

WinDriver™ PCI/PCI Express/PCMCIA Quick Start Guide

A 5-Minute introduction to writing device drivers

Who should use WinDriver?

1. Hardware developers – Use Driver Wizard to quickly test your new hardware.
2. Software developers – Use Driver Wizard to generate the device driver code to drive your hardware. Use the WinDriver tools to test and debug your driver.

Which operating systems does WinDriver support?

1. Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks.
Check the Jungo web site for updates on new operating systems support.
2. WinDriver-based drivers are portable between all supported operating systems without any code modifications.

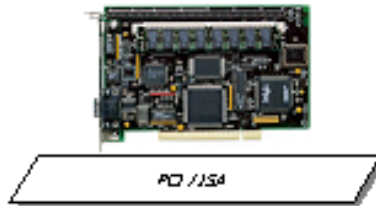
Where can I get more in-depth information?

1. You can download a free, full-featured, 30 days evaluation of WinDriver, including documentation, from our web site: <http://www.jungo.com/download.html>
2. For white papers, user manuals, and other documentation, visit Jungo's on-line documentation page: <http://www.jungo.com/support/manuals.html>

7 steps to building your driver:

1. Set Up

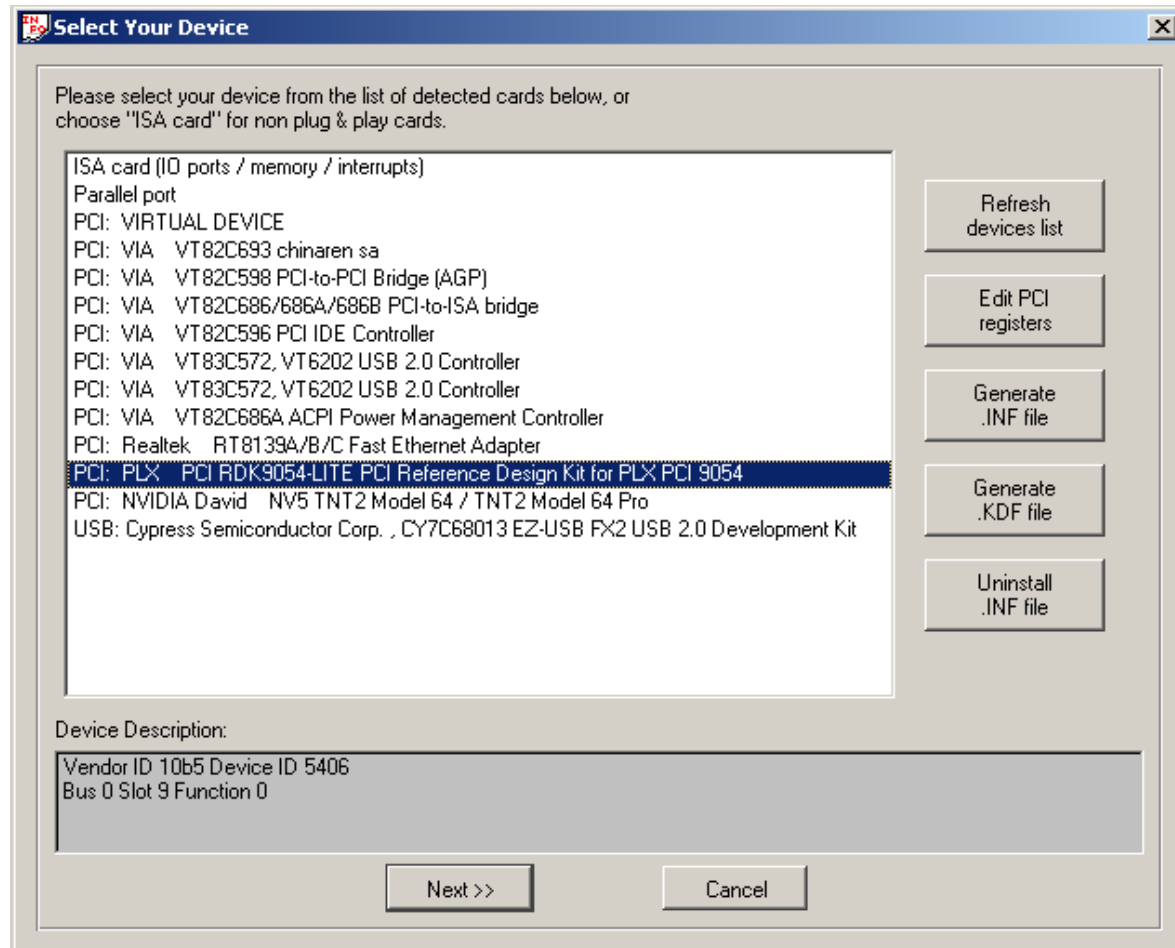
- (1) Plug your device to the PC



- (2) Install WinDriver

2. Select your device

- (1) Start DriverWizard by choosing: **WinDriver | DriverWizard** from Windows **Start** Menu (on Windows), or by running **/WinDriver/wizard/wdwizard**.
- (2) In the dialogue box that appears, choose **New device driver project**.
- (3) DriverWizard will show all Plug-and-Play cards plugged in your machine.
- (4) For Plug-and-Play devices: select your device from the list of devices.
For non-Plug-and-Play (ISA) devices: select the **ISA card** option to define your device's resources.
To generate code for a non-attached PCI device: select the **PCI: VIRTUAL DEVICE** option.



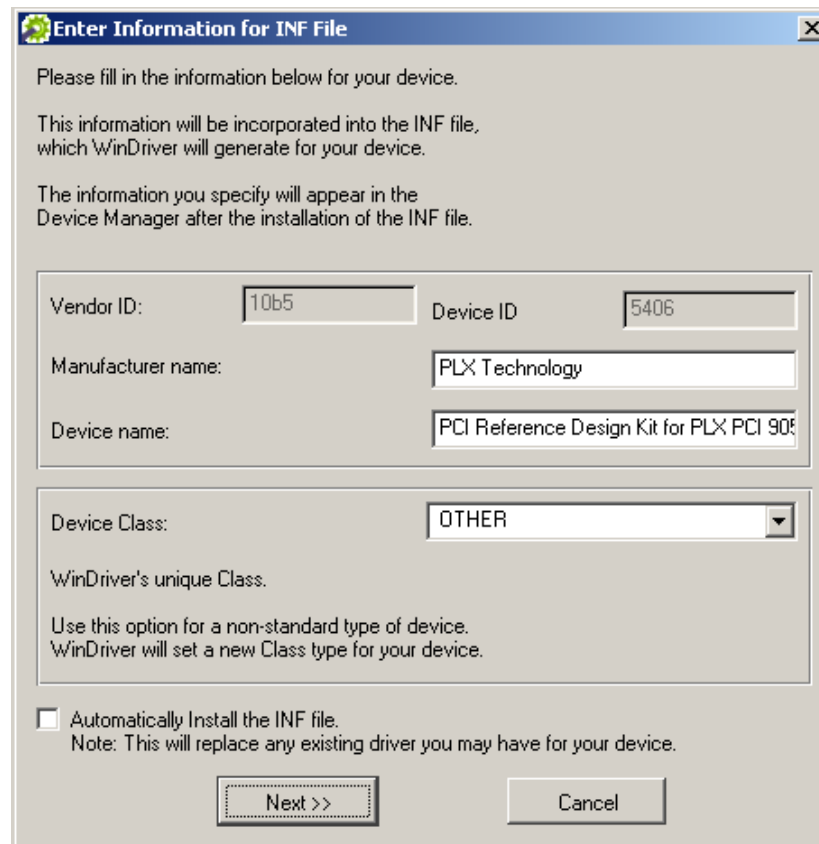
3. Generate an INF File for Plug-and-Play Devices (Windows 98/Me/2000/XP/Server 2003)

When developing a driver for a Plug-and-Play device (PCI/PCMCIA/CardBus) on Plug-and-Play Windows operating systems (**Windows 98/Me/2000/XP/Server 2003**), in order to correctly detect the device's resources and communicate with the device using WinDriver, you need to create an INF file that registers your device to work with WinDriver.

DriverWizard automates the INF generation and installation process for you.

To generate an INF file with DriverWizard:

- (1) Click the **Generate .INF file** button from the wizard's **Select Your Device** dialogue.
- (2) DriverWizard will display information detected for your device – Vendor ID, Device ID, Device Class, manufacturer name and device name – and allow you to modify the manufacturer and device names and the device class information.



Enter Information for INF File

Please fill in the information below for your device.

This information will be incorporated into the INF file, which WinDriver will generate for your device.

The information you specify will appear in the Device Manager after the installation of the INF file.

Vendor ID: 10b5 Device ID: 5406

Manufacturer name: PLX Technology

Device name: PCI Reference Design Kit for PLX PCI 904

Device Class: OTHER

WinDriver's unique Class.

Use this option for a non-standard type of device. WinDriver will set a new Class type for your device.

☐ Automatically Install the INF file.
Note: This will replace any existing driver you may have for your device.

Next >> Cancel

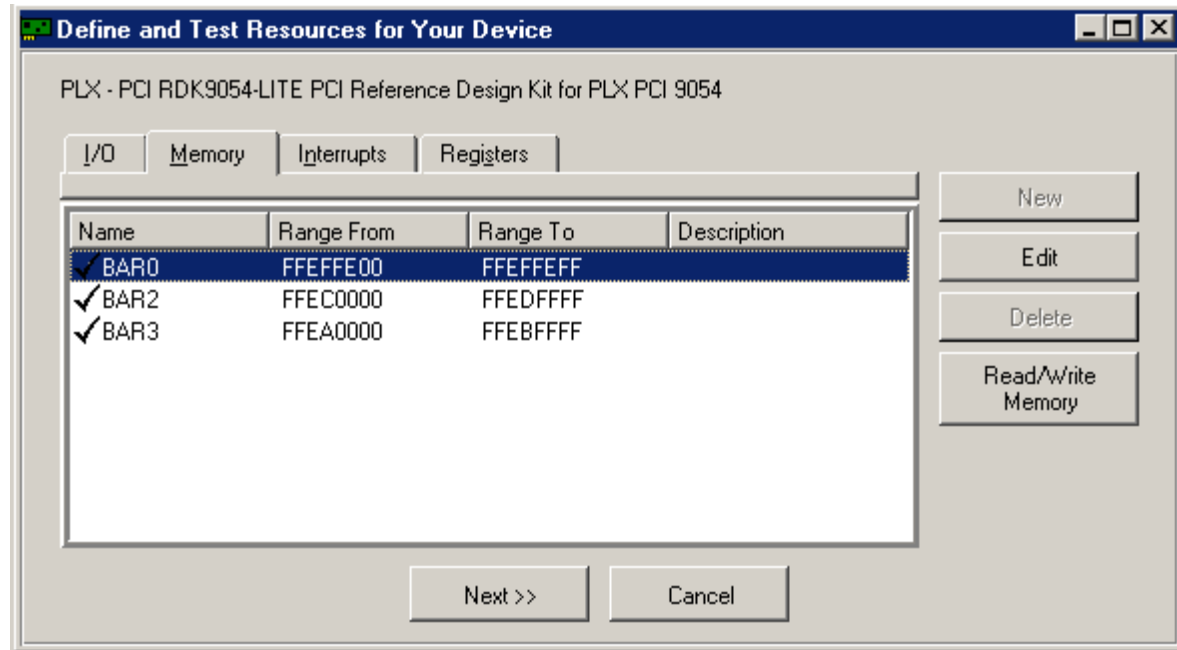
- (3) On **Windows 2000/XP/Server 2003** you can choose to automatically install the INF file from the DriverWizard by checking the **Automatically Install the INF file** option in the DriverWizard's INF generation dialogue. On **Windows 98/Me** you must install the INF file manually, using Windows **Add New Hardware Wizard** or **Upgrade Device Driver Wizard**, as explained in the WinDriver documentation. If the automatic INF file installation on Windows 2000/XP/Server 2003 fails, DriverWizard will notify you and provide manual installation instructions for this operating system as well.
- (4) Click **Next** in the INF generation dialogue in order to generate the INF file and install it (if selected).

- (5) When the INF file installation completes, select and open your device from the list described in step 2 above.

4. Detect / define your hardware's resources

- (1) DriverWizard will automatically detect your Plug-and-Play hardware's resources (I/O ranges, Memory ranges, PCI configuration registers and Interrupts). You can define additional information yourself, such as defining registers for your device as well as assigning read/write commands for these registers to the interrupt.

For non-Plug-and-Play hardware (ISA) – define your hardware's resources manually.



Interrupt Information

Name: ☒ Shared

Interrupt Number:

Interrupt Acknowledge:

Access Register	Command	Data
<input type="button" value="Not Defined"/>	<input type="button" value="Read"/>	<input type="text"/>

Description:

Register Information

Name: ☐ Auto Read

Resource Name: Access Mode:

Offset: Size:

Description:

Memory Information

Name:

Range From: Range To:

Description:

PCI Configuration Registers

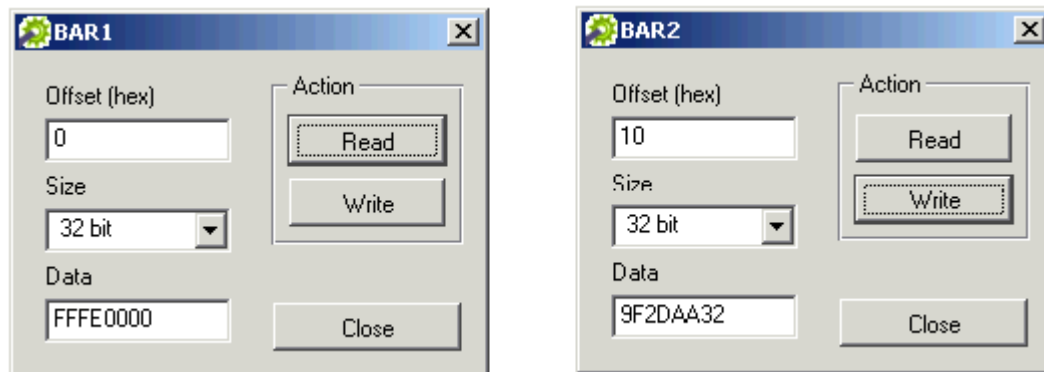
Name	Offset	Size	Data
VID	00	2	1085
DID	02	2	5406
CMD	04	2	0017
STS	06	2	0290
RID	08	1	08
CLCD	09	3	068000
CALN	0C	1	08
LAT	0D	1	05
HDR	0E	1	00
BIST	0F	1	00
BAR0	10	4	FFEFFE00
BAR1	14	4	0000FD01
BAR2	18	4	FFEC0000
BAR3	1C	4	FFEA0000
BAR4	20	4	00000000
BAR5	24	4	00000000
CIS	28	4	00000000
SVID	2C	2	1085
SDID	2E	2	9054

☐ Show all

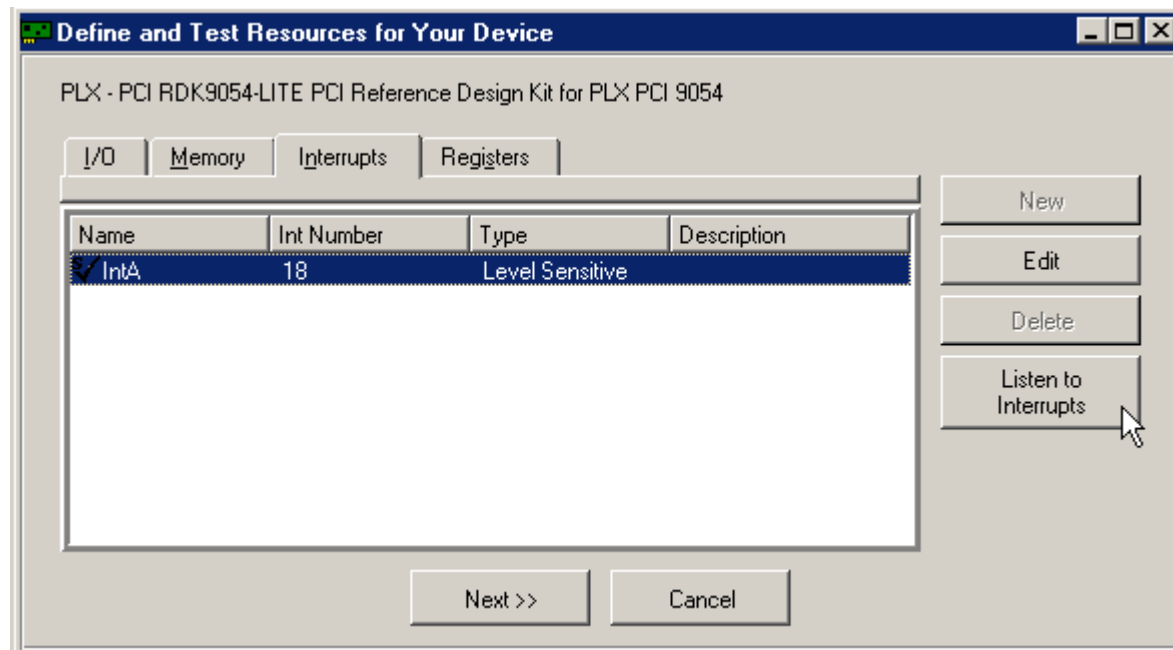
5. Test your hardware

Before writing your device driver, it is important to make sure your hardware is working as expected. Use DriverWizard to diagnose your hardware:

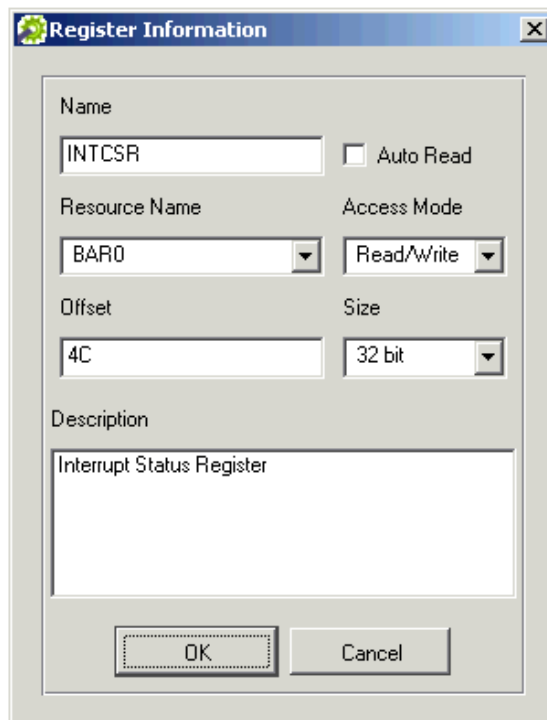
- Read and write to the I/O ports, memory space and your defined registers.



- "Listen" to your hardware's interrupts.



NOTE: For level sensitive interrupts, such as PCI interrupts, you must use DriverWizard to define the interrupt status register and assign the read/write command(s) for acknowledging (clearing) the interrupt, before attempting to listen to the interrupts with the wizard, otherwise the OS may hang! The specific interrupt-acknowledgment information is hardware-specific.

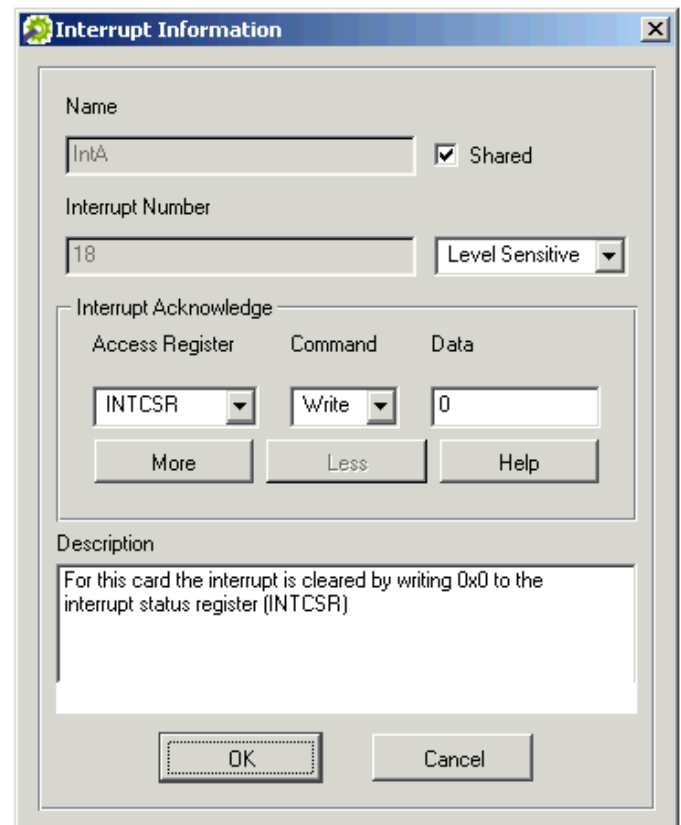


The **Register Information** dialog box is used to define the interrupt status register. It contains the following fields:

- Name:** A text box containing "INTCSR".
- Resource Name:** A dropdown menu showing "BAR0".
- Access Mode:** A dropdown menu showing "Read/Write".
- Offset:** A text box containing "4C".
- Size:** A dropdown menu showing "32 bit".
- Description:** A text area containing "Interrupt Status Register".
- Auto Read:** An unchecked checkbox.

Buttons at the bottom: **OK** and **Cancel**.

Define the interrupt status register



The **Interrupt Information** dialog box is used to define the transfer commands for acknowledging (clearing) the level sensitive interrupt. It contains the following fields:

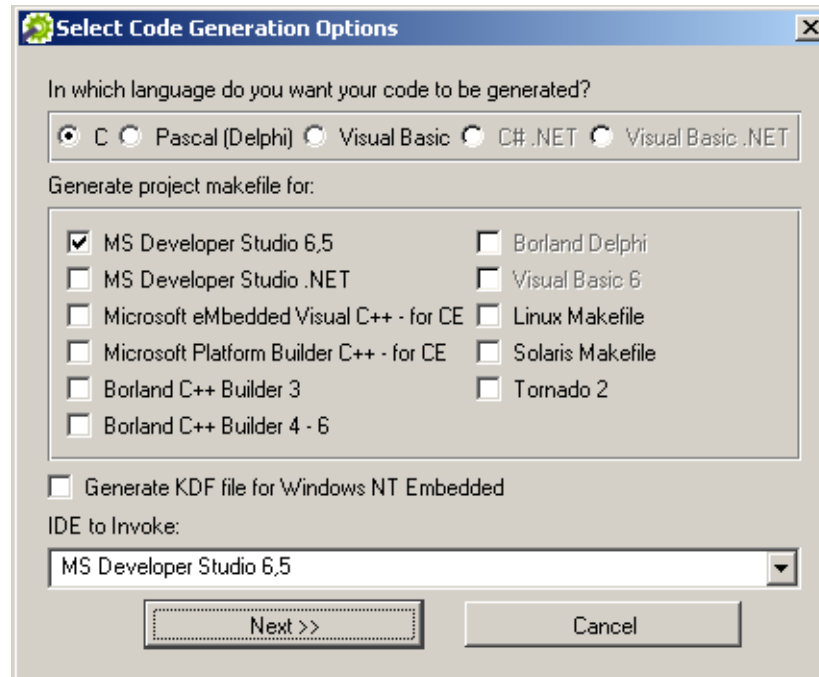
- Name:** A text box containing "IntA".
- Shared:** A checked checkbox.
- Interrupt Number:** A text box containing "18".
- Level Sensitive:** A dropdown menu showing "Level Sensitive".
- Interrupt Acknowledge:** A section containing:
 - Access Register:** A dropdown menu showing "INTCSR".
 - Command:** A dropdown menu showing "Write".
 - Data:** A text box containing "0".
- Description:** A text area containing "For this card the interrupt is cleared by writing 0x0 to the interrupt status register (INTCSR)".

Buttons at the bottom: **OK** and **Cancel**.

Define the transfer commands for acknowledging (clearing) the level sensitive interrupt

6. Generate the Driver Code:

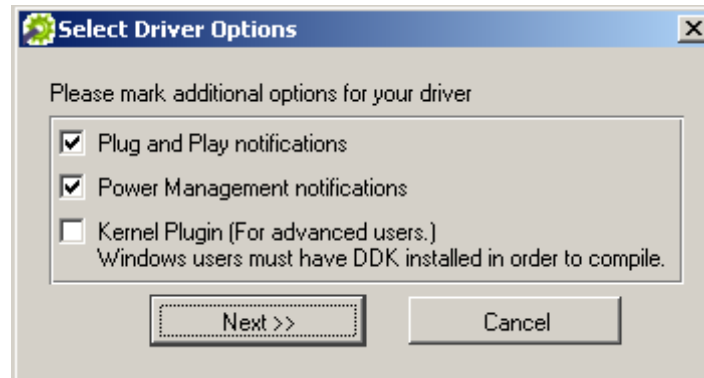
- (1) Use DriverWizard to generate your skeletal device driver. Click on the **Next** button or select the **Generate Code** option from the **Build** menu.
- (2) Choose the code language and indicate which development environments you would like to have project/make files for:



The dialog box titled "Select Code Generation Options" contains the following elements:

- A question: "In which language do you want your code to be generated?"
- A row of radio buttons for language selection: ☒ C, ☐ Pascal (Delphi), ☐ Visual Basic, ☐ C#.NET, and ☐ Visual Basic .NET.
- A section titled "Generate project makefile for:" containing a list of development environments with checkboxes:
 - ☒ MS Developer Studio 6,5
 - ☐ MS Developer Studio .NET
 - ☐ Microsoft eMbedded Visual C++ - for CE
 - ☐ Microsoft Platform Builder C++ - for CE
 - ☐ Borland C++ Builder 3
 - ☐ Borland C++ Builder 4 - 6
 - ☐ Borland Delphi
 - ☐ Visual Basic 6
 - ☐ Linux Makefile
 - ☐ Solaris Makefile
 - ☐ Tornado 2
- A checkbox: ☐ Generate KDF file for Windows NT Embedded
- A label: "IDE to Invoke:"
- A dropdown menu currently showing "MS Developer Studio 6,5".
- Two buttons at the bottom: "Next >>" and "Cancel".

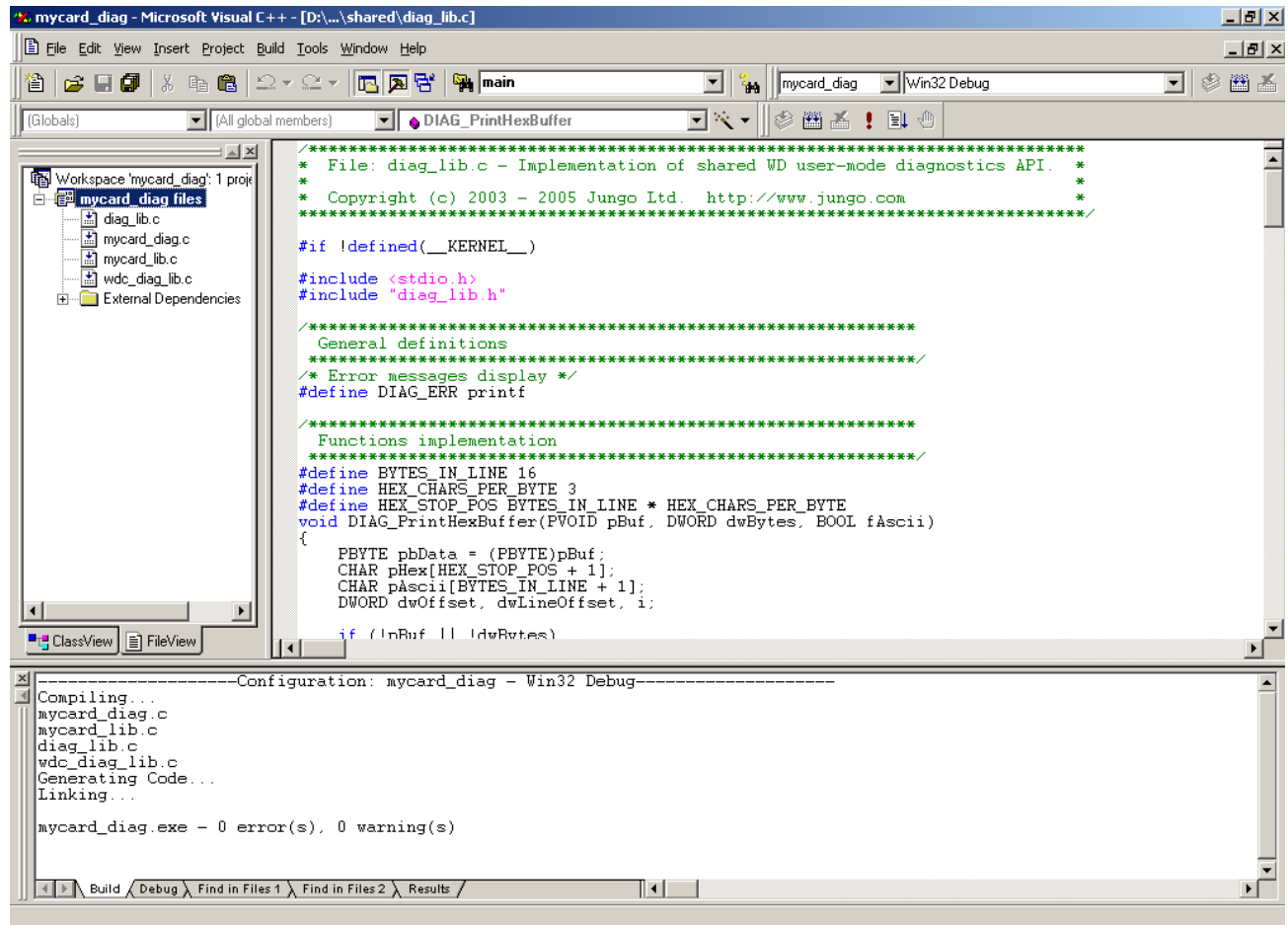
- (3) Indicate whether you wish to handle Plug-and-Play and power management events from within your driver code and whether you wish to generate Kernel PlugIn code. (Note: In order to build a Kernel PlugIn driver on Windows you must have an appropriate Microsoft DDK installed):



- (4) Click **Next**. DriverWizard will launch the desired development environment you chose in step 6.2 above.

7. Compile and run

- The following code is generated:
 - API for accessing your hardware from the application level (and from the kernel).
 - A sample application that uses the above API to access your hardware.
 - Project/make files for all of the selected build environments.
 - An INF file for your device (for Plug-and-Play hardware on Windows 98/Me/2000/XP/Server 2003).
- Use the project/make file that DriverWizard generated with your selected compiler..
- Compile the sample diagnostics application, and run it! This sample is a robust skeletal code for your final driver.
- Modify the sample application to suit your application needs, or start from one of the many samples provided with WinDriver.



The screenshot shows the Microsoft Visual C++ IDE with the following components:

- File Explorer:** Shows the project structure for 'mycard_diag'. The 'mycard_diag files' folder contains:
 - diag_lib.c
 - mycard_diag.c
 - mycard_lib.c
 - wdc_diag_lib.c
 - External Dependencies
- Source Code Editor:** Displays the content of 'diag_lib.c'. The code includes:


```

      /*****
      * File: diag_lib.c - Implementation of shared WD user-mode diagnostics API.
      * Copyright (c) 2003 - 2005 Jungo Ltd. http://www.jungo.com
      *****/

      #if !defined(__KERNEL__)

      #include <stdio.h>
      #include "diag_lib.h"

      /*****
      * General definitions
      *****/
      /* Error messages display */
      #define DIAG_ERR printf

      /*****
      * Functions implementation
      *****/
      #define BYTES_IN_LINE 16
      #define HEX_CHARS_PER_BYTE 3
      #define HEX_STOP_POS BYTES_IN_LINE * HEX_CHARS_PER_BYTE
      void DIAG_PrintHexBuffer(PVOID pBuf, DWORD dwBytes, BOOL fAscii)
      {
          PBYTE pbData = (PBYTE)pBuf;
          CHAR pHex[HEX_STOP_POS + 1];
          CHAR pAscii[BYTES_IN_LINE + 1];
          DWORD dwOffset, dwLineOffset, i;

          if (!pBuf || !dwBytes)
      
```
- Output Window:** Shows the build configuration 'mycard_diag - Win32 Debug' with the following output:


```

      Compiling...
      mycard_diag.c
      mycard_lib.c
      diag_lib.c
      wdc_diag_lib.c
      Generating Code...
      Linking...

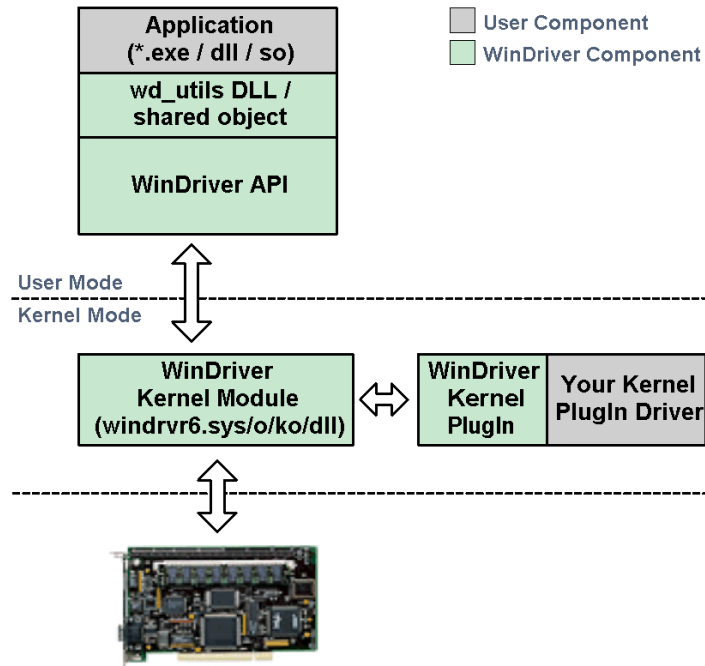
      mycard_diag.exe - 0 error(s), 0 warning(s)
      
```

Questions & Answers:

Q: How does WinDriver work?

A: With WinDriver, your device driver is developed in the user mode (as part of your application or as a separate DLL). This dramatically shortens development time by enabling you to use your standard 32-bit tools (MSDEV/Visual C++, Borland, Delphi, Visual Basic etc.) to develop and debug your driver.

The device driver developed with WinDriver (YourApp.exe) accesses your hardware through the WinDriver kernel module (**windrvr6.sys/o/ko/dll**) using the standard WinDriver functions.



Q: How can I achieve optimal performance with WinDriver?

A: After your driver is complete, for optimal performance you can easily transfer the performance-critical portions of your driver code (Interrupt handlers, I/O handlers, etc.) to WinDriver's Kernel PlugIn, which runs at the kernel-mode level.

For example – write your Interrupt Handler code in the user mode. After debugging this code in the user mode, you can move the code into the Kernel PlugIn. This will cause your interrupt handler to be executed in the kernel level, thereby allowing it to operate at maximal performance.

This architecture enables you to develop and debug all of your driver code in the user mode, using WinDriver's API, and then migrate only the performance-critical portions of the code to the kernel mode via the simple Kernel PlugIn mechanism.

Practical Exercises:

The following exercises take you through some of WinDriver's functionality in a quick 5-minute session. You can try these exercises after downloading the WinDriver 30 days evaluation from: <http://www.jungo.com/download.html>.

Exercise #1: Reading and writing to PCI memory

Goal: Learn how to read and write to a PCI memory range, and how to define registers.

Overview: This exercise will demonstrate how you can read and write to your PCI card's memory through the Driver Wizard, and generate an application that does the same. You will do this by reading and writing to your PCI (or AGP) screen card.

Exercise Steps:

- Step #1:** Start DriverWizard from the **Start** menu – **Start | Programs | WinDriver | DriverWizard** (on Windows) or by running **/WinDriver/wizard/wdwizard**.
- Step #2:** In the Wizard's menu, press **File | New Host Driver Project**.
- Step #3:** Choose your screen card from the list of Plug-and-Play cards displayed in the dialogue. Locate your screen card by identifying the name of your card's vendor in the displayed list.
- Step #4:** Press the **Memory** tab. Your card's memory ranges will be displayed. One of these memory ranges is mapped to the screen – i.e. bytes in the memory range correspond to pixels on the screen (This is usually Bar 0. Look for the bigger sized memory range). Read from offset 0 (the top left of the screen). Now move a window over the top left of the screen to change its color and read again. If the value changed – than this is the correct memory range. Now write to this offset (try FFFFFFFF and 00000000 alternately), and see the color of the pixel change!
- ** Note:** Writing to the wrong memory range can freeze the computer.
- Step #5:** Define a register called "TopLeft" which represents the top left of the screen (i.e. the correct memory range, and offset 0). In the description enter: "This register represents the top left pixel on the screen". Read and write to this register. Define a register called "Somewhere", whose offset is FF (i.e. some other pixel on the screen) and enter the following description: "This register represents a pixel somewhere on the screen".

- Step #6:** Press **Next** or select **Build | Generate Code** in the menu and proceed to generate code. Driver Wizard will create the functions to access your hardware's resources. You can call these functions directly from your application in the user mode. DriverWizard will also create a sample application that uses these functions to access your device!
- Step #7:** Compile and run the sample application. Use it to read and write from your screen card.

That's how simple it is – try it!

**** Note:** You can now copy the sources of the project to any other supported operating system (Windows 98/Me/NT/2000/XP/Server 2003/CE.NET, Linux, Solaris and VxWorks), recompile, and run again!

A part of the API generated by DriverWizard in Exercise (where screencard is the name selected for the driver project when generated the code with DriverWizard):

```
<screencard\_lib.h>

/* SCREENCARD run-time registers */
/* [Values should correlate to the registers' indexes in the gSCREENCARD_Regs array] */
typedef enum {
    SCREENCARD_TopLeft,    /* TopLeft - This register represents the top left
                           pixel on the screen */
    SCREENCARD_Somewhere, /* Somewhere - This register represents a pixel
                           somewhere on the screen */
    SCREENCARD_REGS_NUM,  /* Number of run-time registers */
} SCREENCARD_REGS;

DWORD SCREENCARD_LibInit(void);
DWORD SCREENCARD_LibUninit(void);

WDC_DEVICE_HANDLE SCREENCARD_DeviceOpen(const WD_PCI_CARD_INFO *pDeviceInfo);
BOOL SCREENCARD_DeviceClose(WDC_DEVICE_HANDLE hDev);

DWORD SCREENCARD_IntEnable(WDC_DEVICE_HANDLE hDev,
                           SCREENCARD_INT_HANDLER funcIntHandler);
DWORD SCREENCARD_IntDisable(WDC_DEVICE_HANDLE hDev);
BOOL SCREENCARD_IntIsEnabled(WDC_DEVICE_HANDLE hDev);
```

```
DWORD SCREENCARD_EventRegister(WDC_DEVICE_HANDLE hDev,
                                SCREENCARD_EVENT_HANDLER funcEventHandler);
DWORD SCREENCARD_EventUnregister(WDC_DEVICE_HANDLE hDev);
BOOL SCREENCARD_EventIsRegistered(WDC_DEVICE_HANDLE hDev);

DWORD SCREENCARD_GetNumAddrSpaces(WDC_DEVICE_HANDLE hDev);
BOOL SCREENCARD_GetAddrSpaceInfo(WDC_DEVICE_HANDLE hDev,
                                  SCREENCARD_ADDR_SPACE_INFO *pAddrSpaceInfo);

<screencard_diag.c>

/* -----
   SCREENCARD run-time registers information
   ----- */
/* Run-time registers information array */
const WDC_REG gSCREENCARD_Regs[] = {
    { AD_PCI_BAR1, 0x0, WDC_SIZE_8, WDC_READ_WRITE, "TopLeft",
      "This register represents the top left pixel on the" },
    { AD_PCI_BAR1, 0x50, WDC_SIZE_8, WDC_READ_WRITE, "Somewhere",
      "This register represents a pixel somewhere on the " },
    };
const WDC_REG *gpSCREENCARD_Regs = gSCREENCARD_Regs;
```

Exercise #2: Handling Interrupts

Goal: Learn how to test your hardware's interrupts, and write an interrupt handler.

Overview: In this exercise you will use DriverWizard to "Listen" to the interrupts that your floppy disk drive generates. You will then use DriverWizard to generate an application that listens to this interrupt and handles it with a user-mode interrupt handler.

Exercise Steps:

- Step #1:** Start DriverWizard from the **Start** menu – **Start | Programs | WinDriver | DriverWizard** (on Windows) or by running **/WinDriver/wizard/wdwizard**.
- Step #2:** In the DriverWizard menu, press **File | New Host Driver Project**. DriverWizard will now display a list of the Plug-and-Play devices connected to your machine.
- Step #3:** Since we will be using the floppy disk drive, choose ISA from the menu.
- Step #4:** Press the **Memory** tab.
- Define at least one memory range (it can be a dummy range). Simply click **New** in the memory tab and then click **OK** to create a memory range for addresses 0x0-0x0, which will be enough for the purpose of successfully compiling and building the generated code and testing the floppy disk interrupt handling.
- Step #5:** Press the **Interrupts** tab.
- Define the Floppy Drive interrupt: Name the interrupt "FloppyInterrupt" and choose 6 as the interrupt number.
- Step #6:** Right-click the mouse on the interrupt you created, and press **shared**. This is required in order to share the floppy drive interrupt with the operating system.
- Step #7:** Press **Listen to Interrupt**.
- To see the floppy drive interrupts: Access the floppy drive (for example, by writing "a:" in a DOS console screen, or by choosing the floppy drive in your file browser).
- Step #8:** Press **Next** or select **Build | Generate Code** in the menu – DriverWizard will create the functions to access your resources and handle the interrupt you have defined. You can call these functions directly from your application in the user mode. DriverWizard will also create a sample application that uses these functions to access YOUR device!
- Step #9:** Compile and run the sample application.
- Step #10:** Enable and listen to your interrupt from within the sample application – see the floppy drive interrupts. Later, modify the interrupt handler and insert your own functionality into it.

That's how simple it is – try it!

**** Notes:**

1. Using WinDriver's Kernel PlugIn feature, your interrupts and I/O calls can be handled from kernel mode, thereby achieving optimal performance!
2. You may also test your PS Mouse interrupts (Interrupt 12 under Windows NT), and your Keyboard Interrupts (Interrupt 1 Under Windows NT).