KRF-TECH

WINDRIVER FOR PLX I/O DEVICES V5.0
Supports PLX PCI 9054, PCI 9080, PCI 9052, PCI 9050, and PCI 9060 devices

Developer's Guide
COPYRIGHT

Copyright © 1997-1999 KRFTech LTD. All Rights Reserved

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

Windows, Win32, Windows 95, Windows 98, Windows NT and Windows 2000 are trademarks of Microsoft Corp. WinDriver and KernelDriver are trademarks of KRFTech. Other brand and product names are trademarks or registered trademarks of their respective holders.
# Table of Contents

WINDRIVER OVERVIEW  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION TO WINDRIVER</td>
<td>5</td>
</tr>
<tr>
<td>BACKGROUND</td>
<td>6</td>
</tr>
<tr>
<td>WINDRIVER OVERVIEW</td>
<td>7</td>
</tr>
<tr>
<td>WINDRIVER FEATURE LIST</td>
<td>10</td>
</tr>
<tr>
<td>WINDRIVER ARCHITECTURE</td>
<td>12</td>
</tr>
<tr>
<td>WHAT PLATFORMS DOES WINDRIVER SUPPORT?</td>
<td>13</td>
</tr>
<tr>
<td>CAN I TRY WINDRIVER BEFORE I BUY?</td>
<td>13</td>
</tr>
<tr>
<td>HOW DO I DEVELOP MY DRIVER WITH WINDRIVER? (OVERVIEW)</td>
<td>15</td>
</tr>
<tr>
<td>WHAT DOES THE WINDRIVER TOOLKIT INCLUDE?</td>
<td>17</td>
</tr>
<tr>
<td>CAN I DISTRIBUTE THE DRIVER CREATED WITH WINDRIVER?</td>
<td>20</td>
</tr>
<tr>
<td>DEVICE DRIVER OVERVIEW</td>
<td>20</td>
</tr>
</tbody>
</table>

INSTALLATION AND SETUP  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM REQUIREMENTS</td>
<td>24</td>
</tr>
<tr>
<td>INSTALLING WINDRIVER</td>
<td>26</td>
</tr>
<tr>
<td>INSTALLING WINDRIVER CE</td>
<td>27</td>
</tr>
<tr>
<td>INSTALLING WINDRIVER FOR LINUX</td>
<td>29</td>
</tr>
<tr>
<td>INSTALLING WINDRIVER ALPHA NT (NT FOR THE ALPHA PROCESSOR)</td>
<td>31</td>
</tr>
<tr>
<td>CHECKING YOUR INSTALLATION</td>
<td>31</td>
</tr>
</tbody>
</table>

THE DRIVERWIZARD  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>WINDRIVER DEVELOPER'S GUIDE</td>
<td></td>
</tr>
<tr>
<td>DRIVERWIZARD - AN OVERVIEW</td>
<td>33</td>
</tr>
<tr>
<td>WIZARD WALKTHROUGH</td>
<td>34</td>
</tr>
<tr>
<td>DRIVERWIZARD NOTES</td>
<td>38</td>
</tr>
<tr>
<td>SHARING A RESOURCE</td>
<td>38</td>
</tr>
<tr>
<td>DISABLING A RESOURCE</td>
<td>38</td>
</tr>
<tr>
<td>DRIVERWIZARD LOGGER</td>
<td>39</td>
</tr>
<tr>
<td>AUTOMATIC CODE GENERATION</td>
<td>39</td>
</tr>
<tr>
<td>CREATING YOUR DRIVER</td>
<td>42</td>
</tr>
<tr>
<td>USING THE DRIVERWIZARD TO BUILD A DEVICE DRIVER</td>
<td>42</td>
</tr>
<tr>
<td>WRITING THE DEVICE DRIVER WITHOUT THE WIZARD</td>
<td>44</td>
</tr>
<tr>
<td>WIN CE - TESTING YOUR DRIVER ON YOUR CE EMULATION UNDER WINDOWS NT.</td>
<td>45</td>
</tr>
<tr>
<td>USING THE HELP FILES</td>
<td>46</td>
</tr>
<tr>
<td>DEBUGGING</td>
<td>47</td>
</tr>
<tr>
<td>USER MODE DEBUGGING</td>
<td>47</td>
</tr>
<tr>
<td>KERNELTRACER</td>
<td>49</td>
</tr>
<tr>
<td>USING KERNELTRACER</td>
<td>49</td>
</tr>
<tr>
<td>WINDRIVER FOR PLX I/O DEVICES FUNCTION REFERENCE</td>
<td>52</td>
</tr>
<tr>
<td>OVERVIEW</td>
<td>52</td>
</tr>
<tr>
<td>WHAT IS THE PCI DIAGNOSTICS PROGRAM?</td>
<td>53</td>
</tr>
<tr>
<td>USING YOUR PCI CHIP-SET DIAGNOSTICS PROGRAM</td>
<td>53</td>
</tr>
<tr>
<td>WINDRIVER PLX CHIP-SET API FUNCTION REFERENCE</td>
<td>59</td>
</tr>
<tr>
<td>P90XX_OPEN()</td>
<td>60</td>
</tr>
<tr>
<td>P90XX_COUNTCARDS()</td>
<td>61</td>
</tr>
<tr>
<td>P90XX_CLOSE()</td>
<td>62</td>
</tr>
<tr>
<td>P90XX_ISADDRSPACEACTIVE()</td>
<td>63</td>
</tr>
<tr>
<td>P90XX_GETREVISION()</td>
<td>64</td>
</tr>
<tr>
<td>P90XX_READREG()</td>
<td>65</td>
</tr>
<tr>
<td>P90XX_WRITEREG()</td>
<td>65</td>
</tr>
<tr>
<td>P90XX_READBYTELOCAL()</td>
<td>66</td>
</tr>
</tbody>
</table>
P90XX_**READ**WordLOCAL () 66
P90XX_**READ**DWordLOCAL () 66
P90XX_**WRITE**ByteLOCAL () 66
P90XX_**WRITE**WordLOCAL () 66
P90XX_**WRITE**DWordLOCAL () 66
P90XX_**READ**BLOCK() 67
P90XX_**WRITE**BLOCK() 67
P9050_**READ**SpaceByte() 68
P9050_**READ**SpaceWord() 68
P9050_**READ**SpaceDWord() 68
P9050_**WRITE**SpaceByte() 68
P9050_**WRITE**SpaceWord() 68
P9050_**WRITE**SpaceDWord() 68
P9050_**READ**SpaceBLOCK() 68
P9050_**WRITE**SpaceBLOCK() 68
P90XX_**INT**ENABLE() 70
P90XX_**INT**DISABLE() 72
P90XX_DMA**OPEN**() 73
P90XX_DMA**START** () 74
P90XX_DMA**CLOSE**() 75
P90XX_DMA**READ**WriteBLOCK() 76
P90XX_DMA**IS**Done() 77
P90XX_Pulse**LOCAL**Reset() 78
P90XX_EEPROM**READ**() 79
P90XX_EEPROM**WRITE**() 79
P90XX_**READ**PCI**REG** () 80
P90XX_**WRITE**PCI**REG**() 80

**IMPROVING PERFORMANCE**  82

**IMPROVING THE PERFORMANCE OF YOUR DEVICE DRIVER - OVERVIEW**  82
**PERFORMANCE IMPROVEMENT CHECKLIST**  83
**IMPROVING THE PERFORMANCE OF YOUR USER MODE DRIVER**  85

**DEVELOPING IN VISUAL BASIC AND DELPHI**  88
TROUBLE-SHOOTING 90

P90XX_CARD_REGISTER() FAILS. 91
COMPUTER HANGS ON INTERRUPT 92

DYNAMICALLY LOADING YOUR DRIVER 94

WINDOWS NT AND 9X 94
DYNAMIC LOADING - BACKGROUND 94
WHY DO YOU NEED A DYNAMICALLY LOADABLE DRIVER? 95
DYNAMICALLY LOADING AND UNLOADING YOUR DRIVER 95
DYNAMICALLY LOADING YOUR KERNEL PLUGIN 96
LINUX 97

DISTRIBUTING YOUR DRIVER 98

GET A VALID LICENSE FOR YOUR WinDriver 98
WINDOWS 9X AND NT: 99
FOR WINDOWS CE: 100
ADD WinDriver TO THE LIST OF DEVICE DRIVERS WINDOWS CE LOADS ON BOOT 100

APPENDIX 102

PC-BASED DEVELOPMENT PLATFORM PARALLEL PORT CABLE
SPECIFICATION (FOR WINDOWS CE): 102
LIMITATIONS ON DEMO VERSIONS: 104
VERSION HISTORY LIST: 105
PURCHASING WinDriver 109
DISTRIBUTING YOUR DRIVER - LEGAL ISSUES 111
WinDriver Overview

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

Introduction to WinDriver

WinDriver is a device driver development toolkit that makes the very difficult task of developing a device driver an easy and quick task for you. The driver you develop using WinDriver will be source code compatible between all supported operating systems (WinDriver currently supports Windows 95, 98, NT 3.51, NT 4.00, 2000, CE and Linux - Solaris and VxWorks coming soon!) It will also be binary compatible between Windows 95, 98, NT and 2000. Bus architecture support includes PCI / PCMCIA / ISA / ISA PnP / EISA and USB. WinDriver provides a complete solution for creating high performance drivers which handle interrupts and I/O at optimal rates.

WinDriver includes enhanced support for the following PLX I/O Devices: PCI 9054, PCI 9080, PCI 9052, PCI 9050, and PCI 9060. This support provides even easier access to your PLX-based PCI card, and includes pre-programmed access to special PLX features. All access to memory ranges, registers and interrupt handling are already pre-programmed for you. Even using the DMA only involves a call to a simple DMA routine. The WinDriver PLX API is implemented using the generic WinDriver API (source code
Documentation on the generic WinDriver API can be found in the WinDriver Manual which is available on your “WinDriver for PLX I/O Devices” CD as a PDF file. In this manual you will see references to the WinDriver API (e.g. WD_Open() function) and to the PLX chip set API (e.g. P9054_Open() function). WinDriver provides APIs for the PLX 9050, 9052, 9054, 9060 and 9080 chip sets. The convention for talking about a family of functions for these chip sets is P90xx. For example, when referring to the "Open" function for these chip sets, P90xx_Open() will be used. When opening a PLX 9054 based card, P9054_Open() will be used.

The last several chapters of this manual explain how to tune your driver code to achieve optimal performance. The “Kernel PlugIn” feature of WinDriver will be explained there. This feature allows the developer to write and debug all of the device driver in the User Mode, and later ‘drop’ performance critical parts of it to the Kernel Mode. This way, your driver development achieves optimal Kernel Mode performance, with User Mode ease of use.

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

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

Good luck with your project!

Background

In the Windows 9x, NT, 2000, and Linux, a programmer cannot access hardware directly from the application level (the “User Mode”) where development work is usually done. Hardware access is allowed only from within the operating system itself (the “Kernel Mode” or “Ring 0”), by software modules called “Device Drivers”. In order to access a custom
hardware device from the application level, a programmer must do the following:

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

2. Learn how to write a device driver.

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

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

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

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

WinDriver Overview

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

**Multi Platform** - The driver created with WinDriver will run on Windows 95/98/NT3.51/NT4.0/2000 CE and Linux ,(NT version available for x86 and Alpha processors), - i.e. write once - run on any of these platforms.
Friendly Wizards - The DriverWizard (included) is a Graphical diagnostics tool that lets you write to, and read from the hardware, before writing a single line of code. With a few clicks of the mouse, the hardware is diagnosed - memory ranges are read, registers are toggled and interrupts are checked. Once the card is operating to your satisfaction, the DriverWizard creates the skeletal driver source code, giving access functions to all of the resources on the hardware.

PLX Chip set APIs - WinDriver provides APIs which dramatically shorten your development time when using one of the PLX PCI chip sets.

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

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

"User Mode ease - Kernel Mode performance!"
To conclude -- using WinDriver, all a developer has to do to create an application that accesses the custom hardware is:

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

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

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

The new hardware access application now runs on all Windows platforms (including CE) and on Linux (just recompile).
WinDriver Feature List

- Easy User Mode driver development.
- Kernel PlugIn for high performance drivers.
- Friendly DriverWizard allows hardware diagnostics without writing a single line of code. Later, the DriverWizard creates most of the driver code for the developer.
- Supports any PCI/ISA/ISA PnP/EISA/PCMCIA/USB chip regardless of manufacturer.
- Enhanced support for the PLX 9050, 9054, 9060 and 9080 PCI bridges, thereby hiding the PCI bridge details from the developer.
- Applications are binary compatible across Windows 9x and Windows NT/2000.
- Applications are source code compatible across Windows 9x, NT, 2000, CE and Linux.
- WinDriver can be used with common development environments including Visual C++, Borland C++, VB4 and Delphi.
- No DDK, ETK or any system-level programming knowledge is required.
- Detailed examples in C, Delphi and Visual Basic are included.
- Supports I/O, DMA, Interrupt handling and Access to memory mapped cards.
- Supports Multiple CPU and Multiple PCI-bus platforms.
- Includes Dynamic Driver Loader.
- Comprehensive documentation and help files.
• Six months Free technical support.
• No run time fees or royalties.

Notes:
In version 4.1 PCMCIA is only supported in the Windows CE version.
WinDriver Architecture

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

When using the PLX specific API, your application calls the PLX API, which uses the WinDriver API to call your PLX-based hardware.

WinDriver's design minimises performance hits on your code, even though it is running in the User Mode. However, some hardware drivers need performance which is not achievable from the User Mode. This is where WinDriver's edge sharpens - after easily creating and debugging your code in the User Mode, you may ‘drop’ the performance critical modules of your code.
(such as a hardware interrupt handler) to the WinD river Kernel PlugIn without changing a single line of it. Now, the WinD river Kernel will call this module from the Kernel Mode, thereby achieving maximal performance. This allows you to program and debug in the User Mode, and still achieve kernel performance where needed. In Windows CE there is no separation between User Mode and Kernel Mode, therefore you may achieve optimal performance directly from the user mode, eliminating the need to use the Kernel PlugIn in this OS.

What Platforms does WinDriver Support?

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

Can I Try WinDriver Before I Buy?

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

Limitations of the different evaluation versions

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

Windows 9x and NT

A message screen will appear on each first use of WinD river saying that this is an un-distributable version of WinD river for evaluation purposes only.

The Evaluation will expire within 30 days of installation.
**Windows CE**
A message screen will appear on each first use of WinDriver saying that this is an un-distributable version of WinDriver for evaluation purposes only.

The WinDriver CE kernel will stop working after 10 minutes and will have to be reloaded.

The WinDriver NT kernel (needed to test your hardware in the CE emulation environment) will stop operating after 30 days.

**Linux**
A message screen will appear on each first use of WinDriver saying that this is an un-distributable version of WinDriver for evaluation purposes only.

The WinDriver Linux kernel will stop working after 10 minutes and will have to be reloaded.

Note that when the WinDriver Linux kernel stops you may not receive a warning message notifying you of this and you may wrongly think that the kernel agent is malfunctioning. If your WinDriver Linux kernel stops responding, just unload it and then reload it again. It will then work for another 10 minutes.

**WinDriver Alpha NT (Windows NT for Alpha processors)**
This version of WinDriver is identical in every way to the WinDriver for Windows NT on the x86 platform.
How Do I Develop My Driver with WinDriver?

(Overview)

On Windows 9x, NT and 2000
Start the DriverWizard (See the ‘DriverWizard’ chapter for details). Diagnose your card, and let DriverWizard generate a skeleton code for you. The code generated by DriverWizard is a diagnostic program, containing functions that read and write to any resource detected or defined (including custom defined registers), and enables and listens to your card interrupts. Modify the code generated by the DriverWizard, to suit your particular application needs.

When using the PLX chip set - use the diagnostic application found in \windriver\plx\90xx\p90xx_diag\. This code is a diagnostic program, containing functions that read and write to all resource on your board, and listens to your card interrupts. Modify this code to suit your particular application needs.

Run and debug your driver in the User Mode.

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

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

Diagnose your hardware via the DriverWizard and then let it generate your driver’s skeleton code. Modify this code using Visual C++ to meet your specific needs.

When using the PLX chip set - use the diagnostic application found in \windriver\plx\90xx\p90xx_diag\. This code is a diagnostic program, containing functions that read and write to all resource on your board, and
listens to your card interrupts. Modify this code to suit your particular
application needs.

Test and debug your code and hardware from the CE emulation running on
the NT machine.

If you cannot plug your hardware in to your NT machine you may still use the
DriverWizard based code or the P90xx_diag sample by compiling them on the
NT machine, and downloading them to your CE machine for testing.

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

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

The following modules are included in your WinDriver toolkit:

WINDRIVER MODULES

- 'WinDriver' - (\windriver\include) - The general-purpose hardware access toolkit.
- 'WinDriver for PLX I/O Devices' API - (\windriver\plx\9050, ~\9054, ~\9060, and ~\9080) - The PLX specific API. Each of these directories includes the following directories:
  - \lib - the special chip set API for the PLX chip set, written using the WinDriver API.
  - \P90xx_DIAG - a sample diagnostics application, which was written using the special library functions available for the these chip sets. This application may be compiled and 

• **DriverWizard (accessible through Start menu\Programs\WinDriver\DriverWizard)** - A graphical debugging tool which collects debugging information on your driver as it runs. In Linux you may use the console version of this file.

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

• **WinDriver electronic manual’ (accessible through ‘Start menu\Programs\windriver’)** - Full WinDriver manual, in pdf (Adobe Acrobat) format.

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

**Utilities:**

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

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

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

**The CE version Also Includes:**

• **\REDIST\...\X86EMU\WINDRVR_CE_EMU.DLL**: The DLL that communicates with the WinDriver kernel for the X86 HPC emulation mode of Windows CE.
• \REDIST\...\X86EMU\WINDRVR_CE_EMU.LIB: The import library for linking with WinDriver applications that are compiled for the X 86 HPC emulation mode of Windows CE.

Samples:

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

• WinDriver samples - (\windriver\samples) - Samples which demonstrate different common drivers.

• WinDriver for PLX I/O Devices' samples - (\p9054_diag ~\p9080_diag etc.) - Source code of the diagnostics applications for the specific chipsets that WinDriver supports.
Can I Distribute the Driver Created with WinDriver?

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

Device Driver Overview

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

**Monolithic drivers:**
These are the "classic" device drivers, which are primarily used to drive custom hardware. A monolithic driver is accessed by one or more user applications, and directly drives a hardware device. The driver communicates with the application through IO control commands - (IOCTLs), and drives the hardware through calling the different DDK functions.
Layered drivers:
Layered drivers are device drivers that are part of a "stack" of device drivers, that together process an IO request. An example of a layered driver is a driver which intercepts calls to the disk, and encrypts / decrypts all data being written / read from the disk. In this example, a driver would be hooked on to the top of the existing driver and would only do the encryption decryption.
Miniport drivers:
There are classes of device drivers in which much of the code has to do with the functionality of the device, and not with the device's inner workings. In these classes of drivers, these code elements will be duplicated.

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

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

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

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

For Layered or Miniport drivers, kernel programming is necessary. To simplify this difficult task, KRFTech provides "KernelD river" - a C++ toolkit which provides classes that encapsulate thousands of lines of kernel code, enabling you to focus on your driver's added-value functionality, instead of your OS internals. KernelD river is available directly from KRFTech (www.krftech.com) or through KRFTech's distribution channels. KernelD river supports the PLX PCI I/O Devices as well.
Installation and Setup

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

System Requirements

**For Windows 95 / 98**
1. An x86 processor
2. Any 32bit development environment supporting C, VB or Delphi.

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

**For Windows CE**
1. Windows NT Workstation 4.0 host development platform
2. Microsoft Developer Studio 97 including:
   - Microsoft Visual C++ V5.0 or higher
   - Windows CE Platform SDK
If you are using a commercial Windows CE handheld Computer like the HP Jornada or the Sharp Mobilon, you will need the following items in addition:

3. Your handheld computer.

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

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

1. Your target platform.

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

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

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

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

For Linux

1. An x86 processor

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

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

Installing WinDriver for Windows 9x / NT / 2000

1. Insert the WinDriver CD to your driver.

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

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

4. Registered users: Choose ‘Install registered version’ when prompted for which version to install.

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

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

   - Minimal - To install only the generic WinDriver toolkit.

   - Custom - To choose which modules of the WinDriver to install. You may choose which APIs will be installed.
6. Registered users: You will now be prompted for an 8-digit password to continue the installation. Type in the password you received when purchasing WinDriver. Take care when entering the password. The installation will fail if the wrong password is written here. Note that the password is case sensitive.

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

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

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

**Installing WinDriver CE**

**IF YOU ARE INSTALLING WINDRIVER CE FOR A HANDHELD COMPUTER:**

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

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

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

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

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

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

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

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

1. Repeat steps 1-2 in above.

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

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

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

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

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

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

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

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

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

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

Installing WinDriver for Linux

Installing WinDriver for Linux on your Linux machine

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

2. Create a directory /usr/bin/windriver

   /usr/bin> mkdir windriver
3. Make windriver your active directory

   /usr/bin> cd windriver

4. Extract the file wd500plx.tgz

   /usr/bin/windriver> tar –xvzf
   /mnt/cdrom/LINUX/wd500plx.tgz

5. Run the script install_windriver.

   /usr/bin/windriver> install_windriver

**The following steps are for Registered Users only**

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

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

7. Extract the file found in wd500reg.zip and enter the password you have received with the WinDriver package.

   /usr/bin/windriver/redist/register> unzip
   wd500reg.zip

8. Replace the evaluation WinDriver kernel (windrvr.o) with the registered version you have extracted in item 6.

   If using Linux Kernel 2.0.x:
   copy 2.0/windrvr.o /lib/modules/misc

   If using Linux Kernel 2.2.x:
   copy 2.2/windrvr.o /lib/modules/misc

9. To activate source code you have developed in the evaluation version simply follow the instructions in \windriver\ redist\ register\ register.txt.
Installing the DriverWizard on your Windows machine

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

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

Installing WinDriver Alpha NT (NT for the Alpha processor)

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

2. Unzip the file “Wd500axp.zip” found on \Alpha.

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

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

Checking Your Installation

On your Windows machine (Including Alpha processors):

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

2. Registered users: Make sure that your WinDriver license is installed (see the ‘Installing WinDriver’ section). If you are an evaluation version user, you do not need to install a license.
3. For PCI cards - Insert your card into the PCI bus, and check that the DriverWizard detects it.

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

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

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

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

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

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

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

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

On your Linux machine:
1. Run the precompiled speaker sample found in \windriver\samples\speaker\LINUX\speaker

2. If the sample program works - you have installed you WinDriver for Linux properly.
The DriverWizard

DriverWizard - An Overview

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

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

It is recommended to start your driver development by letting the DriverWizard generate the driver code for you. If you are developing a driver for a PLX based card, it is recommended to move straight to the "Enhanced Support for specific PCI" chapter, and to start your driver development from
there. You will not need the Wizard, since the specific PLX PCI support includes functions for accessing all of your board specific resources, and for handling all PLX specific functions such as DMA.

The 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 / ISA / PCMCIA bus, and use the DriverWizard to check that the hardware is performing as expected.

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

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

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

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

**Wizard Walkthrough**

Following are the five steps in using the DriverWizard:

1. Insert your card in your hardware bus (PCI / ISA / PCMCIA)

2. Run the Wizard

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

• Choose your PnP card from the list of cards detected by DriverWizard or configure it manually (for non PnP cards like ISA).
3. Diagnose your card

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


- Choose the ‘Generate code’ option from the Build menu.
- Select the WinDriver option on the ‘Choose type of driver’ screen. Selecting the KernelDriver option will generate kernel source code designed for full kernel mode drivers – See the KernelDriver documentation or the KRFTech web site for more details.
• On the following screen, choose your desired development environment for the various operating systems.

• Press the ‘generate code’ button at the bottom of the screen.

5. Compile and run the generated code.

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

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

Sharing a Resource

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

To define a resource as shared:

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

Disabling a Resource

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

Disabling a resource:

1. Select the resource.
2. ‘Right - click’ on the resource name.
3. Choose ‘Disable’ from the menu.
DriverWizard Logger

The DriverWizard Logger is the blank window that opens up along with the card 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 analyse his card’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 card and have ensured that it runs according to your specifications, you are ready to write your driver.

**Step One – Generating your code.**
Choose ‘Generate Code’ from the ‘Build’ menu.

DriverWizard will generate the source code for your driver, and place it under the same directory of the project file (xxx.wdp where xxx is your project name).

In the source code directory you now have a new ‘xxxlib.h’ file which states the interface for the new functions that DriverWizard created for you, and the source of these functions ‘xxxlib.c’. In addition, you will find the sample main() function in the file ‘xxxdia.c’.

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

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

   **xxx_lib.c** – Here you can find the implementation of your hardware
specific API, (found in xxx_lib.h), using the regular WinDriver API.

xxx_lib.h – This is the header file of the diagnostic program. Here you can find all your hardware specific API created by the DriverWizard. You should include this file in your source code to use this API.

2. A general PCI utility library

A diagnostics program, which is a console application with which you can diagnose your card. This application utilises the special library functions which were created for your card by the Wizard. Use this diagnostics program as the skeleton for your device driver.
P90xx_DIAG.c – This is the source code of the diagnostics program the DriverWizard creates.

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

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

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

**Step 2 - Compiling the generated code**

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

For Windows platforms, DriverWizard generates the project files (for MSDEV 4, 5 and 6). After the code generation, the MSDEV will launch automatically. You may immediately compile and run the generated code.

**For Other OSs or IDEs**

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

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

Compile and run the project.
The project contains a working example of the custom functions that DriverWizard created for you. Use this example to create the functionality you want.
Creating Your Driver

This chapter takes you through the WinDriver driver development cycle.

**IMPORTANT NOTE:**
If your card's PCI bridge is PLX-based, then WinDriver's special chip-set APIs will dramatically shorten your development time. If this is the case, read the following overview, and jump straight to the PLX chapter.

Using the DriverWizard to Build a Device Driver

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

2. Use the DriverWizard to generate the skeleton code for your device. (See the 'DriverWizard' Chapter for details)
3. If you are using a PLX chip as your PCI bridge - it is recommended that you use the p9050_diag.exe | p9054_diag.exe | p9060_diag.exe | p9080_diag.exe as a skeleton for your driver code. These executables are applications which access all the registers and memory ranges through the respective bridge. Their full WinD river source code is included. (See the respective chipset chapter for more details on using the diagnostics applications).

4. Use any 32bit compiler (such as MSD EV, Borland C/ C++ and Watcom C/ C++) to build your code.

5. That’s all you need to create your User Mode driver. If you discover that better performance is needed see the “Improving performance” section for details. This section will suggest some performance enhancements you can make in your User Mode driver, or instruct you on how to move parts of your code to the WinD river Kernel PlugIn. This will eliminate any performance problem.
Writing the Device Driver without the Wizard

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

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

2. Add these lines to the source code:

   ```
   #include <windows.h>
   #include <winioctl.h>
   #include "windrvr.h"
   ```

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

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

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

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

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

8. For PCMCIA Cards: call WD_PcmciaScanCards to get a list of the PCMCIA cards installed. Choose your card and call WD_PcmciaGetCardInfo().
Note: WD_PcmciaGetCardInfo() inserts a ITEM_BUS item as the first element of the WD_ITEMS array of the WD_CARD structure that it returns. This item must be present for PCMCIA card configuration to work correctly. If you are filling up the WD_CARD structure yourself without the help of WD_PcmciaGetCardInfo(), then you must set up this item yourself and it must be the first entry in the WD_ITEMS array.

9. For USB cards – see the electronic reference manual or check out WinDriver’s web site for details.

10. Call WD_CardRegister().

11. Now you can use WD_Transfer() to perform IO and memory transfers.

12. If the card uses interrupts call WD_IntEnable(). Now you can wait for interrupts using WD_IntWait().

13. To finish call WD_CardUnregister(), and at the end call WD_Close().

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

WinD river is currently the only tool which enables you to test your driver code with your hardware on your NT machine – under the CE emulation environment. This can dramatically shorten your development time by eliminating the need to work via a serial cable each time you want to see how your driver code operates your hardware.

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

**Using the Help Files**

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

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

User Mode Debugging

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

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

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

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

- \WINCE210\PUBLIC\COMMON\DDK\INC\DBGPRINT.H
- \WINCE210\PUBLIC\COMMON\OAK\DEMOS\DBGSAMPI
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 command sent to the kernel is executed.

Using KernelTracer

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

**KernelTracer – Graphical mode:**

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

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

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

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

**Level** - Choose the level of messages you are interested to see for the resources defined. Error is the lowest level of trace, resulting with minimum
output to the screen. Trace is the highest level of tracing displaying every operation the WinDriver Kernel performs.

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

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

To see activity logged with the KernelTracer simply type “KernelTracer dump”.

---

51
WinDriver for PLX I/O Devices
Function Reference

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

Overview

In addition to the regular WinDriver API, described in former chapters, WinDriver also offers a custom API for the PLX PCI chip-sets.

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

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

2. Locate your specific card diagnostic program. See
   \WIN\PLX\P90xx\p90xx_diag\p90xx_diagc

3. Use this source code as a skeleton for your device driver.
4. Modify the code to suit your application. Use your PCI chip specific function reference to add your own code. More help and details can be found on the Windriver electronic reference manual.

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

What is The PCI Diagnostics program?

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

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

You may also find that the P90xx_DIAG.C is both an example of using your specific API as well as a useful diagnostics utility.

Using Your PCI chip-set Diagnostics program

Introduction

The custom diagnostics program (P90xx_DIAG.EXE) accesses the hardware using Windriver. Therefore Windriver must be installed before being able to
run P90xx_DIAG. If WinDriver is installed correctly, at boot time a message will appear on screen displaying the WinDriver version installed.

Once WinDriver is running, you may run P90xx_DIAG. Click Start | Programs | WinDriver | Samples | Chip_name Diagnostics.

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

Main Menu Options

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

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

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

Your_PCI local registers:
This option is available only after choosing an active card. A list of your PCI registers and their READ value are displayed. In order to WRITE to a register, enter the register number, and then enter the value to write to it.
Access memory ranges on the board:
This option is available only after choosing an active card. Use this option carefully. Accessing memory ranges, accesses the local bus on your card -- If you access an invalid local address, or if you have any problem with your card (such as a problem with the IRDY signal), the CPU may hang.

- To access a local region, first toggle active mode between BYTE/WORD/DWORD, to fit the hardware you are accessing.

- To READ from a local address, choose 'Read from board'. You will be asked for local address to read from.

- To WRITE from a local address, choose 'Write from board'. You will be asked for local address to write to, and the data to write.

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

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

Access EEPROM device (Where available):
This option provides basic read/write access to the serial configuration EEPROM.

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

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

**Pulse Local Reset (where available):**
This option provides a way to reset the local processor, from the host.

- To RESET the local host processor, choose ‘Enter reset duration in milliseconds’. You will be asked for the time in milliseconds. Note: Resolution of delay time is based on PC timer tick, or approximately 55 milliseconds.
Creating your driver without using the PCI diagnostic code as a skeleton.

- Add P90xx_LIB.C to your project or your make file.
- Include “P90xx_lib.h” in your driver source code.

NOTE: In your \ windriver\ PLX\ 90xx\ P90xx_DIAG folder, you will find the source code for P90xx_DIAG.EXE. Double click the ‘mdp’ file (which contains the project environment used to compile this code) in this directory to start your MSDEV with the proper settings for a project. You may use this as a skeleton for your code.

- Call P90xx_Open() at beginning of your code to get a ‘handle’ to your card.

- After locating your card, you may read / write to memory, enable / disable interrupts, access your EEPROM, perform interrupts and more, using the following functions (Some of these functions are not available to all PCI chip-sets or have a different parameter set. Check the header file for your specific chip-set (i.e. 9050, 9054, 9060 or 9080) for exact function definitions):

Reading and Writing to the board:

P90xx_ReadSpaceBlock()
P90xx_WriteSpaceBlock()  
P90xx_ReadSpaceByte()
P90xx_WriteSpaceByte()  
P90xx_ReadSpaceWord()
P90xx_WriteSpaceWord()  
P90xx_ReadSpaceDWord()
P90xx_WriteSpaceDWord()  
P90xx_ReadBlock()
P90xx_WriteBlock()  
P90xx_ReadByte()
P90xx_ReadWord()  
P90xx_ReadDWord()
P90xx_WriteByte()  
P90xx_WriteWord()  
P90xx_WriteDWord()
Interrupt Functions
P90xx_IntIsEnabled()
P90xx_IntEnable()
P90xx_IntDisable()

Register Access
P90xx_ReadReg ()
P90xx_WriteReg ()

PCI configuration Register Access
P90xx_ReadPCIReg()
P90xx_WritePCIReg()

DMA Functions
P90xx_DMAOpen()
P90xx_DMAStart ()
P90xx_DMAIsDone ()
P90xx_DMAClose ()

EEPROM Functions
P90xx_EEPROMRead ()
P90xx_EEPROMWrite ()

Utility Functions
P90xx_IsAddrSpaceActive ()
P90xx_GetRevision ()

• Call P90xx_Close() before end of code.

NOTES:
1. Using the diagnostics application for your specific chip-set will shorten the development process.
2. APIs may slightly vary between PCI chips. See the header files for details.

WinDriver PLX chip-set API Function Reference

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

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

All of the functions outlined in 'Function Reference' are implemented in the \windriver\PLX\90xx\lib\P90xx_lib.c file (full source code is included).
**P90xx_Open()**

Used to open a handle to your card. If several cards with identical PCI chips are installed, the specific card to open may be specified by using `P90xx_CountCards` before using `P90xx_Open`, and calling `open` with a specific card number (See 'prototype' below)

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

**Prototype**

```c
BOOL P90xx_Open (P90xx_HANDLE *phPlx, DWORD dwVendorID, DWORD dwDeviceID, DWORD nCardNum, DWORD dwOptions);
```

**Parameters**

- **phPlx** - Pointer to a PLX handle structure. This pointer will point to a new handle to a PLX card if open succeeds.
- **dwVendorID** - The Vendor ID of the PLX card you wish to locate and open.
- **dwDeviceID** - The Device ID of the PLX card you wish to locate and open.
- **nCardNum** - When several cards exist with the same VendorID and DeviceID, nCardNum is the parameter with which the specific card is chosen. (Use `P90xx_CountCards()` to enumerate your cards).
- **dwOptions** - Use P9054_OPEN_USE_INT to open the card with interrupts, use 0 otherwise.

**Return Value**

TRUE if OK.

**Example**

```c
if (!P9054_Open( &hPlx, 0x10b5, 0x9054, 0, P9054_OPEN_USE_INT ))
{
    printf("Error opening device\n");
}
```
**P90xx_CountCards ()**

Returns the number of cards on the PCI bus that have the given VendorID and DeviceID. This value can then be used when calling P90xx_Open, to choose which board to open. Normally, only one board is in the bus and this function will return 1.

**Prototype**

```c
DWORD P90xx_CountCards (DWORD dwVendorID, DWORD dwDeviceID);
```

**Parameters**

- `dwVendorID` - The VendorID of your card.
- `dwDeviceID` - The DeviceID of your card.

**Return Value**

Returns the number of matching PCI cards found.

**Example**

```c
nCards = P9054_CountCards( 0x10b5, 0x9054 );
```
**P90xx_Close()**

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

**Prototype**

```c
void P9054_Close (P9054_HANDLE hPlx);
```

**Parameters**

- `hPlx` - The handle to the PLX card, which was obtained with `P90xx_Open()`.

**Return Value**

- `none`

**Example**

```c
P9054_Close(hPlx);
```
P90xx_IsAddrSpaceActive()

Checks if the specified address space is enabled. The enabled address spaces are determined by the EEPROM, which at boot time sets the memory ranges requests.

Use this function after calling P90xx_Open() to make sure that the address space(s) that your driver is going to use are enabled.

Prototype

BOOL P9054_IsAddrSpaceActive(P9054_HANDLE hPlx, P9054_ADDR addrSpace);

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

addrSpace - The address space that is checked.

Return Value

TRUE if address space is enabled

Example

if ( !P9054_IsAddrSpaceActive(hPlx, P9054_ADDR_SPACE2) )
{
    printf ("Address space2 is not active!\n");
}
P90xx_GetRevision()  
Returns your PCI chip-set silicon revision.  

Return Value  
Returns the silicon revision.
P90xx_ReadReg ()

P90xx_WriteReg ()

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

Prototype

DWORD P90xx_ReadPCIReg(P90xx_HANDLE hPlx, DWORD dwReg);
void P90xx_WritePCIReg(P90xx_HANDLE hPlx, DWORD dwReg, DWORD dwData);

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

dwReg - The offset of the register from the beginning of Bar0 (where the registers are at). P90xx_lib.h defines the specific registers. You may use offsets for other registers as well.

dwData - The data to write to the register.

Return Value

Data read from register (for P90xx_ReadReg() only).
P90xx_ReadByteLocal()

P90xx_ReadWordLocal()

P90xx_ReadDWordLocal()

P90xx_WriteByteLocal()

P90xx_WriteWordLocal()

P90xx_WriteDWordLocal()

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

Prototype

BYTE P90xx_ReadByteLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr);
void P90xx_WriteByteLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr, BYTE data);

WORD P90xx_ReadWordLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr);
void P90xx_WriteWordLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr, WORD data);

DWORD P90xx_ReadDWordLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr);
void P90xx_WriteDWordLocal (P90xx_HANDLE hPlx, DWORD dwLocalAddr, DWORD data);

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().
dwLocalAddr - The local address to read from or write to.
data - The data to write to the board.

Return Value

For READ - Data read from board.
**P90xx_ReadBlock()**

**P90xx_WriteBlock()**

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

**Prototype**

```c
void P90xx_ReadBlock (P90xx_HANDLE hPlx, DWORD dwOffset,
                       PVOID buf, DWORD dwBytes, P90xx_ADDR addrSpace, P90xx_MODE mode);

void P90xx_WriteBlock (P90xx_HANDLE hPlx, DWORD dwOffset,
                        PVOID buf, DWORD dwBytes, P90xx_ADDR addrSpace, P90xx_MODE mode);
```

**Parameters**

- **hPlx** - The handle to the PLX card, which was obtained with P90xx_Open().
- **dwOffset** - Offset to write to or read from within the specified address space.
- **buf** - Your buffer for the data being read or being written.
- **dwBytes** - The number of bytes to read or write.
- **addrSpace** - The address space to read from or write to.
- **mode** - The resolution of the chunks of data to read or write. Can be one of P90xx_MODE_DESC, P90xx_MODE_DESC_BYTE, P90xx_MODE_DESC_WORD or P90xx_MODE_DESC_DWORD.

**Return Value**

none
P9050_ReadSpaceByte()
P9050_ReadSpaceWord()
P9050_ReadSpaceDWord()
P9050_WriteSpaceByte()
P9050_WriteSpaceWord()
P9050_WriteSpaceDWord()
P9050_ReadSpaceBlock()
P9050_WriteSpaceBlock()

Available ONLY for the PLX 9050 chipset. These functions enable you to read from or write to a specific offset on a memory bar, without specifying a local address.

Prototype

BYTE P9050_ReadSpaceByte (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);
void P9050_WriteSpaceByte (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, BYTE data);
WORD P9050_ReadSpaceWord (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);
void P9050_WriteSpaceWord (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, WORD data);
DWORD P9050_ReadSpaceDWord (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset);
void P9050_WriteSpaceDWord (P9050_HANDLE hPlx, P9050_ADDR addrSpace, DWORD dwOffset, DWORD data);
void P9050_ReadSpaceBlock (P9050_HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);
void P9050_WriteSpaceBlock (P9050_HANDLE hPlx, DWORD dwOffset, PVOID buf, DWORD dwBytes, P9050_ADDR addrSpace, P9050_MODE mode);
Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

addrSpace - The address space to read from or write to.

dwOffset - The offset within the address space from where the reading or writing will start.

data - The data to write.

Return Value

For READ - Data read from board.
**P90xx_IntIsEnabled()**

Checks whether interrupts are enabled or not.

**Prototype**

```c
BOOL P90xx_IntIsEnabled (P90xx_HANDLE hPlx);
```

**Parameters**

`hPlx` - The handle to the PLX card, which was obtained with `P90xx_Open()`.

**Return Value**

`TRUE` if interrupts are already enabled (e.g. if `P90xx_IntEnable()` was called).
**P90xx<IntEnable()**

Enable interrupt processing.

**IMPORTANT NOTE:** All PCI chip-sets use level sensitive interrupts, therefore you must edit the implementation of this function (found in your `windriver\PLX\90xx\lib\P90xx_lib.c`) to fit your specific hardware. The comments in the function will instruct you where your changes must be inserted. See more about this in the section regarding PCI interrupts implementation.

**Prototype**

```c
BOOL P90xx_IntEnable (P90xx_HANDLE hPlx, P90xx_INT_HANDLER funcIntHandler);
```

**Parameters**

- `hPlx` - The handle to the PLX card, which was obtained with `P90xx_Open()`.
- `funcIntHandler` - Pointer to your function which will be called once an interrupt occurs.

**Return Value**

- `TRUE` if successful.
P90xx_IntDisable()

Disable interrupt processing.

Prototype

void P90xx_IntDisable (P90xx_HANDLE hPlx);

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

Return Value

none
**P90xx_DMAOpen()**

Open a DMA to/from PLX card. Returns a handle to a DMA operation, which you provide to P90xx_DMAStart() and P90xx_DMAReadWriteBlock().

**Prototype**

```c
P90xx_DMA_HANDLE P90xx_DMAOpen (P90xx_HANDLE hPlx,
    DWORD dwLocalAddr, PVOID buf, DWORD dwBytes,
    BOOL fIsRead, P90xx_MODE mode,
    P90xx_DMA_CHANNEL dmaChannel);
```

**Parameters**

- `hPlx` - The handle to the PLX card, which was obtained with P90xx_Open().
- `dwLocalAddr` - Local address on card to write to/read from.
- `buf` - The buffer to transfer.
- `dwBytes` - Number of bytes to transfer (must be a multiple of 4).
- `fIsRead` - TRUE: read from card to buffer. FALSE: write from buffer to card.
- `mode` - Local bus width.
- `channel` - DMA channel to use.

**Return Value**

Handle to the DMA operation.
**P90xx_DMAStart ()**

Starts a DMA to / from your PLX based card.

**Prototype**

```c
void P90xx_DMAStart (P90xx_HANDLE hPlx, P90xx_DMA_HANDLE hDma,
                      BOOL fBlocking);
```

**Parameters**

- **hPlx** - The handle to the PLX card, which was obtained with P90xx_Open().
- **hDma** - Handle to the DMA operation which was obtained in the DMA_Open() routine.
- **fBlocking** - 1 = Wait for the DMA to finish before continuint. 0 = don't wait.

**Return Value**

none
P90xx_DMAClose()

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

Prototype

void P90xx_DMAClose (P90xx_HANDLE hPlx, P90xx_DMA_HANDLE hDma);

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

hDma - Handle to the DMA operation which was obtained in the DMA_Open() routine.

Return Value

none
**P90xx_DMAReadWriteBlock()**

Initiates and starts DMA to/from card. The DMAReadWriteBlock() replaces the open / start / close DMA process.

**Prototype**

```c
BOOL P90xx_DMAReadWriteBlock (P90xx_HANDLE hPlx,
                              DWORD dwLocalAddr, PVOID buf,
                              DWORD dwBytes, BOOL fIsRead,
                              P90xx_MODE mode,
                              P90xx_DMA_CHANNEL dmaChannel);
```

**Parameters**

- `hPlx` - The handle to the PLX card, which was obtained with `P90xx_Open()`.
- `dwLocalAddr` - Local address on card to write to / read from.
- `buf` - The buffer to transfer.
- `dwBytes` - Number of bytes to transfer (must be a multiple of 4).
- `fIsRead` - TRUE: read from card to buffer. FALSE: write from buffer to card.
- `mode` - Local bus width.
- `channel` - DMA channel to use.

**Return Value**

Returns TRUE if DMA transfer succeeds.
**P90xx_DMAIsDone()**

Used to test if DMA is done.

**Prototype**

```c
BOOL P90xx_DMAIsDone (P90xx_HANDLE hPlx, P90xx_DMA_HANDLE hDma);
```

**Parameters**

- `hPlx` - The handle to the PLX card, which was obtained with `P90xx_Open()`.

**Return Value**

Returns TRUE if DMA transfer is completed.
P90xx_PulseLocalReset()

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

Prototype

Parameters

hPlx - The handle to the PLX card, which was obtained with P90xx_Open().

Return Value
**P90xx_EEPROMRead()**

**P90xx_EEPROMWrite()**

Reads or writes one to the EEPROM. – Syntax and functionality may vary between different chip-sets. See the header files for details.

**Prototype**

```c
BOOL P90xx_EEPROMReadWord (P90xx_HANDLE hPlx, DWORD dwOffset, PWORD pwData);

BOOL P90xx_EEPROMWriteWord (P90xx_HANDLE hPlx, DWORD dwOffset, WORD wData);
```

**Parameters**

- `hPlx`: The handle to the PLX card, which was obtained with `P90xx_Open()`.
- `dwOffset`: Offset of read or write.
- `wData`: Buffer to read to, or data to write to card.

**Return Value**

Returns TRUE if EEPROM read / write succeeds.
P90xx_ReadPCIReg ()

P90xx_WritePCIReg()

Read from or write to the PCI configuration registers.

**Prototype**

```
DWORD P9054_ReadPCIReg(P9054_HANDLE hPlx, DWORD dwReg);
void P9054_WritePCIReg(P9054_HANDLE hPlx, DWORD dwReg,
                      DWORD dwData);
```

**Parameters**

- **hPlx** - The handle to the PLX card, which was obtained with P90xx_Open().
- **dwReg** - Register to access (offset).
- **dwData** - Data to write.

**Return Value**

Data read from configuration register.
Improving performance

This chapter will help you achieve optimal performance with your device driver, either by fine tuning your implementation, or by moving parts of your code into Windriver's "Kernel PlugIn". More in-depth details on the Kernel PlugIn can be found in the Windriver manual and on-line help.

Improving the performance of your device driver - overview

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

1. Improve the performance of your User Mode driver by fine tuning your use of the Windriver functions, and by using the right Windriver functions for the right tasks (See later in this chapter for details).

2. Move the performance critical parts of your code in to the Windriver's "Kernel PlugIn". The Kernel PlugIn will run your user mode code in the Kernel Mode, thereby achieving optimal performance (See the Windriver
generic manual or on-line help for more information on how to use the Kernel PlugIn).

Note that the Kernel PlugIn is not implemented under Windows CE since in this OS there is no separation between kernel mode and user mode, thus optimal performance can be achieved without using the Kernel PlugIn.

**Performance improvement checklist**

Use the checklist below to determine how the performance should be improved in your driver:

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

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

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

**If you do have performance problems:**

Identify which part of the code the performance problem is at. Classify and solve the problem according to the table below:
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| #1 ISA Card - Accessing an IO mapped range on the card. | • Try to convert multiple calls to WD_Transfer() to one call to WD_MultiTransfer() (See the 'Improving performance - Accessing IO mapped regions' section later in this chapter).  
• If this does not solve the problem, handle the IO at Kernel Mode, by writing a kernel PlugIn. (See the ‘Kernel PlugIn’ chapter for details) |
| #2 PCI Card - Accessing an IO mapped range on the card. | • First, try to change the card from IO mapped to memory mapped by changing bit 0 of the address space PCI configuration register to 0, and then try the solutions for problem #3. You will probably need to re-program the EEPROM to initialise BAR0/1/2/3/4/5 registers with different values.  
• If this is not possible, try the solutions suggested for problem #1.  
• If this does not solve the problem, handle the IO at Kernel Mode, by writing a kernel PlugIn. (See the ‘Kernel PlugIn’ chapter for details) |
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| #3 Accessing a memory mapped range on the card. | • Try to access memory without using WD_Transfer(), by using direct access to memory mapped regions (See the ‘Improving Performance - using direct access to memory mapped regions’ section later in this chapter).
  
• If this does not solve the problem, then there is a hardware design problem. You will not be able to increase performance by using any software design method, or by writing a Kernel PlugIn, or even by writing a full kernel driver. |
| #4 Interrupt latency. (Missing interrupts, Receiving interrupts too late) | You need to handle the interrupts at Kernel Mode, by writing a kernel PlugIn. (See the ‘Kernel PlugIn’ chapter for details) |

**Improving the performance of your User Mode driver**

As a general rule, transfers to memory mapped regions are faster than transfers to IO mapped regions. The reason is that WinDriver enables the user to directly access the memory mapped regions, without calling the WD_Transfer() function.

**Using Direct access to memory mapped regions**

---

85
After registering a memory mapped region, via WD_CardRegister(), two results are returned: dwTransAddr and dwUserDirectAddr. dwTransAddr should be used as a base address when calling WD_Transfer() to read or write to the memory region. A more efficient way to perform memory transfers would be to use dwUserDirectAddr directly as a pointer, and access with it the memory mapped range. This method enables you to read/write data to your memory-mapped region without any function calls overhead (i.e. Zero performance degradation).

**Accessing IO mapped regions**

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

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

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

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

A quick way to save 50% of the function call overhead would be to call WD_Transfer() with a WP_SBYTE (Write Port String Byte), with two bytes at a time. First call would transfer Byte0 and Byte1 to ports 0x300 and 0x301, Second call would transfer Byte2 and Byte3 to ports 0x300 and 0x301 etc. This way, WD_Transfer() will only be called 500,000 times to transfer the block.
The third method would be by preparing an array of 1000 WD_TRANSFER commands. Each command in the array will have a WP_SBYTE command that transfers two bytes at a time. Then you call WD_MultiTransfer() with a pointer to the array of WD_TRANSFER commands. In one call to WD_MultiTransfer() - 2000 bytes of data will be transferred. To transfer the 1MB of data you will need only 500 calls to WD_Transfer(). This is 0.5% of the original calls to WD_Transfer(). The trade off in this case is the memory that is used to set-up the 1000 WD_TRANSFER commands.
Developing in Visual Basic and Delphi

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

**Using DriverWizard**
DriverWizard can be used to diagnose your hardware and verify it is working properly before you start writing code. DriverWizard automatic code generation function currently generates C code only. Automatic generation of Visual Basic and Delphi code will be supported in later versions of WinDriver.

**Samples**
Samples for drivers written using the WinDriver API in Delphi or Visual Basic can be found in:

1. \windriver\delphi\samples
2. \windriver\vb\samples

Use these samples as a starting point for your own driver.
Kernel PlugIn
Delphi and Visual Basic cannot create kernel-mode device drivers, therefore the Kernel PlugIn feature is not available for Delphi and Visual Basic.

Creating your Driver
Development method in Delphi and Visual Basic is the same as for developing under C except for the automatic code generation feature of DriverWizard.

Your work process should be as follows:

- Use DriverWizard to easily diagnose your hardware and verify it is working properly.
- Write your driver code in user mode using the WinDriver API. See details and explanations in the “Writing the device driver without the wizard” section in chapter 4.
- You may find it useful to use the WinDriver samples to get to know the WinDriver APIs and as a skeleton for your driver code.
Trouble-shooting

To determine and verify the cause of your driver problems – Open the KernelTracer (described in chapter 5) and set your desired trace level. This will help narrow down your debugging process and lead you in the right direction.

**P90xx_Open() fails.**

The following may cause P90xx_Open() to fail:

1. **Cause:** WinDriver's kernel is not loaded.

   **Action:** Run 'WDREG.EXE install' (in the \windriver\util directory). This will let Windows know how to add WinDriver to the list of device drivers loaded on boot. Also, copy WINDRVR.SYS (for WinNT) or WINDRVR.VXD (for Win95) to the device drivers directory. A detailed explanation may be found in chapter 7 - 'Distributing your driver'.

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

   **Action:** In this case, the WinDriver will inform you your evaluation license is over, in a message box. Please contact sales@krftech.com to purchase WinDriver.
3. **Cause (for PnP cards only):** The VendorID / DeviceID requested in P90xx_Open() do not match that of the board. (In licensed versions).

   **Action:** Run Your_Card_Name_DIAG.EXE, (Or the P90xx_Diag.exe), and choose scan-pci bus to check the correct VendorID / DeviceID of your hardware.

4. **Cause:** The device is not installed or configured correctly.

   **Action:** Run Your_Card_Name_DIAG.EXE and choose PCI scan. Check that your device returns all the resources needed.

5. **Cause:** Your device is in use by another application.

   **Action:** Close all other applications that might be using your device.

**P90xx_CardRegister() fails.**

P90xx_CardRegister fails if one of the resources defined in the card cannot be locked.

First, check out what resource (out of all the card’s resources) cannot be locked:

Activate the KernelTracer and set the trace mode to trace.

This will output all warning and error debug messages. Now, run your application and you will get a printout of the resource that failed.

After finding out the resource that cannot be locked, check out the following:

Is the resource in use by another application? In order for several resource lock requests to the same IO, Memory or interrupt to succeed, both applications must enable sharing of the resource. This is done by setting fNonSharable = FALSE for every item that can be shared.
Computer hangs on interrupt

This can occur with level-sensitive interrupt handlers. PCI cards interrupts are usually level sensitive.

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

Acknowledging a level sensitive interrupt is hardware specific. Acknowledging an interrupt means lowering the interrupt level generated by the card. Normally, writing to a register on the PCI card can terminate the interrupt, and lower the interrupt level.

When calling P90xx_IntEnable() it is possible to give the WinDriver kernel interrupt handler a list of transfer commands (IO and memory read/write commands) to perform upon interrupt, at the kernel level.

These commands can be used to write to the needed register to lower the interrupt level, thereby ‘re-setting’ the interrupt.

Before calling P90xx_IntEnable(), prepare two transfer command structures (to read the interrupt status and then write the status to lower the level).

```c
WD_TRANSFER trans[1];
BZERO (trans);
trans[0].cmdTrans = WP_DWORD; // Write Port Dword
// address of IO port to write to
trans[0].dwPort = dwAddr;
// the data to write to the IO port
trans[0].Data.Dword = 0;
Intrp.dwCmds = 1;
Intrp.Cmd = trans;
Intrp.dwOptions = INTERRUPT_LEVEL_SENSITIVE;
P90xx_IntEnable(hWD, &Intrp);
```

This will tell WinDriver's kernel to Write to the register at dwAddr a value of '0', upon an interrupt.
The user-mode interrupt handler (The thread waiting on WD_IntWait() - this is your code).

Here you only do your normal stuff to handle the interrupt. You do not need to clear the interrupt level since this was already done by the kernel of WinDriver, with the transfer command you gave WD_IntEnable().
Dynamically loading your driver

Windows NT and 9x

Dynamic loading - background

When adding a new driver to the Windows operating system, you must re-boot the system, for the Windows to load your new driver into the system.

Dynamic loading enables you to install a new driver to your operating system, without needing to re-boot.

WinDriver is a dynamically loadable driver, and provides you with the utility needed to dynamically load the driver you create.

You may dynamically load your driver whether you have created a User Mode or a Kernel Mode driver.
Why do you need a dynamically loadable driver?

A dynamically loadable driver enables your customers to start your application immediately after installing it, without the need to re-boot.

Dynamically loading and unloading your driver

The utility you use to dynamically load and unload your driver is called WDREG.EXE, and may be found in the \windriver\util\WDREG.EXE.

USAGE: WDREG [-name] [-file] [ CREATE ] [ START ] [ STOP ] [ DELETE ] [ INSTALL ] [ REMOVE ]

WDREG.EXE has 4 basic operations:

1. CREATE - Instructs the Windows to load your driver next time it boots, by adding your driver to registry.

2. START - Dynamically loads your driver into memory for use. On Windows NT, you must first ‘CREATE’ your driver before ‘START’ing it.

3. STOP - Dynamically unloads your driver from memory.

4. DELETE - Removes your driver from the registry, so that it does not load on next boot.

For example, to reload WinDriver use:

WDREG STOP START

WDREG.EXE has 2 ‘shortcut’ operations for your convenience:
1. INSTALL - Creates and starts your driver (same as using WD REG CREATE START).

2. REMOVE - Unloads your driver from memory, and removes it from registry so that it does not load on next boot (same as using WD REG STOP DELETE).

You may dynamically load your driver via command line or from within your application as follows:

1. Dynamically loading your driver via command line:

   • From the command line, type WD REG INSTALL. This loads the driver into memory, and instructs Windows to load your driver on the next boot.

2. Dynamically loading your driver in your installation application:

   • Add the WD REG source code to your installation application.

   • The full source code for WD REG may be found at \windriver\samples\wdreg\.

**Dynamically loading your Kernel Plugin**

If you have used WinDriver to develop a Kernel Plugin, you must dynamically load your Kernel Plugin as well as the WinDriver.

To Dynamically load / unload your Kernel Plugin driver ([Your driver name].VXD / [Your driver name].SYS):

Use the WD REG command as described above, with the addition of the “-name” flag, after which you must add the name of your Kernel Plugin driver.
For example, to load your Kernel PlugIn driver called KPTest.VXD or KPTest.SYS, use:

WDREG -name KPTest install

(You should not add the .VXD or .SYS extension to your driver name).

WDREG allows you to install your driver in the registry under a different name than the physical file name.

** USAGE: ** WDREG -name [Your new driver name] -file [Your original driver name] install

For example, typing the following:

WDREG -name “Kernel PlugIn Test” -file KPTest install

Installs the KPTest.VXD or KPTest.SYS driver under a different name.

** Linux **

To dynamically load WinDriver on Linux, execute:

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

To dynamically unload WinDriver, execute:

> /sbin/rmmod windrvr
Distributing your driver

Read this chapter in the final stages of your development. This chapter guides you in creating the distributable package from your driver alongside the WinDriver.

Get a valid license for your WinDriver

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

Alternatively, you may order WinDriver on-line. See http://www.krftech.com for more details.
Windows 9x and NT:

**Copy VxD or SYS files to target computer**

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

- For Windows NT: Copy WINDRV.R.SYS file to C:\WINNT\SYSTEM32\DRIVERS
- For Windows 95: Copy WINDRV.R.VXD file to C:\WIN95\SYSTEM\VMM32

If you have created a WinDriver Kernel PlugIn as well, copy your Kernel PlugIn driver ([Your driver name].VXD or [Your driver name].SYS) to the relevant directory.

**Add WinDriver to the list of Device Drivers**

**Windows loads on boot**

This is done by calling 'WDREG.EXE install'. You can add the WDREG source code (found in \windriver\samples\wdreg\wdreg.cpp) to your own installation code, in order to install WinDriver.

If you have created a WinDriver Kernel PlugIn as well, call "WDREG.EXE -name [Your driver name] install". You can add the 'WDREG' source code (found in \windriver\samples\wdreg\wdreg.cpp) to your own installation code, in order to install WinDriver.

Please see the chapter on 'Dynamically loading your driver' for more details on WDREG.EXE
For Windows CE:

**Copy WinDriver Kernel DLL file to target computer**

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

For Windows CE handheld computer installations:

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

For Windows CE PC:

- Copy WINDRVR.DLL %FLATRELEASEDIR% and use MAKEIMG.EXE to build a new Windows CE kernel NK.BIN. You should modify PLATFORM.REG and PLATFORM.BIB appropriately before doing this by appending the contents of the supplied files PROJECT_WD.REG and PROJECT_WD.BIB respectively. This process is similar to the process of installing WinDriver CE Beta on a CE PC/ETK installation as described in Chapter 2 – INSTALLATION AND SETUP.

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

For Windows CE handheld computer installations, please modify the registry according to the entries documented in the file PROJECT_WD.REG. This can be done using the Windows CE Pocket Registry Editor on the handheld CE computer or by using the Remote CE Registry Editor Tool supplied with the Windows CE Platform SDK. You will need to have Windows CE Services installed on your Windows NT Host System to use the Remote CE Registry Editor Tool.
For Windows CE PC/ETK, the required registry entries are made by appending the contents of the file PROJECT_WD.REG to the Windows CE ETK configuration file PROJECT.REG before building the Windows CE image using MAKEIMG.EXE. If you wish to make the WinDriver kernel file a permanent part of the Windows CE kernel NK.BIN, you should append the contents of the file PROJECT_WD.BIB to the Windows CE ETK configuration file PROJECT.BIB as well.
Appendix

PC-Based Development Platform Parallel Port Cable Specification (For Windows CE):

To use the parallel port shell utility (Ppsh) to transfer a Windows CE image from your development workstation to a PC-based hardware development platform, a custom parallel cable is required. This cable requires a DB-25 male connector at both ends, with pins mapped as follows:

1   10
2     Same
3     Same
4     Same
5     Same
6     Same
7       Same
8       Same
9       Same
10      1
11      14
12      16
13      17
14      11
15      Not Connected On Either End
16      12
17      13
18      Same
19      Same
20      Same
21      Same
22      Same
23      Same
24      Same
25      Same
To order this cable, contact Redmond Cable:

Redmond Cable
15331 NE 90th Street
Redmond, WA 98052

Telephone: (425) 882-2009
Fax: (425) 883-1430

Part Number: 64355913

**Limitations on demo versions:**

**Windows 9x and NT:**
- A DEMO MESSAGE will appear at every first use of WinDriver in each session.
- WinDriver will function for only 30 days after the original installation.

**Windows CE:**
- A DEMO MESSAGE will appear at every first use of WinDriver in each session.
- The WinDriver CE Kernel (windrvr.dll) will operate for no more than 10 minutes at a time.
- WinDriver CE emulation on Windows NT will stop working after 30 days.
Linux:

- The Linux Kernel will work for no more than 10 minutes at a time.

**Version history list:**

**NEW IN V2.02:**

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

**NEW IN V2.10:**

- For memory mapped cards, changed item dwUserAddr to dwTransAddr.
- Use dwTransAddr when calling WD_Transfer(). added dwUserDirectAddr for direct memory transfers without calling WD_Transfer(). dwUserDirectAddr NOT YET IMPLEMENTED.

**NEW IN V2.11:**

- For PCI cards: Structure used for calls to WD_PciScanCards() was changed. Use pciScan.searchId.dwVendorId and pciScan.searchId.VendorId and the same for dwDeviceId.
NEW IN V2.12:

- For memory mapped cards: you can now directly access the memory region, without calling WD_Transfer(). The pointer to the memory region is returned in dwUserDirectAddr returned by WD_CardRegister().

- DMA transfers: DMA contiguous buffer allocation by WinDriver is available by setting dwOptions = DMA_KERNEL_BUFFER_ALLOC, when calling WD_DMAUnlock(). The linear address of the buffer allocated will be returned in pUserAddr, and the physical address in Page[0]. The buffer is available until calling WD_DMAUnlock().

NEW IN V3.0:

- Added DriverWizard to package. DriverWizard enables the programmer to 'talk' and 'listen' to his card via a windows user-interface. The DriverWizard then creates the source code for the driver.

- DMA option DMA_LARGE_BUFFER added for locking regions larger than 1MB.

- Removed limitation of 20 concurrent DMA buffers in use.

NEW IN V3.01

- Support for Win98 and Windows 2000

NEW IN V3.02

- Minor improvements in DriverWizard
• Supports Windows NT checked build

NEW IN V3.03

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

NEW IN V4.0:

• WinDriver Kernel PlugIn - allows running parts of the driver code from the Kernel Mode.
• Sleep function - For accessing slow hardware.
• ISA Plug and Play support.
• Debugging monitor - Allows tracking of errors, warnings and trace messages from the WinDriver's kernel module.
• Dynamic driver loader - WinDriver enables the driver created to be loaded and unloaded without rebooting the machine.
• Enhanced source code generation for interrupts - DriverWizard creates full interrupt source code.
• PLX 9050 library enhancements - EEPROM read/write support functions and Enhanced interrupt handling.

NEW IN VERSION 4.1

• New support for Linux, Windows CE and Alpha NT.
• Support for ISA PnP cards.
• Support for PCMCIA cards in Windows CE.
- Graphical KernelTracer introduced.

- Robust support for Delphi and VB (Visual Basic). More Delphi and VB samples.

- New support for the PLX 9054 and 9080 chipsets. Support includes EEPROM access and bus master DMA implementation.

- V4.1 Includes The Enhanced WinDriver Wizard:
  - Automatic Vendor and Device detection.
  - Automatic handling and code generation for Level sensitive interrupts.
  - Wizard allows multiple concurrent register and memory dialogs.
  - Improved GUI.

NEW IN VERSION 5.0

- WinDriver for PLX I/O Devices - Initial PLX version.
Purchasing WinDriver

Choose the WinDriver product that suits your needs:

- Choose ‘WinDriver’ for NT/2000 or 9x support.
- Choose ‘WinDriver Bundle’ for Windows NT and 9x support (no re-writing or re-compiling needed).
- Choose ‘WinDriver CE’ for Windows CE support.
- Choose ‘WinDriver Linux’ for Linux support.
- Choose ‘WinDriver Alpha NT’ to run your driver on the Alpha NT platform.
- Choose ‘WinDriver ToolBox’ to receive all the above operating systems support in one package. The driver you develop will run under all supported environments.

Fill in the order form found in ‘Start/WinDriver/Order Form’ on your Windows start menu, and send it back to KRFTech via email/fax/mail (see details below).

Your licensed WinDriver will be sent back via email/fax immediately.

E-MAIL:

Support: support@krftech.com
Sales: sales@krftech.com
PHONE / FAX:

Phone:

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

Fax:

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

WEB:

http://www.krftech.com/

ADDRESS:

KRFTech
Moshav Hadar Am, 42935
ISRAEL
Distributing your driver - legal issues

WinDriver is licensed per-seat. The WinDriver license allows for one developer to develop an unlimited number of device drivers, and to freely distribute the created driver without royalties.

You may not distribute the windrvr.h file, or any source file that describes the WinDriver's functions. Please see the \windriver\docs\license.txt file for the full WinDriver license agreement.