty())
{

    ....

    Irp = m_WaitQueue.RemoveHead();

    ....
}
```

Description

It is possible to change the device driver's functionality by connecting a filter device on top of it. The filter device intercepts and processes messages before they reach the device driver. It usually interacts with the device driver which is below it in the driver stack, and does not directly interact with the hardware itself.

The **KdFilterDevice** class is derived from **KdDevice** . This enables **KdFilterDevice** to access the hardware directly.

When initializing the filter device, you must provide **KdFilterDevice** with the name and type of the lower device (which the Filter device is 'on top' of). The **KdFilterDevice** then connects to the lower device, and from this point on, it intercepts all the messages (IRPs) that are directed to the lower device.

KdFilter automatically defines a 'lower device' object which represents the device that the filter driver is attaching to, and provides mechanisms for sending the IRPs down to the lower device.

Description

The KdLowerDevice class should be used when communicating with other devices. To send messages to other devices, a lower device is created and attached to the device with which the communication channel is requested. The lower device object attaches to the other device, and provides a pipe through which IRPs can be sent.

Description

This class is the base class of a miniport driver. A specific miniport driver must be derived from this class. There should be only one instance of the KdNdisMiniport class created for a driver, which should be created in **DriverEntry()**. This class registers itself with NDIS and NDIS will then start the initialization process.

Initialization

After the miniport class has been created its **Init()** function must be called in order to register the miniport with NDIS. This will start the whole initialization process of creating and initializing an adapter for each NIC supported.

Behind the scenes

After your miniport class is created and its **Init()** function is called, NDIS will call the miniport driver's initialization call-back function for each NIC supported by your miniport driver (The list of supported NIC cards for a driver is obtained by NDIS from the registry). Each time NDIS calls this function, your miniport will create an adapter class to represent the NIC involved and the miniport will store this adapter in an internal adapter list. For this to happen, your miniport class must override the **CreateAdapter()** function. This function must create the specific adapter relating to the NIC involved. After the initialization is completed, the primary task of the miniport class will be to dispatch the NDIS requests to the specific adapter objects.

Description

This class is the base class for adapters (i.e. NICcards). Each adapter represents a NIC and is created by the miniport class during the initialization process. After the initialization process is completed, NDIS will communicate with a NIC via a set of virtual functions overridden by the adapter derived class representing that NIC. This set of functions defines how the adapter will handle interrupts, how it will send or receive data etc.

Initialization

After an adapter class is created, it will be initialized via a call made to an Init() function supplied by the adapter class. The Init() function should do all the initialization of the adapter class and the NIC such as registering the port addresses with NDIS, initializing the interrupt and resetting the card. After the initialization process, the adapter should be fully operational and ready to receive or send data and handle interrupts.

Quick Sample ... NE2000 NDIS Miniport Driver

The following are code excerpts which display how an NDIS driver should be written for an NE2000 network adapter using the KernelDriver's Miniport classes:

```
//The Miniport for ne2000
class KdNe2000Miniport:public KdNdisMiniport
{
public:

    KdNe2000Miniport(DWORD dwOptions = 0);
    virtual ~KdNe2000Miniport();

    //overridden to create the specific ne2000 adapter
    //and specific initializations
   virtual NDIS_STATUS CreateAdapter(KdNdisAdapter **pAdapter,NDIS_HANDLE
                                     WrapperConfigurationContext}

  {
      *pAdapter = new KdNe2000Adapter;
      if(*pAdapter)
          return NDIS_STATUS_SUCCESS;
      else
          return NDIS_STATUS_FAILURE;
  }
 .
 .
 .
};

class KdNe2000Adapter : public KdNdisAdapter
{

public :

    KdNe2000Adapter();
    virtual ~KdNe2000Adapter();

    // This is used by the base class KNdisMiniport during initialization
    //(in BaseInit) to tell NDIS the medium type supported by this adapter
    virtual NDIS_MEDIUM GetMediumType() {return NdisMedium802_3; }
```

```
    // The ne2000 data structure
    NE2000_ADAPTER Info;

private:

    static UINT Ne2000SupportedOids[];
    // Used for padding short packets with blanks
    static UCHAR BlankBuffer[];

    // The overriden Init().Called by the base class
    // This is where we read the configuration info for setting the adapter.
    virtual NDIS_STATUS Init(PNDIS_STATUS OpenErrorStatus,
                             KdNdisConfiguration *pConf,
                             NDIS_HANDLE WrapperConfigurationContext);
    {
     // Implementation of all the initialization is done here . }
    }

    // These are overridden handlers which will be called by the NDIS
    // for interacting with the NIC
    virtual VOID DisableInterruptHandler();
    virtual VOID EnableInterruptHandler();
    virtual VOID HaltHandler();
    virtual VOID HandleInterruptHandler();
    virtual VOID ISRHandler(OUT PBOOLEAN InterruptRecognized,
                            OUT PBOOLEAN QueueDpc);
    virtual NDIS_STATUS QueryInformationHandler (IN NDIS_OID Oid,
                                IN PVOID InformationBuffer,
                                IN ULONG InformationBufferLength,
                                OUT PULONG BytesWritten,
                                OUT PULONG BytesNeeded);

    virtual NDIS_STATUS ResetHandler(OUT PBOOLEAN AddressingReset);
    virtual NDIS_STATUS SendHandler(IN PNDIS_PACKET Packet, IN UINT Flags);
    virtual NDIS_STATUS SetInformationHandler(IN NDIS_OID Oid,
                                IN PVOID InformationBuffer,
                                IN ULONG InformationBufferLength,
                                OUT PULONG BytesRead,
                                OUT PULONG BytesNeeded);
```

```
    virtual NDIS_STATUS TransferDataHandler(OUT PNDIS_PACKET Packet,
                            OUT PUINT BytesTransferred,
                            IN NDIS_HANDLE MiniportReceiveContext,
                            IN UINT ByteOffset,
                            IN UINT BytesToTransfer);
.
.
.
}


NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    // Receives the status of the NdisMRegisterMiniport operation.
    NDIS_STATUS Status;

    KdNe2000Miniport *pMiniport = new KdNe2000Miniport();
    if(!pMiniport)
         return STATUS_UNSUCCESSFUL;

    //Register the miniport with NDIS
    Status = pMiniport->Init(DriverObject,RegistryPath);

    if (Status != NDIS_STATUS_SUCCESS)
    {
       delete pMiniport;
       return STATUS_UNSUCCESSFUL;
    }

    return STATUS_SUCCESS;
}
```

Overview

WinDbg is a kernel mode debugger which is freely supplied by Microsoft in the NT DDK, and through the microsoft web site at`http://www.microsoft.com/msdownload/PLATFORMSDK/SDKALL/09000.HTM`.

WinDbg runs on a host machine (the machine where your debugger runs) connected to a target machine (the machine where your driver runs) via a serial cable link.

The following sections provide the basics needed for establishing a debugging environment, and using WinDbg to debug your driver. For more in-depth information on WinDbg and on debugging device drivers, see the **Microsoft MSDN help**.

Connect the Host and Target Machines

Use a Null Modem serial cable to connect the host and target machines. After connecting the two machines, test your connection (See Appendix Using a Serial Cable for Kernel Debugging (Null Modem Serial Cable) for details on how to create a Null Modem cable, and how to test the connection).

On the Target Machine

- Install the checked build of Windows NT. It is necessary that this be the same version of Windows NT as you install on the host.

The DDK provides two versions of the Windows NT operating system, called the**free build** and the **checked build**. Getting Started explains how to install each version.

The free build of Windows NT is the retail version that end users buy. Free binaries are built with full optimisation and contain a minimal set of debugging symbols, such as function entry points and global variables.

The checked build is shipped only with the DDK; it is used in debugging drivers and other system code. Checked binaries provide error checking, argument verification, and system debugging code not present in the free binaries.

- Install your driver. (See Section Installing, Loading and Registering Your Driver that explains how to install, load and register your driver.).

- Enable kernel debugging for this system using the appropriate options for the port that is connected to the host machine.

enable kernel debugging on the target machine (For an x86 based machine), modify a line in the **[operating systems]** section of boot.ini. In it, you can specify the following options:

- **DEBUG** - Enables kernel debugging.

- **NODEBUG** - Disables kernel debugging. If no options are specified, **NODEBUG**is implied. **NODEBUG** disables **DEBUGPORT**, **BAUDRATE**, and **CRASHDEBUG**.

- **DEBUGPORT=Port** - Specifies the serial port (Port is COM1, COM2, and so forth.) used on the target machine. This parameter is the functional equivalent of **DEBUG** with the additional ability to override the default debug COM port. The default port without using **DEBUGPORT** is COM2, if it exists, on an x86 machine.

- **BAUDRATE=**BaudRate - Specifies the baud rate (BaudRate is 9600, 19200, etc.) used by the target machine. Use the highest rate that works for your machines. Specifing BAUDRATE enables kernel debugging and renders **DEBUG** redundant.

- **CRASHDEBUG** - Causes the debugger to activate only when the system bugchecks.

- **MAXMEM=SizeInMB** - Specifies the amount of RAM to be made available to the system in megabytes. This option can be used to simulate a system with a low amount of physical memory to test the performance and operation of a driver under those conditions. For example, you can restrict a 16 MB machine to 10 MB.

- **SOS** - Causes the OS loader to display the names of the drivers as they load when Windows NT is booting.

Example:

OSLOADOPTIONS= MAXMEM=10 DEBUGPORT=COM1 BAUDRATE=115200

The following is an equivalent entry, for an Intel x86 platform, in boot.ini.

multi(0)disk(0)rdisk(0)partition(1)\NT="Windows NT" /MAXMEM=10 /DEBUGPORT=COM1

/BAUDRATE=115200

On the Host Machine

- Install the free build of Windows NT.
    - Install WinDbg.
    - Copy the symbols from the target machine to the host machine into the symbol tree.

must have access to the symbols for the target machine so that the system can interpret the information on the target machine. The symbols for the Windows NT kernel and drivers are stored in ***.dbg*** files; the symbols for your driver are stored in the driver's ***.sys*** file itself, or may have been split off into a ***.dbg*** file.

- These symbols must be copied into a specific tree layout on disk in order for WinDbg to access them correctly. The root directory of the symbol tree may be created at any level in the disk drive's existing directory tree structure. However, it must specifically be named **SYMBOLS**. The Symbols directory must then have three subdirectories specifically named **DLL**, **SYS**, and **EXE**. When the directory tree is created, it will have the following layout:

```
\SYMBOLS

    \DLL
    \SYS
    \EXE
```

Symbol files for the system can be found on the Windows NT system CD-ROM in the ***\SUPPORT*** directory. The following files should be copied into the directories specified for each file:

1. **ntoskrnl.dbg** should be copied into ***\SYMBOLS\EXE***. If the computer is a multiprocessor machine, it will be necessary to copy **ntkrnlmp.dbg**to **ntoskrnl.dbg** in the ***\SYMBOLS\EXE***directory.

2. **hal.dbg** should be copied into ***\SYMBOLS\DLL***. If the computer does not use the standard x86 HAL, then the **appropriate HAL symbols** for that machine should be copied to **hal.dbg** in the ***\SYMBOLS\DLL***directory.

3. **yourdriver.sys** (or **yourdriver.dbg**, if the symbols have been split off into a separate ***.dbg*** file) should be copied into ***\SYMBOLS\SYS***.

4. **otherdrivers.sys** with which your driver interacts (or **otherdrivers.dbg**, if the symbols have been split off into separate ***.dbg*** files) should be copied into ***\SYMBOLS\SYS***.

- Specify the symbol search path with the User DLLs command from the ***Options***menu of WinDbg. If you start WinDbg KD from the command line, you can use the***-y*** command-line option to specify the symbol search path. The path should point to the named Symbols directory. For example, if Symbols created a subdirectory of ***C:\Foo***, then the path to apply would be ***C:\Foo\Symbols***.

- Copy your driver's source to the host machine.

- Start WinDbg with kernel debugging enabled.

There are several methods that you can use to start WinDbg as the kernel debugger.

METHOD 1:

Start WinDbg from the command prompt with the kernel debugging command-line options

start WinDbg platform port speed where platform is the target machine type (i386, MIPS, PPC), port is the COM port (COM1 ... COMn) to which the serial cable on the host is attached, and speed is the com port speed (9600, 19200, 57600, ...).

METHOD 2:

Start WinDbg from the command prompt with **ntoskrnl** or **ntoskrnl.exe**as a command-line option:

start windbg ntoskrnl[.exe]

Use the ***Kernel Debugger*** dialog from the ***Options***menu to specify the target platform, com port, and speed.

METHOD 3:

Start WinDbg by double-clicking the WinDbg icon in the Program Manager. Use the ***Kernel Debugger*** command from the ***Options***menu to enable kernel debugging and specify the target platform, com port, and speed.

Once you have set up a kernel debugging session with the appropriate parameters, save it as a named workspace with the ***Save As*** command from the***Program*** menu. You can then start a kernel debugging session with those parameters by specifying the workspace name on the command line:

start windbg -w workspacename ntoskrnl

- Using the ***User DLLs dialog box*** from the ***Options***menu, specify the symbol search path to the symbol tree where you copied the required symbols.

- Using the ***Debug dialog box*** from the ***Options***menu, specify the source search path to where you copied the source for your driver.

- Select ***Go*** from the ***Run*** menu and wait for '**KD: waiting to connect...**' message to appear in the Command window.

START WINDBG

- Boot the target machine and select the version of Windows NT for which you enabled kernel debugging. To have WinDbg automatically stop the system at initial load time, you can set the ***Initial Breakpoint*** option in the ***Options***dialog.

- To break into WinDbg, press CTRL+C on the host machine, or press SYSRQ on the target machine. To continue execution, select ***Go*** from the ***Run***menu.

Options

- **-a**: Ignore all bad symbols (but still print warning message).
    - **-g**: Go now; start executing the process.
    - **-h**: Causes child processes to inherit access to WinDbg's handles.
    - **-I**: Ignore workspace; like running without any registry data.
    - **-k[platform port speed]**: Run as a kernel debugger with the specified options:
        1. **platform** is the target machine type (i386, MIPS)
        2. **port** is the com port (com1 ... com4)
        3. **speed** is the com port speed (9600, 19200, 57600, ...)
- **-l [text]**: Sets the window title for WinDbg.
    - **-m**: Start WinDbg minimized.
    - **-p id**: Attach to the process with the given ID.
    - **-s [pipe]**: Start a remote.exe server, using the named pipe in pipe.
    - **-v**: Verbose option; WinDbg prints module load and unload messages.
    - **-w name**: Load the named workspace.
    - **-y path**: Search for symbols along the specified path(s). You can specify multiple paths by separating them with a semicolon.
- **-z crashfile**: Debug the specified crash dump file.
    - **filename[.ext]**: Program to debug or file to edit.

DUMPCHK Options

The following are valid program options:

- **-?:** Displays a help message. This overrides any other option specified.
    - **-v**: Verbose mode.
    - **-p**: Print header of crash dump file only, does not validate the file.
    - **-q**: Perform a quick test only.
    - **CrashDumpFile:** The name of the crash dump file.

DUMPREF [options] SourceDumpFile ShareName SymbolPath

DUMPREF is used to create a reference file to point to a dump check file in another location.

DUMPREF Options

The following are valid program options:

- **-?**: Displays a help message. This overrides any other option specified.
    - **-d** FileName: Specifies a file name for the destination.
    - **SourceDumpFile**: The name of the crash dump file.
    - **ShareName**: The name of the network share where the crash dump file is located.
    - **SymbolPath**: The path to the symbols for the crash dump file.

Comments

This utility creates a small reference file that points to the actual crash dump file. You can use the reference file as though it were the crash dump file; WinDbg KD will find the actual crash dump file out on the network. This lets you send the reference file, which is only a few bytes long, to someone for debugging, rather than sending the actual crash dump file, which may be many megabytes long.

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.

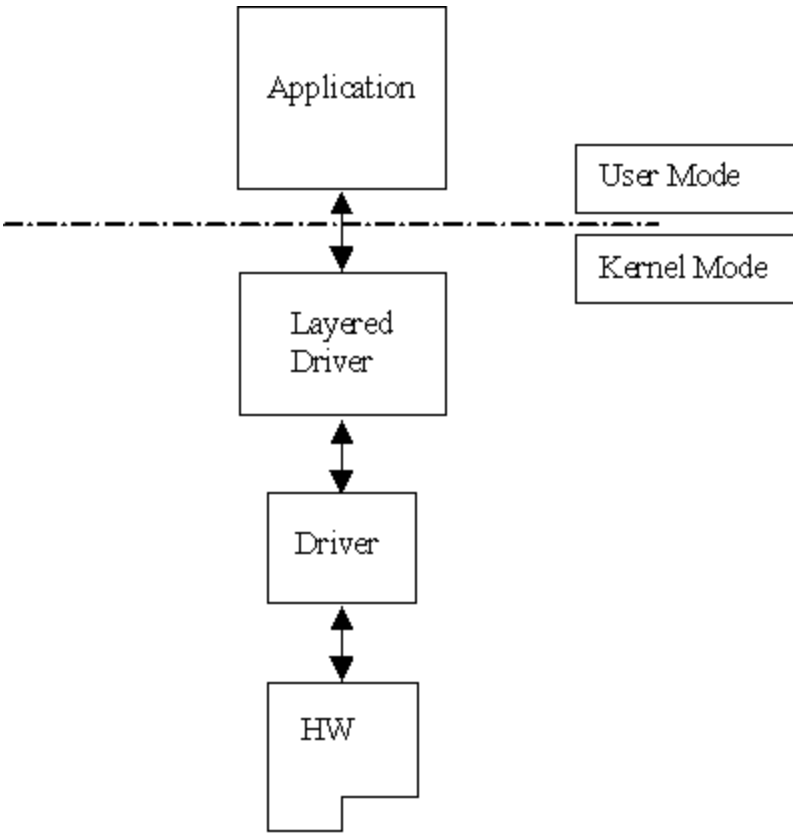

Figure 1.2: Layered Drivers

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 1.3: 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.

Restricting hardware access via /dev/windrvr

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*).

List of KernelDriver samples

KernelDriver's samples should be used as a quick-start for your own driver development, and for finding examples of using the KernelDriver classes. The samples may be found under the ***\KernelDriver\Samples*** directory.

Following is a quick look-up list of the samples available with KernelDriver. Check the Jungo web site `http://www.jungo.com/kerneldriver.html` for more samples that are periodically released.

- **SIMPLE** - Sample driver and a user mode application that calls the driver. Use this sample to learn how a basic device driver is built.
- **INT** - Sample driver for using interrupts. The driver implements connecting / disconnecting an interrupt, Interrupt Service Routine, and Deferred Procedure Call.
- **PORTIO** - Sample driver for accessing an I/O range. Implements reading from / writing to a generic port, and reporting resources. This sample includes a user mode application that reads and writes from the port.
- **MAPMEM** - Sample driver for accessing a memory range. Implements mapping/un-mapping physical memory to user space. This sample includes user mode applications that read and write from the memory.
- **SYS_THREAD** - Sample driver for creating threads in the kernel.
- **FILE** - Sample driver for reading from and writing to a file.
- **ASYNC** - Shows how to notify a user mode application or service of an asynchronous event from kernel mode. Implements queuing IRPs.
- **DELAY** - Shows how to delay a thread.
- **SERIAL** - A complete serial driver. This driver can replace the existing NT serial driver.
- **PLX9050** - Sample contains a kernel mode driver that accesses PLX9050 based hardware and an application that diagnoses the hardware through this sample driver.
- **PLX9054** - Sample contains a kernel mode driver that accesses PLX9054 based hardware and an application that diagnoses the hardware through this sample driver.
- **PLX9080** - Sample contains a kernel mode driver that accesses PLX9080 based hardware and an application that diagnoses the hardware through this sample driver.
- **NE2000** - Shows how to write a NE2000 NDIS Miniport Driver.

Directory Structure

Open a new directory for your driver. Under your new directory create two sub directories:
sys - Where your device driver code will be stored.
exe - Where your application (that accesses this device driver) will be stored.

Writing the Driver

- Create a new file with the **DriverEntry()** function. This function is the main entry point into your device driver (similar to how the main() function in C is the main entry point into your application). In this function, a new driver object must be created. The base class for creating a driver in the KernelDriver class library is called **KdDriver**. If you have access to the source code of the KernelDriver class library (contact sales@jungo.com for information), you can see the implementation of this class in the file *kddriver.cpp*.

Sample 1

```
// Write the Windows NT DriverEntry procedure

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath)
{
    new KdDriver(Status, DriverObject, RegistryPath);

    if (!NT_SUCCESS(Status) && KdDriver::Driver())
        KdDriver::Driver()->Unload();

     return Status;
}
```

- Now, all system calls to your driver will be forwarded to the driver's dispatch functions (by default, all driver dispatch functions forward system requests to the devices it controls).

- Next, define all of the devices (using KdDevice) that your driver controls. Each device communicates with the application level through a file handle. The application calls **CreateFile()** to initiate communications, and **ReadFile()**,**WriteFile()** and **DeviceIOControl()** to communicate with the device. In most cases, a driver will have only one device. If your driver has distinct separate communication paths, it may need more than one device. For example, if you have two serial ports, the serial driver will create a **Com1**device and **Com2** device, which will communicate with applications through files called Com1 and Com2.

- Each KdDevice object created, is associated by the operating system to the one driver that is defined in the project. Each KdDevice has a unique name. When an application wants to communicate with the device, it sends an I/O control signal (IOCTL) to the device by opening a file handle to a file.

Sample 2

```
// Open the Device called "Simple Driver" . .  . . .
hndFile = CreateFile( "\\\\.\\SimpleDriver",.....);
```

```
// Was the device opened?
if (hndFile == INVALID_HANDLE_VALUE)
    {
      . . . . . .
    }

IoctlResult = DeviceIoControl(
        hndFile,    // Handle to device
        IoctlCode,  // I/O Control code for Read
    . . . .
        );

CloseHandle(hndFile); // Close the Device "file".
```

In each device, implement the dispatch routines that you need. One of these is the dispatch routines that handles the IOCTLs (The messages that the application sends to the driver).

Example on WinDriver API access from kernel mode

Access to the WinDriver API is implemented by the kernel module **windrvr.o / windrvr.sys / windrvr.vxd** and needs to be activated. Under Windows, the KernelDriver installation does this automatically. Under Linux, you need to build and insert the module windrvr.o into the kernel. We covered these steps in Chapter Installing KernelDriver.

For a code example of how to access the WinDriver API from within KernelDriver, we'll use the sample ***plx\9054*** under the KernelDriver directory. This directory has kernel mode driver samples for Windows NT/2000 (sys), Windows 95/98/ME (vxd) and Linux (linux_module). Please load up the source files in your favourite editor as we walk through this code. This section is specifically written for Linux but the steps for other operating systems are very similar.

Lets now step into p9054_driver.c as an example of how to access the WinDriver API.

Step 1 - #include "kd.h" - this is the basic KernelDriver include file. It also includes windrvr.h so you don't need to include it explicitly.

Step 2 - #include windrvr_int_thread.h (this contains the convenience wrapper functions and InterruptThreadDisable functions that are used for interrupt processing.

Step 3 - Make a call to ***KD_OpenWinDriver()*** - This opens a connection to the windriver module so that you can now access the full WinDriver API including interrupt handling.

Step 4: Link phase - you need to link in **wd_kd.o** (you can see this in the makefile under the **9054\ linux_module** directory). This will give you access to malloc, free (kernel mode implementations so you need not call kmalloc/kfree), KP_DeviceIoControl and WD_Open.

Step 5: Now you can make calls to ***WD_Open*** /***WD_Close***, These are used in the kernel in the same way as you see in the code for the user mode driver applications written using WinDriver.

Compiling under Windows

Your driver project directory should include a '***sys***' or '***vxd***' directory -- where the driver sources are found, and an '**exe**' directory -- where the application (that accesses this driver) is found. In each directory, a Microsoft Developer Studio project file is found (Created by DriverWizard, or available from the sample which was used as your skeletal driver). This file contains all the information that your development environment needs to recognize the project files, and to successfully compile the driver or application.

For the sample applications:

- In the sample's root directory you will find a '***.dsw***' file, which contains information about loading both the '.***exe***' and '***.sys***' projects.

- To build your sample, double click the '***.dsw***' file in your sample's directory. This will start up your Microsoft Developer's studio.

- A different alternative is to load the '***.sys***' and the '***.exe***' project files into your Developer's Studio, and to compile them separately.

Compiling under Linux

- Open an xterm window

- Change directory to the path where you generated the source code for the module project (say under /home/kdprj)

- cd /home/kdprj/linux/

    - Build the module kdprj.o - use the command *make*

- Move to the directory having the sample user mode diagnostics application *cd /home/kdprj/usr_mode/linux*

- Compile the sample diagnostics program kdprj_diag - use the command *make*

Installing under Windows

The following instructions take you through copying your driver to the drivers directory, and loading the driver to be used by the operating system (Where <DriverName> is your driver's name):

For Windows NT SYS drivers copy the **<DriverName>.sys** file that was created to the ***\winnt\system32\drivers***directory. This is where all the loadable drivers reside. For Windows VxD drivers, copy the **<DriverName>.vxd** file that was created to the ***\windows\system\vmm32*** directory. This is where loadable VxD drivers reside under Windows 95/98/ME.

From the command line type ***wdreg.exe -name<DriverName> -startup manual create start*** (this registers your driver in the NT registry and gets it started). More information about WDREG may be found in Section. For a Windows VxD driver, use the command line ***wdreg.exe -vxd -name<DriverName> -startup manual create***

Installing under Linux

To install the driver under linux, follow the steps given below:

- Copy the relevant driver module to /lib/modules/misc

- execute the command '/usr/local/WinDriver/util/wdreg <modulename>'

where <modulename> is the name of the module.

Running Your Driver under Windows

In the previous section, the wdreg utility already installed and loaded your driver into memory. But to actually run your driver, you have to send it commands from the user mode test application. So run the executable created in the '***exe***' directory. This is the sample application that accesses the sample driver you created.

Running Your Driver under Linux

In the previous section, the wdreg script already installed and loaded your driver into memory. But to actually run your driver, you have to send it commands from the user mode test application. So run the executable created in the '***usr_mode/linux***' directory. This is the sample application that accesses the sample driver you created.

Under Windows

You have the following options:

- Add the WDREG source code to your installation application. (The source code for WDREG is available in the ***\kerneldriver\samples\wdreg***directory). Rewrite the main() routine as a function that accepts a string argument in the same format as applications receive argc/argv arguments. Then call this function from within your application.

- Launch the wdreg.exe program with the appropriate arguments by using the *system()* system call, or the *exec()* system calls.

NOTE: YOU NEED TO RUN WDREG AS ADMINISTRATOR UNDER WIN NT / WIN 2000 SINCE WDREG ACCESSES PRIVILEGED REGISTRY ENTRIES FOR WHICH ADMINISTRATIVE RIGHTS ARE REQUIRED.

Under Linux

- Launch the wdreg script with the appropriate arguments by using the *system()* system call, or the *exec()* system calls.

Class KdDriver

Base class for a driver object.

Class KdDevice

Base class for a device object.

Class KdIrp

IRP class for communicating with drivers.

Class KdBusAddress

Translates physical bus addresses to physical CPU addresses.

Class KdMapProcess

Maps a physical memory into the virtual address space of a subject process.

Class KdDpc

Handle DPCs for the device.

Class KdIrq

Handles connecting / disconnecting of interrupts to interrupt service routines (ISRs), and Synchronization functions with the interrupt.

Class KdRegistry

Reading/Writing from/to the registry.

Class KdResources

Reports device resources usage to the Operating System.

Class KdSyncObject

This is the base class for KdEvent, KdMutex, KdTimer, KdSysThread, KdTimedCallback.

Class KdSpinLock

Implements spin locks.

Class KdEvent

Provides synchronization on events (Event can be set, waited on, queried or cleared).

Class KdMutex

Provides mutual exclusive locks on critical sections.

Class KdSemaphore

Provides semaphores.

Class KdTimer

Enables you to specify the absolute or relative time at which a timer is to expire.

Class KdTimedCallback

A timer object which calls a specified function when the timer expires.

Class KdMem

Describes the physical addresses of a virtual memory range. (MDL structure)

Class KdString

Provides easy handling of strings in the kernel (like CString in MFC).

Class KdFile

Implements reading to / writing from a file.

Class KdList

Template class for a linked list.

Class KdIrpList

Template class for a linked list of IRPs.

Class KdFifo

Template class for a FIFO queue.

Class KdPhysAddr

Wrap PHYSICAL_ADDRESS struct.
Do type casting.

Class KdLowerDevice

Act as a lower device for the driver. KdLowerDevice implements passing IRPs to the actual lower device.

Class KdFilterDevice

A base class for filter devices.

Class KdNdisMiniport

The base class of a Miniport driver.

Class KdNdisAdapter

The base class of a Miniport adapter.

Description

The NT kernel uses the driver object to manage one or more devices. Each driver object has several methods (Dispatch functions) which the OS calls upon certain events. When an I/O request to a device is made, the NT's I/O manager sends it to the driver which manages the device. The Driver then passes the request on to the requested device. The driver object also has other methods, such as initialization and shut-down.

The Driver object is the base of all drivers. Each device driver is made up of exactly one driver object, and one or more device objects.

The Driver object should be defined in the **DriverEntry()** routine. Any device objects that are defined later on in the project will be associated with this driver.

Initialization

KdDriver is the class that describes the driver. The KdDriver contains the main entry points into the device driver, through which the operating system invokes the various driver functionalities. You should derive a new driver class from the KdDriver class so that you can add your driver specific functionality to it. For example, the default behavior of the KdDriver class is to forward all I/O requests directly to the called device. If your driver wants to count the number of requests before dispatching them to the device, you will have to inherit the KdDriver class to a new class in which you will override the relevant dispatch routine.

Any special initialization should be made in the constructor of the new class. This includes the construction and initialization of new devices. This can also be done in the **DriverEntry()** function.

In the initialization stage, a driver will typically:

Query - the system for the presence of the devices it must control.

Allocate - system resources that these devices may require (Alternatively, this may be done by the devices).

Create - the needed devices.

Handling IRPs

When an I/O request packet (IRP) is handled by the I/O manager, it is sent to the driver that created the device for which the IRP is intended. By default, the driver will forward this IRP to the device. You may override this default behavior by overriding your driver's dispatch method. This should be done when your driver needs to process or modify the IRP being passed down to the device.

Description

The device class represents a logical I/O entity with which an application or other drivers can communicate. Although it is the driver class that initially receives the I/O request packet from the I/O manager, it is the device class which does the real processing of the request.

Every driver implements at least one subclass of the KdDevice class. In each subclass, the developer determines the device's behavior by overriding the device's default dispatch routines.

Symbolic links

When a device class is created, it is given a unique name which identifies it. This name is not visible to the user mode name space, which needs to open a file handle to this device. Therefore, a symbolic link must be made to the device. The symbolic link creates a name that is visible to the user mode name space, through which a handle to the device can be opened. By default the KdDevice class creates a symbolic link with the same name as the device. There may be multiple links referencing the same device.

Description

The I/O Request Packet (IRP) is the method by which kernel mode device drivers communicate with other drivers, and with NT's I/O manager.

Upon receiving a request from an application, the NT I/O Manager creates an IRP which describes this request. The IRP is sent to the driver, which may process it and send it on to the device. After processing the IRP, the device can do one of the following:

- Mark the IRP as completed, and return it to the application.
- Transfer the IRP to another device.
- Queue the IRP for later processing.

The IRP contains two parts - the **IRP** itself, and the **IRP stack**. The IRP stack contains information on the IRP as it moves between drivers. Each new device that the IRP goes through, adds information to the IRP stack. As the IRP traverses the chain of devices, the IRP stack pops the relevant entries.

KdIrp wraps the IRP structure, and allows easy access to the most commonly used entries in the complex IRP structure.