A DEVICE DRIVER ARCHITECTURE FOR TELEMETRY APPLICATIONS

Marc C. DiLemmo
Aydin Telemetry

ABSTRACT

This paper illustrates a device driver implementation used to support a PC compatible telemetry device. Device requirements included operation on Windows NT 4.0, Windows 95, Windows NT 5.0 and Windows 98 platforms. A single device driver was not possible due to the differences between driver requirements on the various operating systems. The Windows Driver Model (WDM) was considered for NT 5.0 and Win98, however, NT 4.0 and Win95 does not support the WDM. To minimize software development and support efforts, it was clear that an architecture compatible to both WDM, NT 4.0 and Windows 95 needed to be developed. The resulting layered device driver architecture provides a common upper interface and uses a register based model to describe the hardware at the lower interface. The common upper interface is compatible with all of the target operating systems and presents a consistent Applications Programming Interface (API) for the telemetry application developer. The lower interface is specific for each platform but contains minimal device specific functionality. A simple register I/O driver is easily implemented using all of the target operating systems. The layered architecture and register based interface to the hardware results in a multiple operating system code set which differs only at the lowest layer.

KEY WORDS

Device Drivers, Telemetry Application Software, WDM, Windows Operating Systems.

INTRODUCTION

Operating system specific components of a telemetry hardware and software application make it difficult to produce a product which can be hosted on multiple platforms. Specifically, developing an application which must operate under current and future versions of the Microsoft Windows operating systems can become a difficult software development and maintenance problem. Clearly, an application and device driver architecture, which allows a single software code set to function correctly regardless of the OS, is desired. Flexible interface standards between the various architecture components must be defined to allow it to support different telemetry hardware configurations.

A single application code set must exploit the common features of the subject OS's. Finding a common feature set usually means tradeoffs. Some of the newer, more preferred features of one OS may not be exploited because the other OS's do not have those features. Every attempt should be made to use these preferred features keeping the additional complexity to a minimum.

The telemetry application, which resulted in the subject device driver architecture, included a 20 Mbps bit synchronizer, the PCI3335. The project requirements dictated that this PCI based device must operate under the current and future versions of the Microsoft Windows operating systems. Specifically, Windows NT 4.0, Windows 95 and the soon to be released Windows NT 5.0 and Windows 98.

OPERATING SYSTEM SURVEY

All of the subject OS's utilize the Win32 API. This API provides much of the functionality required by a typical telemetry application. Included are user interface, file system, process management, thread management and standard hardware functions. Application specific telemetry hardware is the only functional area which must be provided.

The subject OS's require device drivers to manage optional hardware products. It is this area which differs the most between the OS's. Win95 uses the Virtual Device driver model (VxD.) WinNT 4.0 uses the objected oriented NT driver model. Win98 and WinNT 5.0 will use the Windows Driver Model (WDM.) Win98 will be backward compatible with the VxD. However, it is not clear if the NT driver model will change with the 5.0 release. The following table identifies which OS supports two features which have the most impact on telemetry device drivers.

OS	Driver Model	Object Oriented	I/O Control
Win95	VxD	No	Yes
WinNT 4.0	NT	Yes	Yes
Win98	WDM/VxD	Yes/No	Yes/Yes
WinNT 5.0	WDM/NT?	Yes/Yes	Yes/Yes

The Object Oriented driver feature represents the major difference between the various driver model implementations. This feature moves the responsibility for managing multiple instances of a device out of the vendor supplied driver into the I/O manager of the OS.

The end result is that each instance of a device is named and can be addressed from the application using that unique name. OS's using the VxD model do not have the capability to name individual devices being managed by a vendor supplied driver. The VxD implementation must manage multiple hardware instances internally.

The I/O Control feature is a device I/O capability provided by all of the OS versions. This capability provides a general purpose means of communicating with the desired device driver. The multiple hardware instance restriction identified above also affects the I/O Control function. However its general purpose nature can be used to work around this difference.

If backward compatibility with Win95 and WinNT 4.0 was not required, the obvious driver architecture would be based on the WDM. A WDM based device driver can be used on either Win98 or WinNT 5.0 supporting a single application code set. Since Win95 and WinNT 4.0 must be supported, the differences between the device driver models make a single code set is impossible.

HARDWARE & OPERATING SYSTEM ABSTRACTION

If operating system specific components can be abstracted in such a way as to provide a consistent interface path to the hardware, a significant portion of the application set can remain unchanged for each OS version. The closer the abstraction layer is to the OS differences, the larger the common code set will be. This same technique was used by Microsoft to produce the NT OS. NT's Hardware Abstraction Layer (HAL) is used to portray a consistent hardware model to the upper layers of the OS. The entire OS is portable between different computer hardware implementations and microprocessor architectures. The only component which changes is the HAL. The upper interface between the HAL and the OS is consistent. Therefore, the OS code set remains unchanged. See Figure 1. This same technique can be used at the application level to abstract OS differences and present a consistent interface to the telemetry hardware.

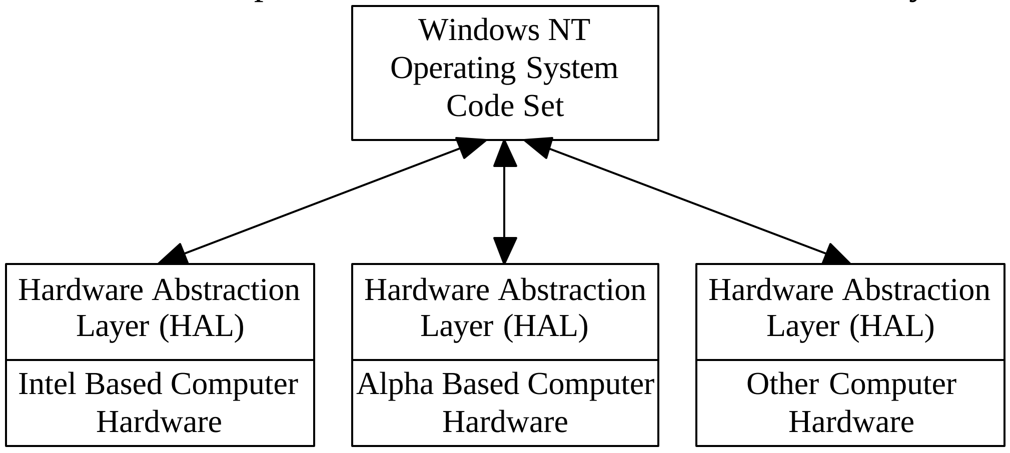
Windows NT Operating System Code Set
Hardware Abstraction Layer (HAL)
Intel Based Computer Hardware
Hardware Abstraction Layer (HAL)
Alpha Based Computer Hardware
Hardware Abstraction Layer (HAL)
Other Computer Hardware

Figure 1 Hardware Abstraction Technique

ABSTRACTION LAYER

The abstraction layer must interface a single telemetry application code set to a hardware device via several, very different OS's. If the abstraction layer is implemented at the Win32 level, which is common between all the subject OS's, a Win32 DLL can be used. The DLL can be designed to provide the consistent API to the application at its upper level and an I/O Control interface to the hardware at its lower level. Since a DLL can determine the version of the OS that is currently running, it can modify its lower level appropriately at run time. Therefore, the Object Oriented differences between the NT and VxD models can be accommodated. See Figure 2.

By describing the hardware with a register based model, the I/O Control interface, at the lower abstraction level, can be simplified. The Win32 DeviceIoControl function can be used to implement register read and write capabilities and other general support functions. This reduces the low level device driver to an interface between its register model and the actual hardware implementation.

Device abstraction at the upper level can provide many benefits to the application. The abstraction buffers the application from changes in telemetry hardware. An upgraded hardware device can easily be installed into the telemetry application simply by replacing the DLL, and possibly its associated lower level device driver, with a new version that maintains the upper abstraction level. For devices of similar functionality, a hardware device from a different vendor can be substituted and the application code set need not be changed in any way.

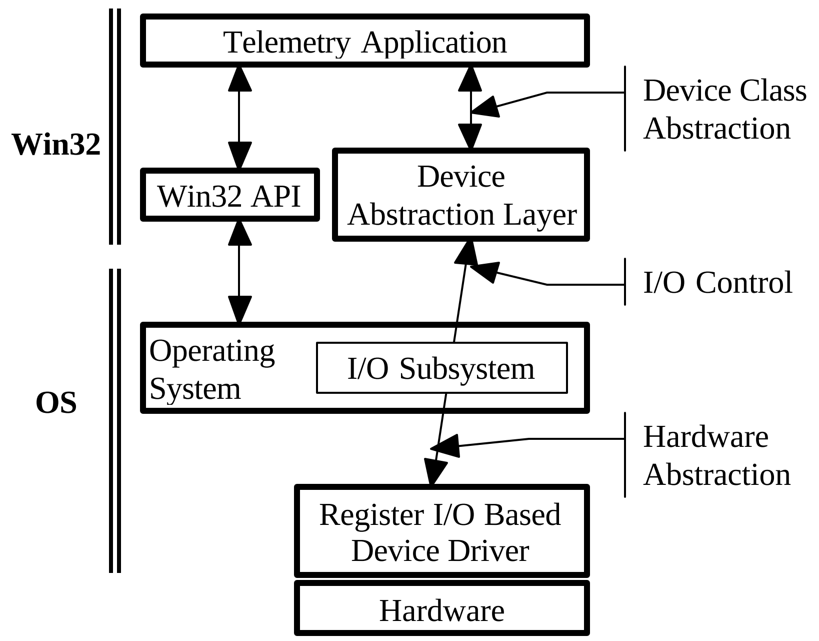

Figure 2 Abstraction Layer

The functions of the device architecture needed to be distributed between the abstraction layer and the hardware device driver. Due to the restrictions placed on the programmer at the device driver level, it was more prudent to locate all computation functions in the abstraction layer. Typically, only the low level hardware I/O and time critical functions appear within the device driver. The following table identifies the locations of the major functions.

Function Implementation	**Location**
Multiple device management	Abstraction Layer
Hardware simulation	Abstraction Layer
Format conversion and register computations	Abstraction Layer
Status data conversion	Abstraction Layer
Register and format caching	Abstraction Layer
Hardware I/O	Device Driver
Interrupt service, data distribution	Device Driver
Automated status polling	Device Driver

UPPER INTERFACE

The upper interface of the device abstraction layer presents a consistent programming model to the application. A standardized programming model was developed which would accommodate a broad range of telemetry devices. Regardless of the hardware purpose, the programming model contained the same function types with names which only differed by the device class. Function arguments consisted mainly of data structures which would change to support the desired hardware capabilities. This consistent programming model proved to be advantageous when multiple programmers were involved in application development.

The abstraction layer appears to the application developer as a library of C callable functions. When each device class is developed, a programmers reference manual, import library and include file are distributed along with the DLL. Figure 3 depicts the relationship between the application and the device abstraction layer.

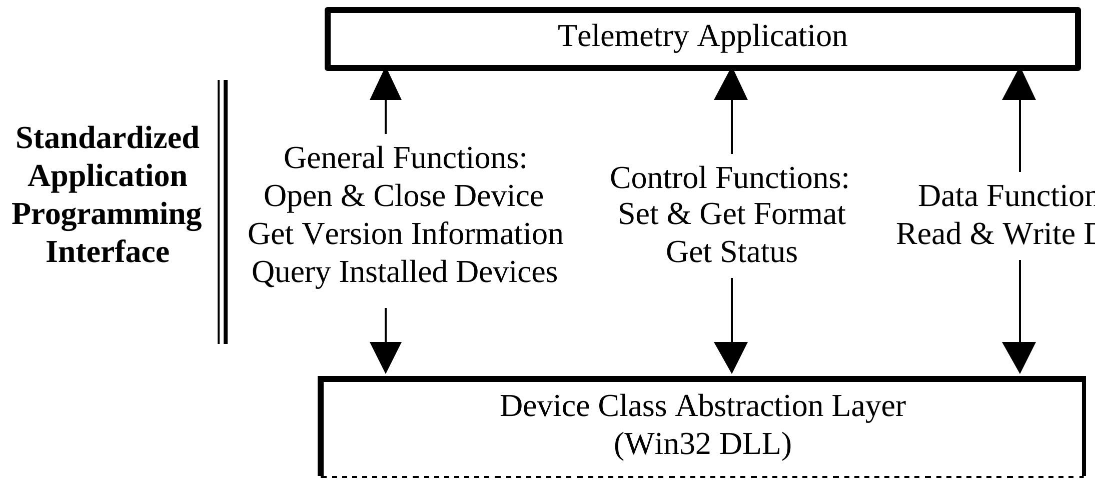

Figure 3 Abstraction Layer Upper Interface

LOWER INTERFACE

The lower interface of the device abstraction layer must accommodate the differences in the supported operating systems. By simplifying the lower level interface to a standard register based model, it becomes an easy task to develop device abstraction layers for different hardware products. All of the hardware specific register computations and formatting takes place in the abstraction layer. The lower interface simply provides a means of mapping an abstract device register to the proper place within the hardware. The hardware can be memory or I/O port based. The abstraction layer will always use the same register based interface. It is the responsibility of the hardware device driver to map the register based data to its proper destination.

The abstraction layer accommodates the different OS's by detecting the host OS at load time. This is a simple process which is facilitated by the Win32 DLL model. All of the subject OS's will dynamically load a desired DLL automatically the first time one of its exported functions is called. The loading process calls a unique function within the DLL and informs it of the load event details. At this time, the abstraction layer DLL can determine the host OS and modify its internal data structures appropriately.

The register based abstraction of the hardware minimizes the differences between the subject OS's. Limiting all interchanges between the abstraction layer and the device driver to the I/O Control interface reduces the OS specific functions to those identified in the following table.

OS Specific Functions	Differences
Loading and unloading the driver	The VxD driver model loads the driver once even if there is more than one hardware device. The NT driver model requires a load for each instance of the same hardware.
Device driver handles	VxD based I/O Control function calls must use the same driver handle for each instance of the same hardware. NT based I/O Control function calls must use the unique handle assigned to each instance of the hardware.

Due to the different implementations of device driver handles between the OS's, a technique was needed to abstract out this difference. All upper abstraction layer functions contain a physical unit number and all lower abstraction layer functions contain a logical unit number. The abstraction layer is responsible for maintaining the mapping between physical and logical units. Physical unit numbers are tied to a specific instance of the hardware. Logical unit numbers are assigned randomly by the OS to each instance of hardware. This logical assignment is implemented differently by the VxD and NT driver models and therefore, must be managed differently depending on the host OS.

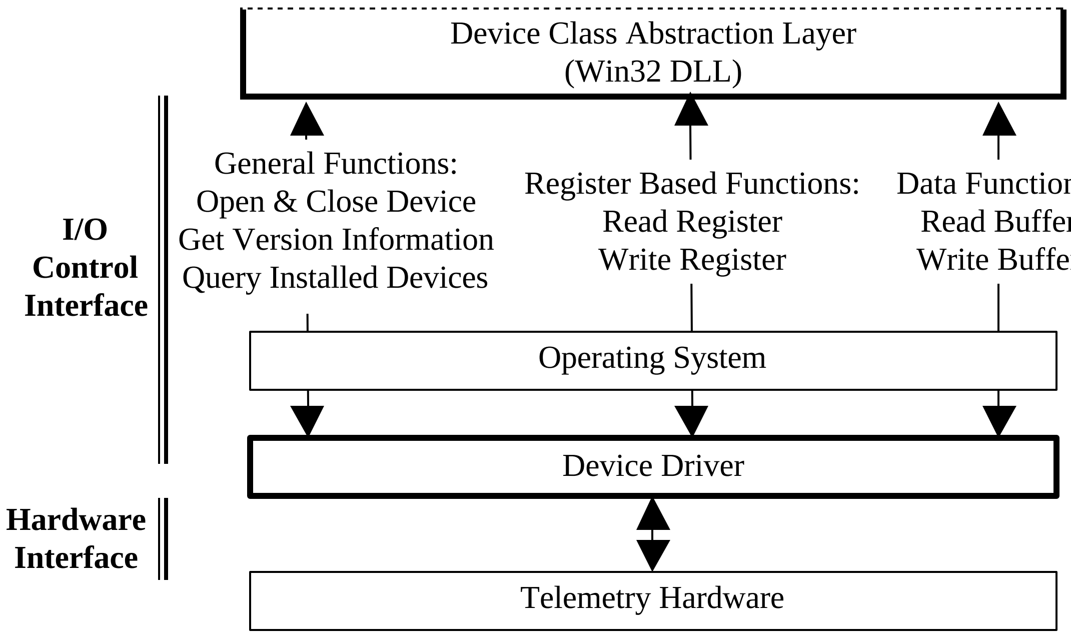

Figure 4 Abstraction Layer Lower Interface

DEVICE DRIVER

The device driver is the only component of the architecture that is completely OS dependent. Device driver development is very difficult and it was imperative to minimize the amount of application specific functionality contained at this level. By treating each telemetry device as an abstract, register based component, a simple interface can be used to accommodate multiple telemetry device types. The device drivers responsibilities include mapping register requests to or from their desired destination or source, managing data transfers and implementing the required OS specific functionality. Figure 5 presents additional details on the device driver to OS relationship.

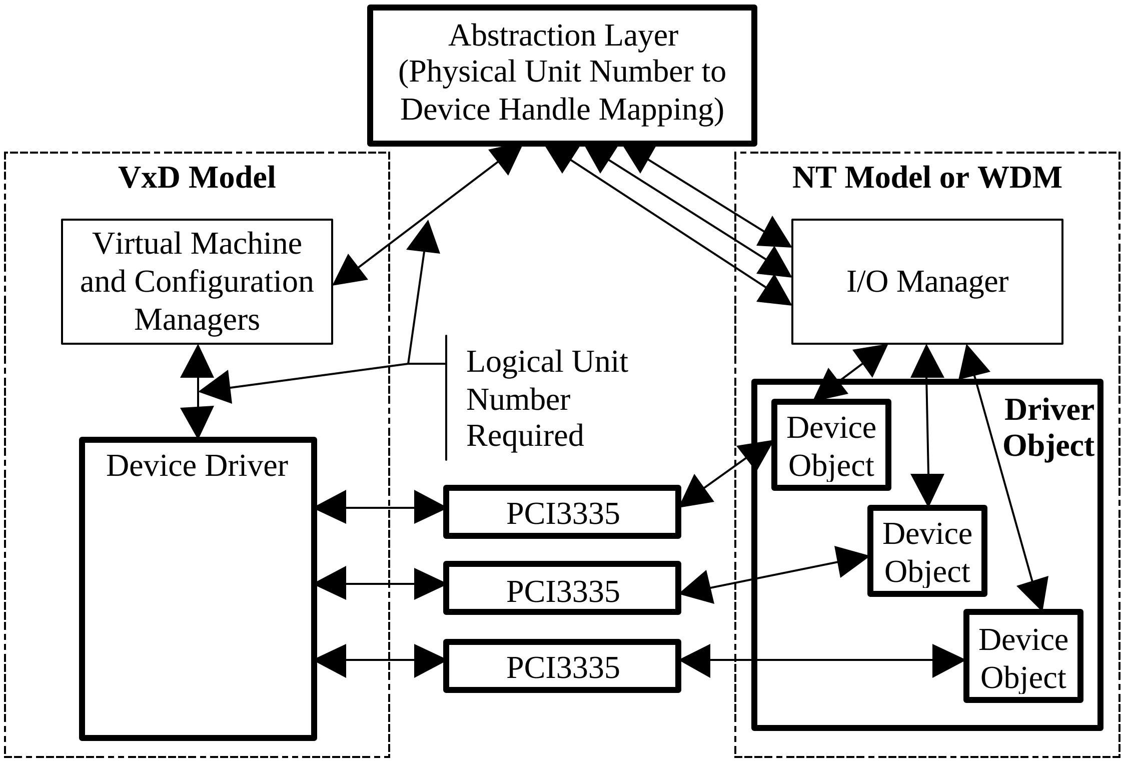
Abstraction Layer (Physical Unit Number to Device Handle Mapping)
VxD Model
NT Model or WDM
Virtual Machine and Configuration Managers
I/O Manager
Logical Unit Number Required
Driver Object
Device Driver
Device Object
Device Object
Device Object
PCI3335
PCI3335
PCI3335

Figure 5 Device Driver to OS Relationship

ABSTRACTION LAYER SUMMARY

An abstraction layer, implemented at an unrestricted level of the host OS, opens up many possibilities for product enhancement. Several functions were added to the abstraction layer of the PCI3335 to improve performance and to speed application development. This abstraction layer contains additional code to cache register contents at the lower interface level and format data structures at the upper interface level. Register caching prevents the abstraction layer from issuing I/O Control calls for register data which did not change. Format caching relieves the application from maintaining format data structures. These two abstraction layer features result in a performance gain.

Building hardware simulation capabilities into the abstraction layer generates solutions to several problems. During application development, actual hardware may not be readily available. By switching the abstraction layer into a simulation mode, application development can continue until final system level testing. Having simulation capability built into abstraction layers for each telemetry hardware device also provides a quick solution to developing a demonstration version of the telemetry application.

Figure 6 provides an example of a set format function as it travels from the application through the abstraction layer and on to the hardware device. A single application level call to PCI3335SetFormat () will cause the abstraction layer to cache the format data, compute a register set based on the new format information, determine which registers need to be updated, cache new register data and write the new register data via the device drivers I/O Control interface.

The abstraction layer allows the same application binary to function regardless of the host OS. The application and abstraction layer DLL's can be copied between different host computers and executed. The only differing component is the low level hardware device driver. For simulation capabilities, only the abstraction layer DLL is required. By switching the DLL in to simulation mode immediately after loading, no I/O Control calls are made to the device driver via the OS.

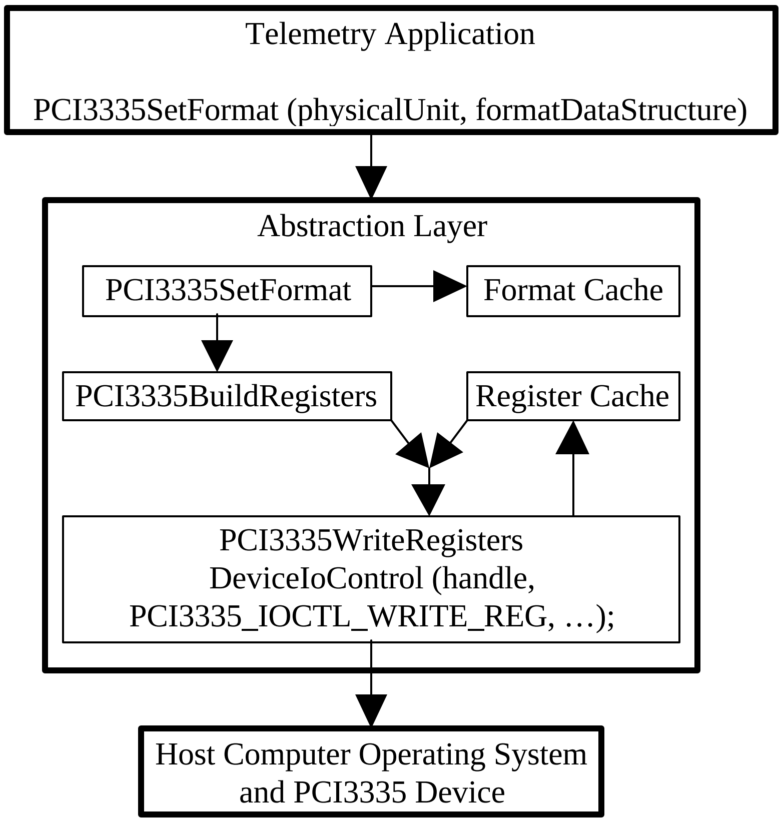

Figure 6 Abstraction Layer Example

CONCLUSION

The device driver architecture presented in this paper demonstrates techniques that simplify the development of telemetry applications which interface to custom hardware. The development of a high level abstraction of the telemetry hardware devices provides solutions to many different problems which face the application developer. Hardware simulation, demonstration and training versions of the product and efficient hardware upgrade or replacement paths are just a few of the problem areas which affect the software application. A carefully crafted, hardware device abstraction layer and a simple, operating system specific, device driver is all that is needed to improve application development time, performance and capabilities.

REFERENCES

Baker, Art, The Windows NT Device Driver Book: A Guide For Programmers, Prentice Hall, Upper Saddle River, New Jersey, 1997.

Oney, Walter, Systems Programming for Windows 95, Microsoft Press, Redmond, Washington, 1996.

NOMENCLATURE

API	Applications Programming Interface
DLL	Dynamically Linked Library
HAL	Hardware Abstraction Layer
OS	Operating System
PCI	Peripheral Component Interface
WDM	Microsoft Windows Driver Model
Win32	Windows API for 32 bit applications
Win95	Microsoft Windows 95
Win98	Microsoft Windows 98
WinNT	Microsoft Windows NT