JUNGO™
Connecting Software to Hardware

Tackling the Driver Development Challenge

Developing high performance, cross platform device drivers in user mode using the "WinDriver" methodology

1. Device Driver Overview

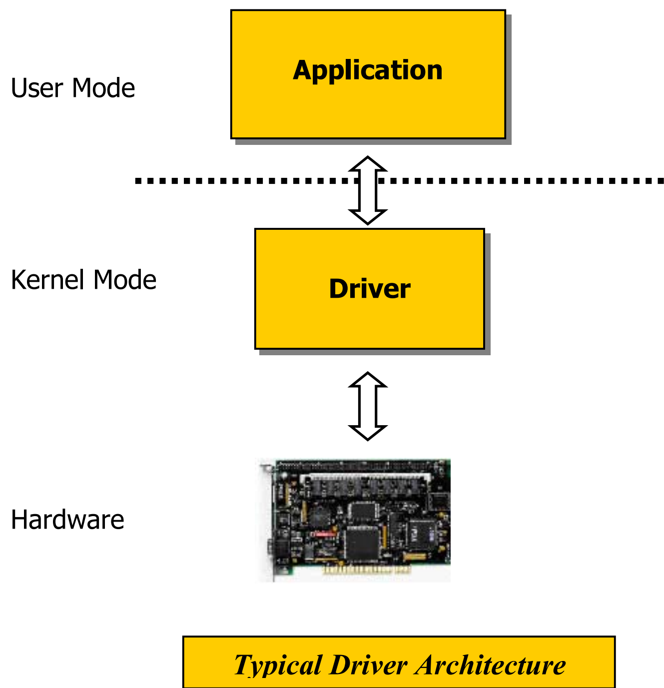
a. What is a Device Driver?

Device drivers shield the cumbersome details of real hardware devices from regular applications. They allow the application programmers to focus on the application problem and not worry about the low-level details of the hardware device.

Multi-process operating systems (such as Windows and Unix) isolate applications running in user mode (application space), by limiting their access to system resources. In these operating systems, applications are not able to access hardware directly. Hardware access is only available to code running at ring-zero, or "Kernel Mode". Therefore, to access hardware, a developer will have to write a device driver in the kernel mode before accessing the hardware from the user mode.

The code that runs at kernel mode privileges and accesses the specific hardware device is called "Device Driver". A device driver is typically the only entity that knows the internal workings of the specific hardware that it was written for, and is the only entity that physically accesses the hardware.

Therefore, a typical application / device driver / hardware layout will have the following architecture:



Typical Driver Architecture

In order to access the hardware, the device driver must call operating system specific APIs, and must conform to the operating system's specific method of writing, linking, and loading device drivers.

b. Basic Knowledge Necessary for Developing a Device Driver

Operating System Architecture

The device driver is tightly coupled to the operating system it is running on. Thorough knowledge of the operating system architecture and internals is required in order to develop a device driver.

Operating System Driver Interfaces

The operating system interfaces that your driver will use to access the hardware must also be studied. In Microsoft Windows 95/98/NT/200/Me/XP, the set of driver interfaces for the operating system are called the DDK (Driver Development Kit). In Microsoft Windows CE, this set is called the ETK (Embedded Toolkit).

Drivers use an operating specific API (e.g. the DDK), requiring developers to write separate device drivers for each different operating system that the hardware is required to run on. Writing a device driver for Windows 95 requires knowledge of the Windows 95 architecture, and Windows 95 DDK, and would not work on other Windows platforms (e.g Windows 2000).

Kernel Mode Debugging

Debugging device drivers is a challenging task that requires special knowledge, and a new set of tools.

When debugging applications, debuggers are launched as separate processes. The debugger analyzes the process being debugged by halting the subject process and stepping through it. Since a device driver is an integral part of the operating system, if it were to be halted by a debugger, the whole operating system would be halted, and no other process (including the debugger) would be able to run. Therefore, debugging a device driver involves connecting two computers with a cable, and using one computer as the host running the debugging software, and the other machine acting as a target where the driver to be debugged resides.

c. Types of Device Drivers

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 I/O control commands (IOCTLs), and drives the hardware by calling the different DDK, ETK, DDI/DKI functions. Monolithic drivers are encountered in all operating systems including all Windows and Unix platforms.

Layered drivers

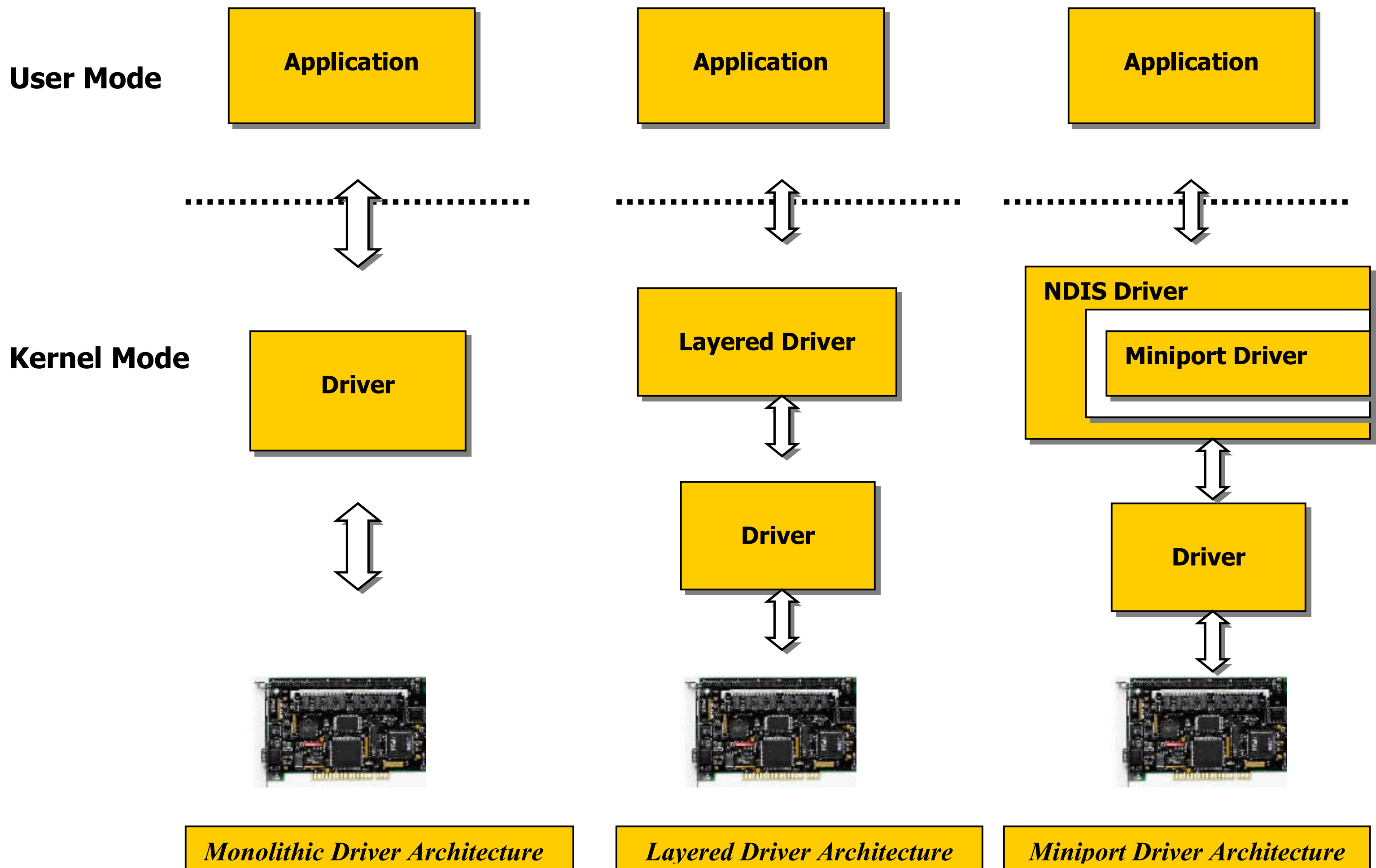
Layered drivers are device drivers that are part of a "stack" of device drivers, that together process an IO request. An example of a layered driver is a driver which intercepts calls to the disk, and

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.

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

An example of a Miniport driver is the "NDIS" Miniport driver. The NDIS Miniport framework is used to create network drivers which hook up to Windows 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 only has to write the code that is specific to the network card that he is developing.



Unix Device Drivers

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

2. The Driver Development Cycle

Writing device drivers requires special skills and is a lengthy process, which must be repeated for every new operating system to be supported. The basic driver development process is as follows:

1. Learn the internals of the operating system to be supported (Windows/Linux/VxWorks…)
2. Learn how to write a device driver on this operating system (for example DDK)
3. Master new tools for development/debugging in kernel mode
4. Write the kernel mode device driver which does the basic hardware input/output (in kernel mode)
5. Write the application in user mode, which accesses the hardware through the device driver written in kernel mode

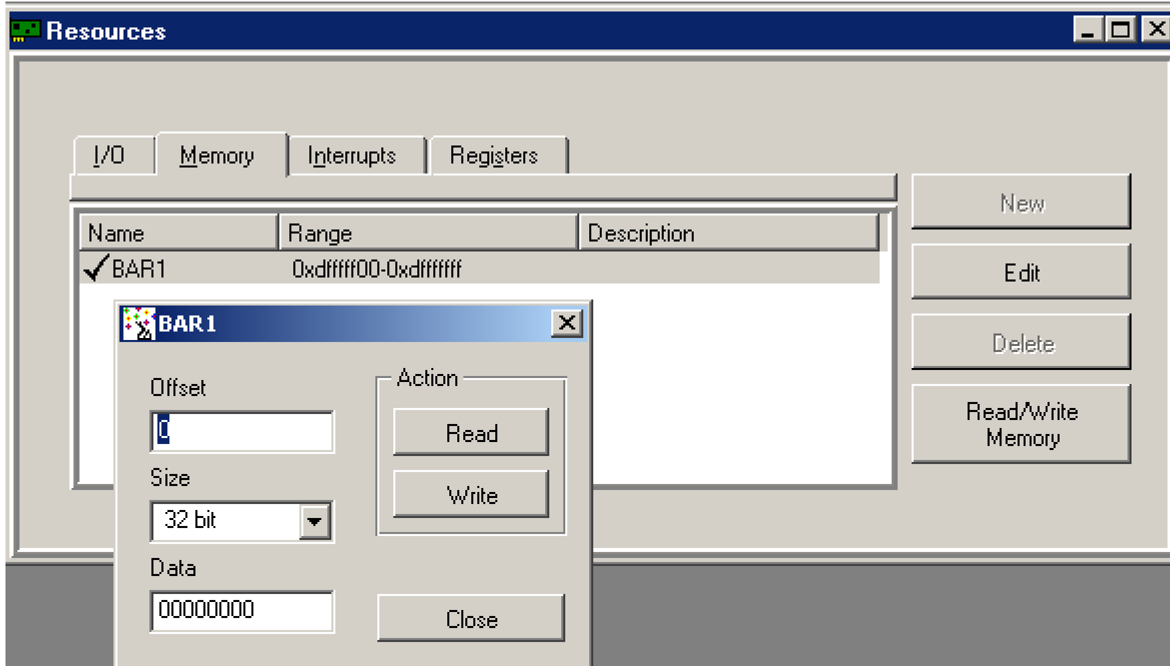
Repeat steps 1-4 for each new operating system on which the code should run.

Based on these steps, a driver development toolkit should automate and simplify the process by providing the means to create applications that access new or existing hardware, without the need to write a kernel mode device driver thereby saving weeks/months of work. Significant additional value is provided if it includes source code compatibility between different operating systems.

Lets work through the development steps to build a model for a driver development toolkit.

3. Working With Untested Hardware – The First Hurdle

Driver development being a time-consuming task, developers must be certain that their hardware is functioning as expected. Ideally, a driver development toolkit should enable diagnosis and interrogation of hardware before having to develop a device driver, meaning read/write to I/O, memory and Registers that the card supports and/or to the USB device's pipes, without having to write a driver. Jungo's driver development toolkit, WinDriver, provides a hardware debugging utility, "DriverWizard" that automatically detects the device's plug and play resources. If the hardware device is not plug-n-play (i.e. ISA), the card's resources can be defined manually according to the device's specification. The Wizard will enable read and write actions to your card, and allow listening of interrupts generated by the hardware device.

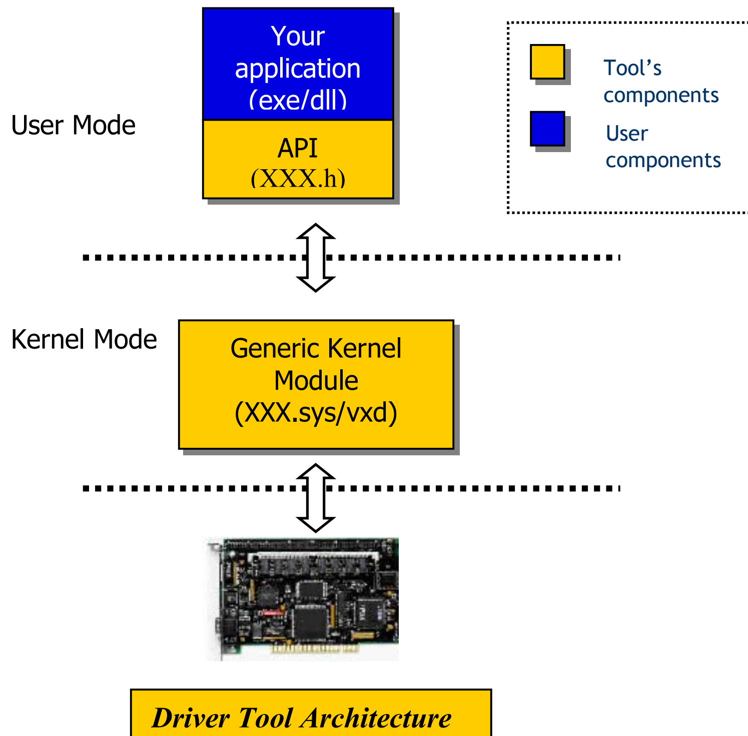


The DriverWizard GUI demonstrating read/write to the hardware's memory

4. Developing in User Mode – An Architecture That Provides a Solution

The recommended architecture should "hide" kernel mode development from the user. This can be achieved by a hardware-access API, available to the application in user mode. The tool, having enabled hardware diagnostics, automatically generates a hardware specific driver code (based on the hardware's resources that were detected). The hardware-specific driver code (the application) uses the API to access the hardware through a generic kernel module, which resides at the kernel level and runs in Ring-0 privileges. The API enables the application to access the hardware's registers, memory ranges and IO ranges, and handle the hardware's interrupts – for PCI/ISA, or transfer data on the device's pipes - for USB

The architecture of a hardware access application based on this structure would be as follows:



Driver Tool Architecture

The above architecture eliminates the need for operating system internal and bus protocol knowledge, as well as kernel mode development, resulting in a reduction in development time.

WinDriver, Jungo's driver development toolkit provides this basic architecture and in practice the tool operates as follows:

Once the hardware is functioning as expected, DriverWizard, automatically generates a cross platform hardware specific API and a simple diagnostics utility source code, which accesses all elements of the hardware through the new hardware specific API that it generated. (See the WinDriver Architecture Diagram below.) This application is a skeleton for the final application and the developer is only required to add his functionality requirements to the code. The Wizard also generates Makefiles for the most common environments (Microsoft Visual Studio, Borland Builder, Linux gmake, and others). The generated code can be compiled and run without modification.

Using the C/C++ output generator, DriverWizard produces the following elements in a subdirectory of your choice:

- XXX_files.txt – A readme file describing the generated files.
- XXX_diag.c – A fully working skeleton application demonstrating the use of the library functions created by DriverWizard for your device.
- XXX_lib.c – A general library of utility functions for device access used by XXX_diag.c.
- XXX_lib.h – A header file for the utility functions.
- Other project files for the selected development platform.

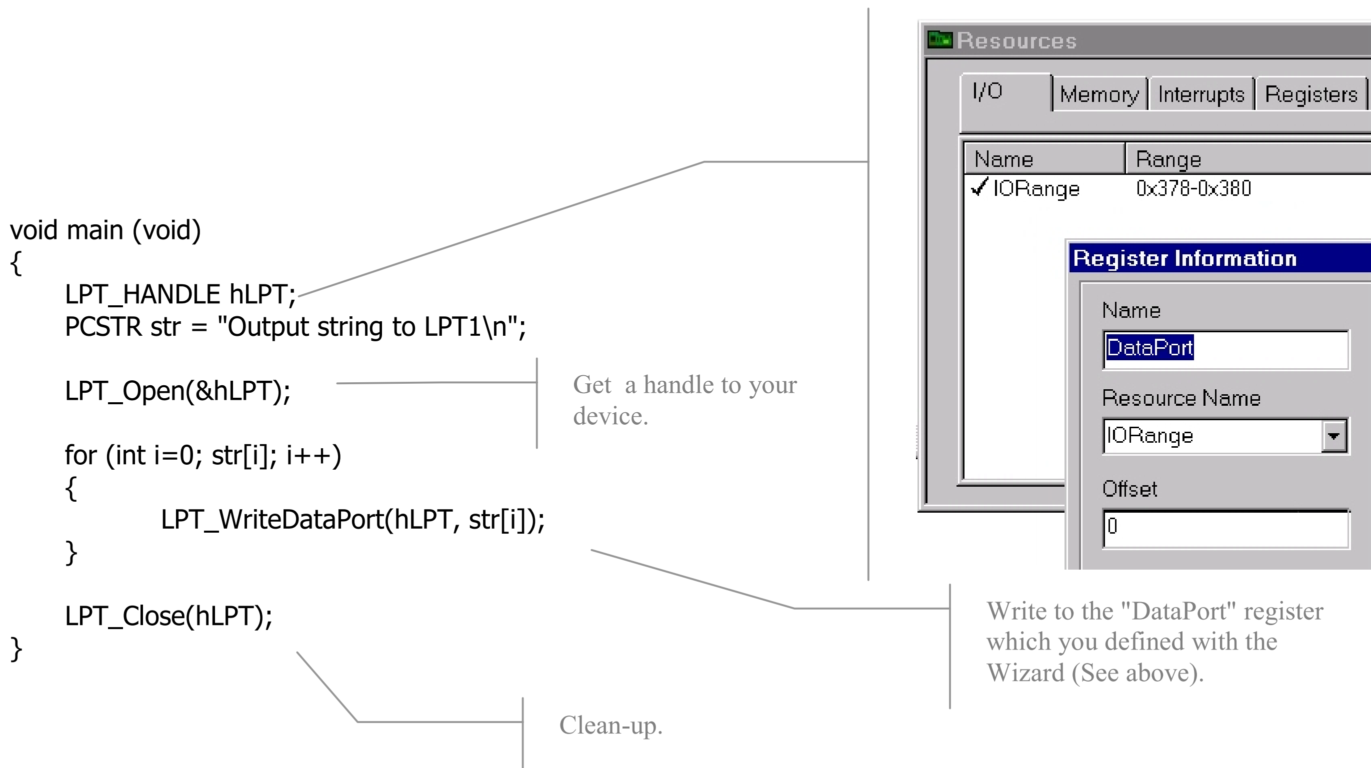
Where XXX is the name of the project.

5. The WinDriver API – An Example

For example, if your hardware has a register called "StatusRegister" which is located in the Bar0 range of your card, the "Hardware Specific" layer might implement the following function:

```
DWORD YourHardware_ReadStatusRegister(…)
{
        …
        return WD_Transfer(…, Bar0, Offset0, READ, …);
}
```

The following sample uses an API generated by the WinDriver Wizard, to send a string of characters to an IO range (LPT1). The "LPT" API was generated by WinDriver's DriverWizard for the LPT hardware (See box).

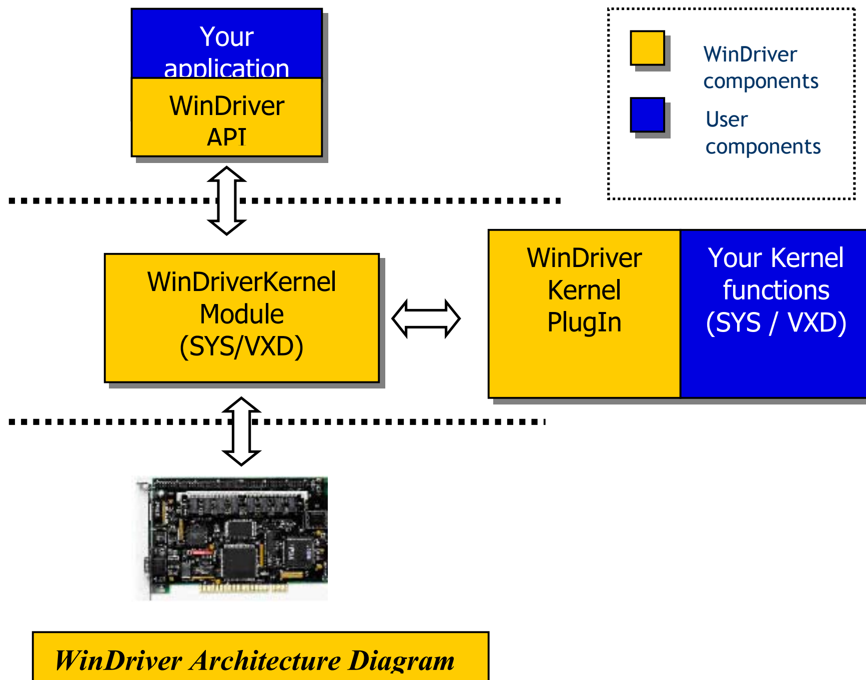


Generic WinDriver API

Initialization & Cleanup WD_Open() WD_Close() **For PCI:** WD_CardRegister() WD_CardUnregister() WD_PciScanCards() WD_PciGetCardInfo() WD_PciConfigDump() WD_IsapnpScanCards() WD_IsapnpGetCardInfo() WD_IsapnpConfigDump() **For PCI:** **I/O and Memory access** WD_Transfer() WD_MultiTransfer()	**For PCI:** **DMA** WD_DMALock() WD_DMAUnlock() **For PCI:** **Interrupt handling** InterruptThreadEnable() InterruptThreadDisable() Low level: WD_IntEnable() WD_IntDisable() WD_IntCount() WD_IntWait() **For USB:** WD_UsbScanDevice() WD_UsbGetConfiguration() WD_UsbDeviceRegister() WD_UsbDeviceUnregister() WD_UsbTransfer() WD_UsbResetPipe() WD_UsbResetDevice() WD_UsbResetDeviceEx()	**Kernel PlugIn** WD_KernelPlugInOpen() WD_KernelPlugInClose() WD_KernelPlugInCall() **Utility & Debugging** WD_DebugAdd() WD_DebugDump() WD_Sleep()

6. Performance

The driver development tool architecture shown in Diagram A above can incur function call overhead from the kernel to the user mode due to memory and interrupts being handled at the application level. This problem can be overcome by a tool architecture that enables running of performance critical parts of the code at kernel level - i.e. allow moving of these parts of code, such as interrupts handlers, to the kernel, while keeping most of the code intact. This would enable the developer to start with quick and easy development in User Mode and progress to performance oriented code only where needed. WinDriver provides a "Kernel PlugIn" architecture, which enables moving performance critical code from User Mode to Kernel Mode, to enable optimal performance. The WinDriver architecture is shown below. Note: The WinDriver components are in orange and the User's components are in blue.



WinDriver Architecture Diagram

In most cases, Kernel PlugIn will not be used, as the WinDriver memory transfer commands are executed at the kernel level in any event. The only cases where performance considerations will require using the Kernel PlugIn, is when the hardware particularly requires a very high interrupt rate, or when the hardware's memory is not memory mapped (i.e. IO mapped). For PCI devices, a simple hardware modification can transform hardware from being IO mapped, to being memory mapped. In the case of memory mapped cards, WinDriver provides a user mode pointer which can be used to transfer data to/from the card's memory directly from the user mode, thus eliminating the need to call the WD_Transfer() API and improving the performance.

7. Debugging

As device drivers take up a significant portion of the total code base executed in kernel mode, it is essential that this large code base be reliable to ensure reliability of the system. Drivers developed in user mode must be monitored to ensure that they do not make illegal function calls or cause system corruption, which involves kernel mode debugging.

WinDriver provides the "DebugMonitor" utility, a powerful graphical and console mode tool for monitoring all activities handled by the WinDriver Kernel (*windrvr.sys/ windrvr.vxd / windrvr.dll / windrvr.o / wdpnp.sys*). DebugMonitor allows monitoring of how each command sent to the kernel is executed.

8. Specific Chip-Set APIs

PCI hardware is equipped with a PCI bridge that provides the PCI functionality to the hardware, and is responsible for communicating with the PCI controller, and for translating PCI bus based

data to the local bus on the hardware itself. PCI bridges can be purchased as separate devices, or as IP cores, which are loaded onto an FPGA on your hardware. Some of the PCI bridges provide DMA functionality (slave and or master DMA), which is utilized by programming the device's registers.

Based on the driver development tool architecture described above, Jungo's driver development toolkit includes vendor-specific APIs for both PCI and USB silicon. Enhanced support includes definitions of all of the PCI bridge registers and memory ranges, and pre-programmed DMA functionality which enables DMA to and from the hardware without having to learn the specific PCI bridge being used; for USB it includes vendor specific operations, including firmware download, pipes reset, device reset and others.

9. Cross Operating System Support

A driver development tool based on the architecture described, provides the driver developer with an API in the application level for accessing hardware. This API calls the kernel module, which does the actual hardware access through the operating system specific kernel APIs. If kernel modules for various operating systems are provided, these will allow the driver to be ported to the operating systems without modification.

Drivers developed using WinDriver are source code compatible between all supported operating systems (WinDriver currently supports Windows 95/98/Me/NT/2000/XP/CE, Linux, Solaris and VxWorks.). The code is binary compatible between Windows 95/98/Me/NT/2000/XP. The same executable created will operate on Windows 95, 98, Me, NT, 2000 and XP. For Unix systems the source is compatible and only requires recompilation. Even if your code is intended only for one of these operating systems, WinDriver provides the flexibility to port your driver to other operating systems without modifying your code.