Microsoft® Windows Driver Model (WDM)

By:
Mohamad (Hani) Atassi

Submitted to:
Dr. Dennis R. Hafemann
Winter Quarter
2001-2002
CS-384
Section 1

January 28, 2002

Abstract

Microsoft Windows Driver Model (WDM) is thoroughly covered in this paper. The paper starts out by defining the device driver and gives a comparison between the different kinds of device drivers that can be used under Windows systems. The discussion then leads to some detail information about the design and architecture of the WDM driver. Finally the paper ends on some techniques that are used in testing and debugging the driver before shipping it to the customer.

Table of Contents

Title Page

1. Introduction ... 1
 1.1. What is the device driver? ... 1
 1.2. Why Windows Driver Model (WDM)? ... 1
 1.3. WDM vs. NT style and the old VxD drivers ... 2
 1.4. An Overview of the operating systems ... 4
  1.4.1. Windows 98 overview ... 4
  1.4.2. Windows 2000 overview ... 5

2. A New Way of Thinking ... 8
 2.1. Device Driver Environment ... 8
 2.2. Win32 Program Interface ... 9

3. The Layering architecture of WDM device drivers ... 11
 3.1. Standard Bus and Class Drivers ... 13
 3.2. Example of Driver Stack ... 14

4. Device drivers design ... 17
 4.1. WDM Device driver components ... 17
  4.1.1. Driver Entry points and callbacks ... 18
  4.1.2. Dispatch Routines ... 18
  4.1.3. Calling other drivers ... 19
 4.2. Kernel mode programming ... 19
  4.2.1. Using Standard Run-Time Library functions ... 20
  4.2.2. Error Handling ... 20
 4.3. Memory management ... 21
 4.4. Synchronization ... 23
  4.4.1. Interrupt Request Level (IRQL) method ... 23
  4.4.2. Spin lock method ... 24

5. Power Management ... 25

6. Interrupt driven I/O ... 25

7. Testing and debugging techniques of the device driver ... 26

8. Conclusion ... 28

1. Introduction

1.1. What is the device driver?

In short words, the device driver is a software interface to hardware connected to a computer. It is a trusted part of the operating system. From the user application point of view, the device driver is the abstract connection to the device, and the user application should access the device without having to worry about how the hardware must be controlled.

In windows, a driver always makes a device looks like a file. So, the application program can be interact with the device very easily using read and write operations.

1.2. Why Windows Driver Model (WDM)?

In the past, during the ages of Windows 3.X and Windows 95, writing device drivers were restricted to VXD drivers. These kind of drivers are very hard to implement. The programmer should know everything about the hardware and how to share and use the bus.

Clearly, there are many pieces of hardware that are essentially alike, because they share bus or do similar tasks. WDM device drivers can use the facilities of these standard drivers. This approach makes it easier to share a common bus, and makes it much simpler to write new drivers.

In Windows 98, Windows 2000, and Windows XP, device drivers must be designed according to the WDM. Even though you can design VXD drivers for Windows 98, it is not recommended because of the compatibility issue with the succeeding Windows systems.

One of the important aspects of WDM is that Microsoft provides a series of system drivers that have all the basic functionality needed to service many standard types of devices. The first type of system driver supports different types of buses, such as the Universal Serial Bus (USB),

IEEE 1394 (Firewire) and Audio port devices. Other class drivers implement standard Windows facilities, such as Human Input Device (HID) and kernel streaming, etc.

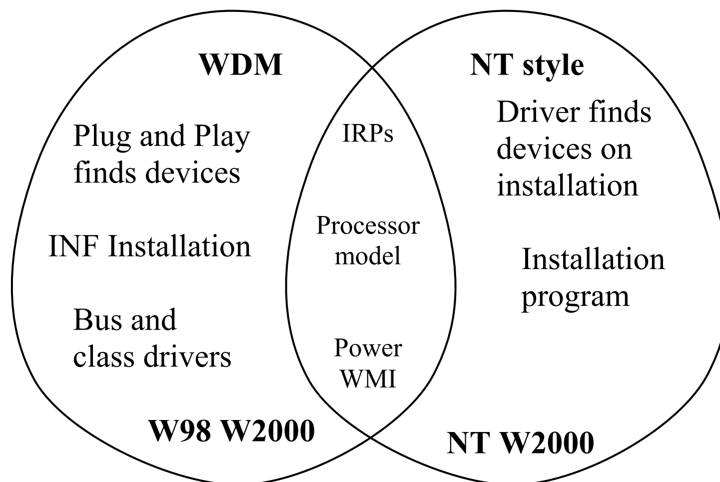
These system class drivers can make it significantly easier to write some types of device driver. For example, the USB system drivers handle all the low-level communications across the bus.

The next section compares different types of the device drivers that available in the Windows world.

1.3. WDM vs. NT style and the old VXD drivers

Indeed writing NT style drivers and WDM drivers is the same job. They are just different in how devices are created. NT style drivers does not support Plug and Play. They are called "NT Style" drivers because they run in NT 3.51, NT 4, Windows 2000, and Windows XP. Fig. 1.1 gives a rough indication of the differences between WDM and NT style drivers.

Figure 1.1 WDM and NT style device drivers



As stated in Fig. 1.1, Plug and Play technology finds the device and no need for extra coding as in NT style drivers. INF files are used to install WDM drivers, but you need a special installation program in a case of NT drivers. The new bus and class drivers are only available to WDM drivers. They both should support the Power Management and the Windows Management Instrumentation features.

The comparison between WDM drivers and VxD drivers is more involved because each one depends on the architecture of the operating system. One of the attributes that WDM drivers has, which VxD drivers are lacked to support, is the portability feature among many platforms. In a matter of fact, the WDM driver is a binary compatible between Windows 98 and Windows 2000 x86, and source code compatible to Windows 2000 Alpha platforms.

VxD drivers communicate directly with the hardware in Windows 3.1 and Windows 9X systems. But on the other hand in Windows 2000 (when I mention Windows 2000, I mean also Windows XP because the later is based on Windows 2000 and NT), it contains a special layer in the kernel called the Hardware Abstraction Layer (HAL). This layer includes platform-specific operations, and the device driver makes calls to it whenever it needs to communicate with the hardware. For example, a read operation might involve calling READ_PORT_UCHAR to read a single data byte from an I/O port. The HAL routine uses a platform-dependent method to actually perform the operation. On the Intel x86 computer, the HAL would use IN instruction; on an Alpha, it would perform a memory fetch.

Before I go any further it is worth looking at the involved operating systems that supports the WDM device drivers.

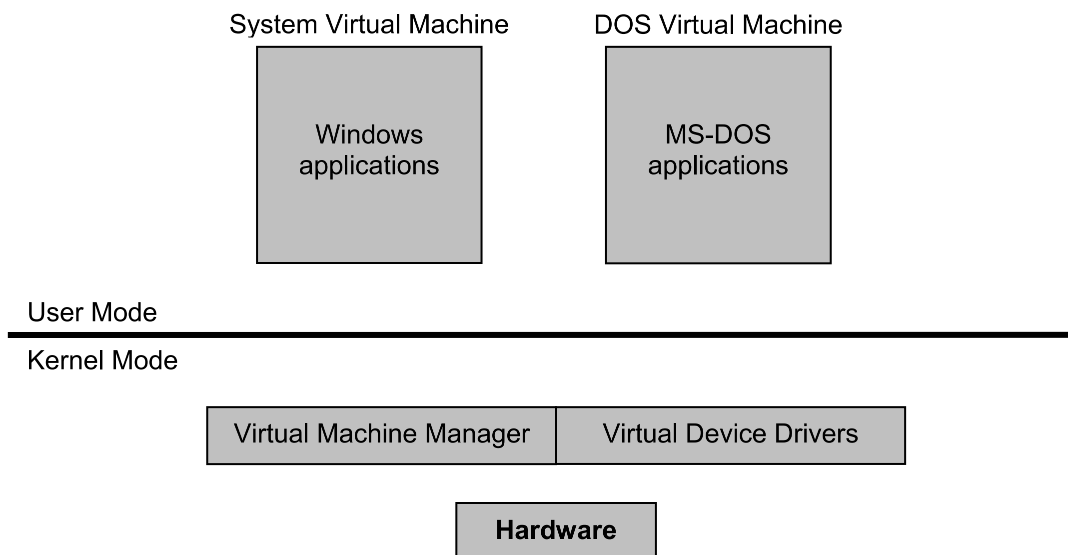
1.4. An Overview of the operating systems

We saw earlier that WDM provides a framework for device drivers that operate in Windows 98 and Windows 2000. Although to the end user these two operating systems are very similar, they work differently on the inside. In this section, I will give a brief overview about each of these operating systems.

1.4.1. Windows 98 overview

Figure 1.2 shows one way of thinking about windows 98.

Figure 1.2 The Windows 98 architecture



The main component of Windows 98 kernel is the Virtual Machine Manager (VMM). To be more precisely, Windows 98 kernel is called the Virtual Machine Manager because it virtualizes the hardware resources of the machine to give each virtual machine the appearance that it has

exclusive control of the system. The VMM also provides virtual memory and includes a scheduler that allocates time between the various virtual machines.

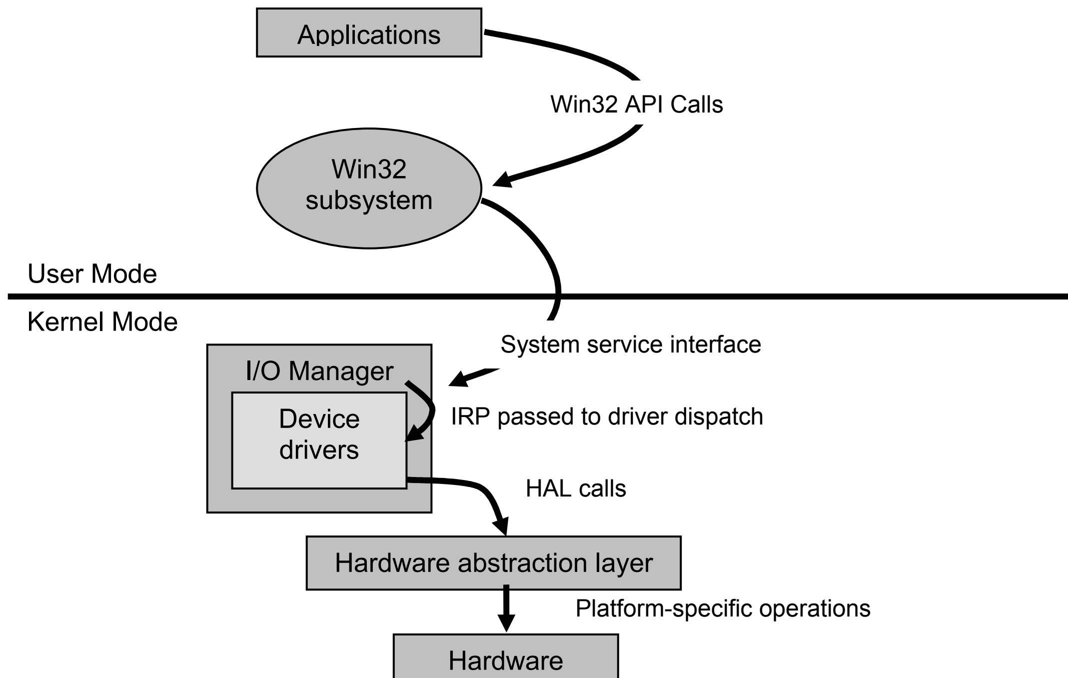
The Virtual Machine Manager consists of a core set of services, plus numerous modules that are loaded when windows starts. These additional modules are known as Virtual Devise Drivers, or VxDs. Because VxDs are loaded as part of the lower layer of the operating system, they operate with more privileges than other applications. None of the normal limits, restrictions, or protection mechanisms apply to Virtual Device Drivers.

When a 32-bit application in Windows 98 needs to read some data from a device, it calls a Win32 API such as **ReadFile**, which a system DLL like KERNEL32.DLL services. But applications can only use ReadFile for reading disk files, communication ports, and devices that have WDM drivers. For any other devices, an application must use some mechanism based on **DeviceIOControl**. Here in Windows 98, most of the implementation of ReadFile resides in the user-mode. The use-mode implementation of ReadFile validates parameters and uses one or another special mechanism to reach a kernel-mode driver. The transition is made by the software interrupt 30h from user mode to kernel mode. Then , in kernel mode the code executed differs between the requested devices.

1.4.2. Windows 2000 overview

Figure 1.3 shows a part of Windows 2000 architecture that emphasizes the features that are important to people who writes device drivers.

Figure 1.3 The Windows 2000 architecture



Again here when an application in user-mode wants to, say, read some data from a device, it would call an API such as ReadFile. A subsystem module such as KERNEL32.DLL implements this API by invoking some sort of platform-dependent system service interface to reach a kernel-mode support routine. In the case of a call to ReadFile, it uses the system service interface to reach a kernel-mode routine that is called **NtReadFile**. NtReadFile is part of a system component that is called the I/O Manager.

Many routines serve a purpose similar to NtReadFile. They operate in kernel mode to service an application's request to interact with a device in some way. They all validate their parameters –the same operation happened in user-mode in Windows 98-. They then create a data structure called I/O request packet (IRP) that they pass to an entry point in some device driver. In the case of an original ReadFile, NtReadFile would create an IRP with a major function code of IRP_MJ_READ. This mechanism is similar to a message passing mechanism in Windows in that

the IRP structure is the Message and the device driver entry point is the Window Procedure. After the device driver takes the execution, it might eventually need to actually access its hardware to perform the passed IRP. In the case of IRP_MJ_READ to a programmed I/O (PIO) sort of devices, the access might take the form of a read operation directed to an I/O port or a memory register implemented by the device. Even though drivers are executed in kernel mode and can talk to there hardware directly, they should use the facilities provided by the Hardware Abstraction Layer (HAL) to access hardware. While the driver is executing the request, the user-mode program can either continue its business or wait for the operation to finish. So, when the driver completes the IRP, it calls a kernel-mode service routine. This call allows the waiting application to resume execution.

NOTE:

Windows 98 should call WDM drivers the same way Windows 2000 calls it. So, when it comes to WDM drivers, the interior architecture of Windows 98 is necessarily similar to that of Windows 2000. A system module (NTKERN.VXD) contains Windows-specific implementations of many Microsoft Windows NT kernel support functions. NTKERN.VXD creates IRPs and sends them to WDM drivers in just about the same way as Windows 2000. WDM drivers almost cannot tell in fact the difference between the two environments.

2. A New Way of Thinking

Writing device drivers is much different than the regular Win32 programming. There are no windows and messages to manipulate. You do not have those protective arms of Windows to stop you from destroying other processes and the operating system. Source level debugging is very difficult to set up. Most of the libraries you used to have are not available – not even the C Standard library and the C++ new operator. Moreover, you have to use makefiles to build drivers, though it is easy to control the build process from a development environment tool such as Visual Studio.

As a device driver is a trusted part of the operating system you can crash the system very easily producing a lot of blue screens! Therefore, device drivers should be programmed very carefully and commented very well. Checking for error return values from every kernel call is a necessity, too.

2.1. Device Driver Environment

Windows 2000 and Windows 98 are multitasking operating systems that share the CPU among an arbitrary number of threads. Much of the time, driver subroutines execute in an environment in which they can be preempted to allow another thread to execute on the same CPU. More than one user application may be bombarding a driver with requests. A user program with open I/O requests may terminate suddenly. A driver may be running in a multiprocessor PC. Windows 2000 uses a symmetric multiprocessor model, in which all the CPUs are considered equal. So, a driver can be running with a different pieces of it on different CPUs, and all CPUs have equal access to memory. The existence of multiple CPUs poses a difficult synchronization problem for device drivers because code executing on two or more CPUs might simultaneously need to

access shared data or shared hardware resources. The Windows 2000 kernel provides a synchronization object called a spin lock that drivers can use to avoid destructive interference in such situations. (See Section 4.4.)

Supporting configurable and hot-pluggable busses also adds to the burden of device driver writers. For example, a Plug and Play device might be removed by the user any time. Also, the kernel might decide at any time that it needs to stop your device so that it can reassign all the hardware resources.

However, as mentioned previously, using the standard WDM bus and class drivers helps to reduce the amount of effort required to write drivers for certain categories of devices.

2.2. Win32 Program Interface

It is worth now to look at how Win32 programs call a device driver. As I said before that Windows treated the device as if it were a file. So, for Win32 programmers, they can use the same API functions used with files as listed in Table 2.1.

Table 2.1 Win32 Program Interface

Win32 function	Action
CreateFile	Open device file
CloseHandle	Close device file
ReadFile ReadFileEx ReadFileScatter ReadFileVlm	Read
WriteFile WriteFileEx WriteFileGather WriteFileVlm	Write
DeviceIoControl	IOCTL
CanelIo	Cancel any overlapped operations
FlushFileBuffers	

A device driver provides one or more named devices to the Win32 programmer. For example, if writing a device driver for a scanner that is attached to any parallel port, the devices might be named “ScannerLpt1”, “ScannerLpt2”, etc. To the win32 programmer, these appear as \\.\ScannerLpt1, etc.

CreateFile function is used to create or open a connection to a device. You also can specify the read and write access mode and whether the file can be shared. CloseHandle is used to close the device file when you have finished using the device file.

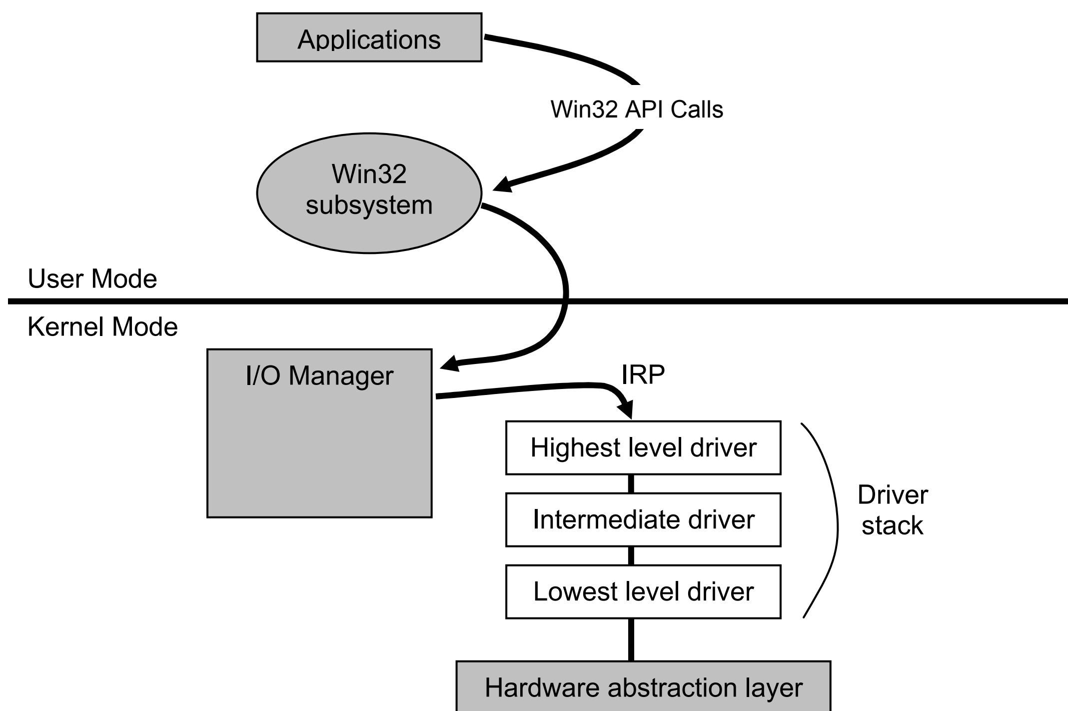
The ReadFile and WriteFile functions are used to issue read or write requests to the device.

DeviceIoControl is used to issue read or write requests that can send data to a driver and read data back, all in one call. There are many predefined IOCTL codes that can be sent to standard drivers, but you can also make up your own for your driver.

3. The Layering architecture of WDM device drivers

WDM device drivers are broken down into many parts or small drivers that are all connected together. The lower level is always the level which communicates directly with the hardware or the Hardware Abstraction Layer. This layering architecture is called a driver stack. When the I/O manager creates the IRP because of a request from a user application, it sends it to the destination driver stack. Then, each layer is responsible to process the request if necessary and pass it to the next layer in the stack. Fig. 3.1 demonstrates how this abstraction works.

Figure 3.1 The Driver Stack



Each driver in the stack breaks down the request into simpler request. Highest level drivers, such as file system drivers, know how files are represented on disk, but not the details of how to get the data. Intermediate level drivers process requests further by breaking down a large request into a series of small chunks.

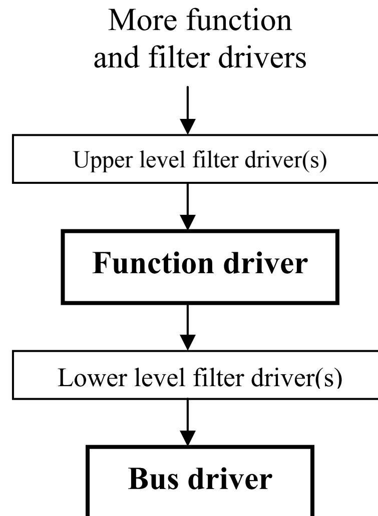
In Windows Driver Model, each hardware devices has at least two device drivers. One of these drivers, which is called the function driver, is what you've always thought of as being the device driver. It understands all the details about how to make the hardware work.

The other of the two drivers that every device has is the bus driver. It is responsible for managing the connection between the hardware and the software. For example, the bus driver for the PCI bus is the software component that actually detects that your card is plugged in to a PCI slot and determines what requirements your card has for I/O-mapped or memory-mapped connection with the host.

The function driver that does know how to control a device's main features is layering above the bus driver. Also, it is possible for more function drivers to be layered on top of the first function driver. An example of this is given shortly.

Some devices has more than two drivers. These extra drivers are called filter drivers. Some of these filter drivers simply watch as the function driver performs I/O. More often, a filter driver is added to modify the behavior of an existing function driver in some way. Upper filter drivers see IRPs before the function driver because it's layered above it. Sometimes an upper filter can perform a workaround for a bug in the function driver or the hardware. Lower filter drivers see IRPs that the function driver is trying to send to the bus driver. This schema of layering and processing requests is shown in Fig. 3.2.

Figure 3.2 Generic WDM driver stack



A WDM function driver is often composed of two separate modules. One file, the class driver, understands how to handle all the WDM protocols that the operating system uses. The other file, called the minidriver, interfaces with the hardware or another class driver. For example, the Human Input Device (HID) class driver uses minidrivers to talk to hardware. One minidriver is provided to talk to HID devices on the USB bus. This HID minidriver translates requests from the HID class driver into requests to the USB bus driver.

3.1. Standard Bus and Class Drivers

Table 3.1 shows the main class and bus drivers that are provided with Windows. These are general purpose drivers that can act as bus drivers and as function drivers.

Table 3.1 Bus and Class drivers

Bus drivers	Description
Advanced configuration and Power Interface (ACPI)	Interacts with the ACPI BIOS to enumerate the devices in the system and control their power use,
Peripheral Component Interconnect (PCI)	Enumerates and configures devices on the PCI
PnPISA	Enumerates and configures devices on ISA devices that are configurable using Plug and Play.
IEEE 1394	Enumerates and controls the IEEE 1394 (Firewire) high speed bus.
Universal Serial Bus (USB)	Enumerates and controls the lower-speed USB bus.

Class Drivers	Description
Stream	Provides a basic for processing high bandwidth, time critical, and video and audio data. It uses minidrivers to interface to actual hardware.
SCSI and CDROM/DVD	Are used to access hard disks, floppies, CDs, and DVDs
Human Input Device (HID)	Provides an abstract view of input devices.
Still Image Architecture	Used to obtain scanner and still image camera using minidrivers

As usual, a variety of minidrivers are used to talk to hardware. For example, requests to a CD-ROM on the IEEE 1394 bus would be routed to the IEEE 1394 bus driver. Some of HID devices are on the USB bus, so a HID minidriver for the USB bus is provided as standard.

3.2. Example of Driver Stack

After looking for the different kinds of Class and Bus drivers, it is worth now to take a look at one real example to see how it all fits together.

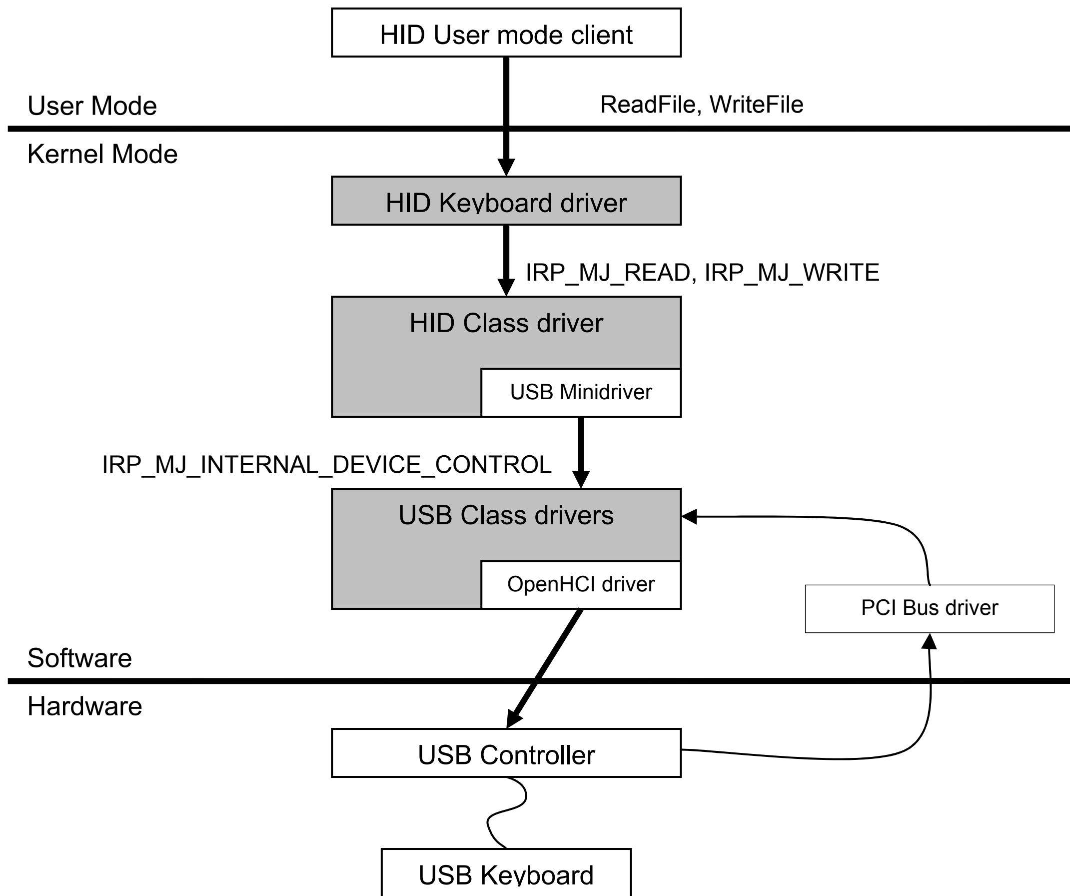
Figure 3.3 shows the hardware and software components that are involved in the handling of a HID keyboard attached to the USB bus. The next points describes briefly what happened when Windows starts.

- The PCI bus driver finds the USB controller and so loads the USB class drivers. In this case the USB OpenHCI host controller driver is used to talk directly to the hardware.
- The USB host controller detects when the keyboard is plugged in.

- The USB bus driver tells the plug and play manager that a new device has arrived.
- The keyboard hardware ID is used to find the correct installation INF file. In this case the HID class driver and its USB minidriver are loader, along with the standard HID keyboard client driver.
- The HID keyboard driver calls the HID class driver with Read IRPs to request keyboard data.
- The HID class driver asks the USB minidriver to retrieve any HID input reports.
- The minidriver uses an internal IOCTL to submit the USB Request Blocks (URBs) to the USB class driver to read data.
- The USB class driver talks directly to the host controller to get any available data.
- These data are interpreted by the USB keyboard driver and the result eventually reaches a Win32 program in message form.

Figure 3.3 also shows that a user mode HID client can access the HID keyboard using standard Win32 ReadFile and WriteFile requests.

Figure 3.3 HID USB Keyboard example



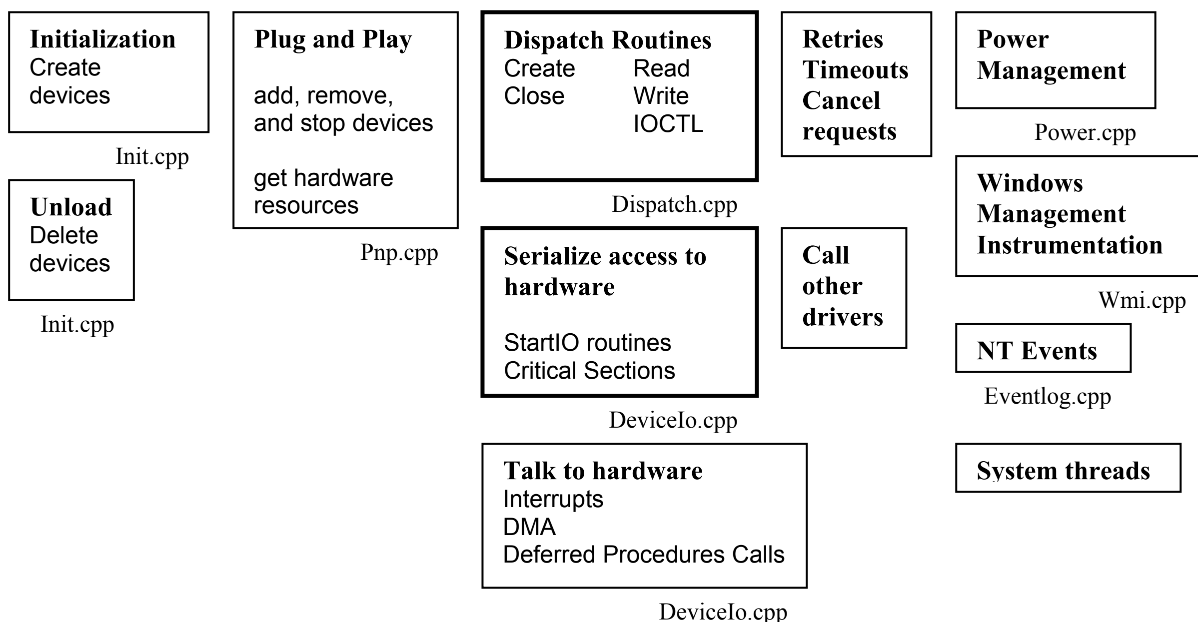
4. Device drivers design

A device driver has many jobs to do, some of them are mandatory and others are optional. In this section, I will give an overview of some of these protocols. I will also show some basic programming techniques in the kernel mode and continue with some design issues.

4.1. WDM Device driver components

Figure 4.1 shows a device driver divided into different modules. The figure also shows an optional naming convention for the modules. Strictly speaking, only the initialization module is mandatory. In practice, all device drivers have dispatch routines to handle user I/O requests. All the other modules are optional, although it is better to write minimal Power Management and Windows Management Instrumentation modules, simply to pass requests to the lower drivers in the stack.

Figure 4.1 Device Driver Component



4.1.1. Driver Entry points and callbacks

A driver has one main initialization entry point. This routine must be called **DriverEntry**. The kernel calls your DriverEntry routine when it is loaded. Subsequently, the kernel may call other routines in the driver. These routines are called callbacks. Basically, in DriverEntry, you register all the callback routines that you need to receive from the kernel. For example, if you want to handle interrupts, you must tell the kernel the name of your Interrupt Service Routine (ISR) callback. Also, two of the important callback routines are I/O Request Packet (IRP) handlers to process IRPs you wish to handle, and **StartIO** callback routine to handle IRPs serially.

4.1.2. Dispatch Routines

The DriverEntry routine, as we saw, must set up a series of callbacks for processing IRPs and StartIO callback routine. For example, the driver might need to process the Create IRP, Close IRP and Read IRP that are generated from CreateFile, CloseHandle and ReadFile Win32 calls from the user application.

The main responsibility to the IRP processing callback routine is to check the parameters of the IRP and then dispatch the IRP for processing elsewhere in the driver. Quite often, IRPs need to be processed serially so that the driver interacts with the hardware in a safe mode. To be more precisely, the IRP processing callback routine sends the IRP to the driver’s queue. The kernel I/O manager then takes the IRPs from out the queue one by one and passes them to the driver’s StartIO callback routine, which has been already set up in the DriverEntry routine.

4.1.3. Calling other drivers

WDM drivers spend a lot of time talking to other drivers. As we saw earlier that WDM drivers are stacked in above of each other. Each layer or stack entry provides one step higher of abstraction. Some types of IPRs, such as Plug and Play, Power Management, and Windows Management Instrumentation IRPs, are often passed immediately to the next device. Only minimal processing is required by the driver.

4.2. Kernel mode programming

As I said earlier that programming in kernel mode is totally a different story for programmers. They should learn new service functions that would help them through out the developing process. Moreover, the C and C++ Standard libraries are not there in the kernel.

Table 4.1 illustrates some of the important components that make up the Microsoft Windows NT operating system. Each component exports service functions whose names begin with a particular two-letter prefix. Those functions are available to use in Windows 98 for WDM drivers because of the system module (NTKERN.VXD) that includes most of NT specific functions.

Table 4.1 Overview of kernel-mode support routines

Windows NT Component	**Description**
I/O Manager (**Io**)	Contains many service functions that drivers use
Process Structure Module (**Ps**)	Creates and manages kernel mode threads
Memory Manager (**Mm**)	Controls the page tables that define the mapping of virtual addresses
Executive (**Ex**)	Supplies heap management and synchronization services
Object Manager (**Ob**)	Provides centralized control over the many data objects with Windows NT works
Security Reference Monitor (**Se**)	Allows file system drivers to perform security checks.
Run-time library component (**Rrl**)	Contains utility routines, such as list and string management routines.
User-mode calls (**Zw**)	Can be used to access files and the registry from the kernel mode.
Windows NT Kernel (**Ke**)	Where all the low-level synchronization of activities between threads amd processors occurs.
Hardware Abstraction Layer (Hal)	Is used to talk to the hardware instead of a direct access

4.2.1. Using Standard Run-Time Library functions

The reason behind not using the standard C runtime library in the kernel-mode is related to some historical events. The kernel of Windows NT was designed at a time when was no ANSI standard for the functions that belonged to the standard library. The other reason is that some standard library functions rely on initializations that only happened in user-mode.

Officially, the kernel-mode driver programmer should only call functions specifically documented in DDK. For example, call **RtlCompareUnicodeString** instead of calling **wscmp**, and **RtlCopyBytes** instead of **memcpy**. The functions are all prefixed with (Rtl), and I advise you to look at the DDK reference help to master them.

4.2.2. Error Handling

All of the kernel functions return a status code of the type **NTSTATUS**. This status code contains information about the error, and the most significant bit of it indicates whether the

function succeeded or failed. You should always use a macro like **NT_STATUS** to check the success of the function. This technique is the same that is used in COM error handling.

You can also use Structured Exception Handling in your code that helps you avoid potential system crashes. In Microsoft compiler, you should use the Microsoft extensions to the C/C++ language. In particular, you can use __try, __finally, and __except statements (all with two preceding underscores). C++ try/catch exceptions won't work in the kernel-mode because they depend on the C++ standard run-time library, which is not available to use. For raising exceptions, you should use **ExRaiseStatus**, **ExRaiseAccessViolation**, or **ExRaiseDatatypeMisalignment**. The rule for those functions is that you should only raise an exception when you know there is an exception handler in the function above you.

4.3. Memory management

Windows NT and Windows 98 run on computers that support a virtual address space. This virtual address can be mapped either to physical memory or to page frames within a swap file on disk. Generally speaking, Windows NT divides the available virtual address into kernel-mode addresses and user-mode addresses. Each user-mode process has its own address context, which maps the user-mode virtual addresses to a unique collection of physical page frames. Windows NT scheduler switches from a thread in one process to a thread in another process. So, the user-mode virtual address changes from one moment to the next.

It's generally unlikely that a WDM driver will execute in the same thread context as the initiator of the I/O requests it handles. In other words, we don't know for sure to which process the current user-mode address context belongs. So, don't take an address that a user mode

application provides and treat that address as a pointer that we can directly dereference. Actually, there are some techniques for accessing data buffers that originate in user mode.

Windows NT divides the kernel-mode address space into paged and nonpaged memory pools. The user-mode address space is always pagable. Things that must reside in memory are in the nonpaged pool, and things that can come and go on demand are in the paged pool

Any module in Windows, including exe and dll, is made up of many sections. When loading a driver image, the system puts sections whose literal names begin with “page” or “.eda” into the paged pool. The other sections are in the nonpaged pool by default. Using some **pragma** directives in the C/C++ compiler we can force some area of code or data to reside in a section that its name starts with “page” so we make it pagable. The next example shows how to create a pagable code and data sections using the Microsoft C++ compiler.

```
#pragma data_seg(“PAGE”)   // Start a pagable section
int g_nPageMe;
#pragma data_seg()         // Return to the default nonpaged pool

#pragma code_seg(“PAGE”)   // Start a pagable section
NSTATUS AddDevice(...) {...}
#pragma code_seg()         // Return to the default nonpaged pool
```

Run-time control of pagability is also available under Windows NT. For example to lock a section of code, you can use **MmLockPagableCodeSection** function. For data, you use **MmLockPagableDataSection**. To unlock the previously locked sections using the previous functions, you use **MmUnlockPagableImageSection** function.

Memory management is very important if the device uses an interrupt. Interrupt service routine (ISR) must be located in the physical memory when the interrupt happens or the system

crashes. So, you must lock the section that includes the ISR or just create the section as nonpagable. The rule stated in the DDK is that you simply mustn't page your ISR.

To allocate area in the heap you mustn't call C++ new operator because it's basically not available. The basic heap allocation service function in kernel mode is **ExAllocatePool**. You specify using the parameters of this function the type of memory area whether or not it can be paged. To free the previously allocated heap, use **ExFreePool**.

4.4. Synchronization

Windows 2000 is a multitasking operating system that can run in a symmetric multiprocessor environment. Because of that we would face many problems while running our code in a system like this and they are:

- The operating system can preempt any subroutine at any moment for an arbitrarily long period of time.
- Code executing simultaneously on another CPU in the same computer can interfere with our code. It's even possible that the exact same set of instructions belonging to some subroutine could be executing in parallel in the context of two different threads.

Windows allow you to solve these general synchronization problems by using interrupt request level (IRQL) and by claiming and releasing spin locks around critical code sections.

4.4.1. Interrupt Request Level (IRQL) method

This method is very useful to avoid destructive preemption on a single CPU. Windows NT assigns a priority level known as the interrupt request level to each hardware interrupt and to selected few software events.

The lowest level is the PASSIVE_LEVEL (0), where the highest level is HIGH_LEVEL (31). Between these two levels are many levels related to many hardware interrupts and events, such as clock levels and the dispatcher level.

The way that IRQL becomes useful is because once a CPU is executing at an IRQL above PASSIVE_LEVEL, an activity on that CPU can be preempted only by an activity that executes at a higher IRQL. User-mode applications execute in PASSIVE_LEVEL and are therefore preemptable by any activity that executes at a higher IRQL priority.

4.4.2. Spin lock method

Since IRQL is a per-CPU concept, it doesn't help to safe data against interference by code running on another processor in the same multiprocessor computer. Spin lock serves that purpose. To acquire a spin lock, code on one CPU executes an atomic operation that tests and sets some memory variable. If the test indicates that the lock variable was previously free, the program continuous. If the test indicates that the lock was previously busy, the program repeats the test-and-set in a loop (spins). Eventually, the owner releases the lock by resetting the variable, and one of the waiting CPUs' test-and-set operations will report the lock as free.

The functions that are used with spin lock are: **KeInitializeSpinLock** to initialize the spin lock object, **KeAquireSpinLock** to do the test-and-set operation, and **KeReleaseSpinLock** to release the spin lock object. (The spin lock object must be in nonpageable memory area).

5. Power Management

The driver should support Power Management in Windows 98 and Windows 2000 if a device's power consumption can be controlled. This applies to both WDM and NT style drivers. Power Management conserves battery power in portables and reduces energy consumption.

The power manager can request that the whole system powers down. Actually, there are six system power states, including fully on and off, with three sleeping and one hibernating state in between. At a device level, there are four device power states, with two sleeping states in between fully on and off. A device can power it down even if the rest of the system is running at full speed.

Drivers support power management by handling Power IRP. Quite a few drivers will just pass the Power IRP down the stack of devices.

6. Interrupt driven I/O

Interrupts are usually used when something important has happened. For example, the printer might interrupts the PC if it ran out of paper, so the driver might need to inform the user to buy some more.

In x86 processor, there is a NonMaskable Interrupt (NMI) pin input and an Interrupt (INTR) pin input. All the devices share the INTR pin by means of an external 8259A controller. This controller is used to provide several interrupt lines to devices IRQ0 to IRQ15 and share them with the INTR pin. Also, the controller provides a different vector for each IRQ number. This number is mapped in the operation system kernel to some interrupt service routine.

If an INTR interrupt occurs, its service routine will run until completion, stopping other INTR interrupt from occurring. However, NMI can still occur because it has a higher priority than INTR interrupts.

As described previously, Windows provides the initial interrupt handler for IRQ0-IRQ15. If your device has connected to one of these interrupts, it calls your interrupt handling routine to service the interrupt. In fact, more than one device can share the same IRQ, so your first job is to determine if it really was your device that interrupted the processor.

7. Testing and debugging techniques of the device driver

Testing the driver basically means that you write a test program that exercises all the functions in the driver. The next points are very important when it comes to testing the driver:

- Check that all sorts of data work as expected. For example, check that invalid parameters are detected properly. Check boundary conditions. For example, if you have a maximum request size of 1024 bytes, check that transfers of 0, 1, 1023, and 1024 bytes succeed and that transfers of 1025 bytes fail.
- Check that all the features of the hardware are exercised.
- Check that a driver works in stress situations. Test with other processors doing the same job simultaneously. A good test is to run the driver while copying a large file to a floppy disk; in Windows 98, this seems to stop a most other activities.
- Check that a driver works when more that one of its devices are in use.
- Check that several user programs or threads can access the driver simultaneously.
- Check that the driver works on a clean PC that has never seen your driver. Test on plain Windows 98 and Windows 2000 systems, without any of the developing environment.

Problems with a driver can be a real pain; any thing goes bad with a driver means crashes and blue screens. The two most common causes of fatal errors are: Accessing nonexistent memory, and accessing paged memory at or above DESPATCH_LEVEL IRQL. So, you have to inspect each call or section of code carefully.

In Windows NT or Windows 2000, you can get it to produce core dumps in a file on disk, called memory.dmp by default. This option is better to be enabled from the Control Panel System applet Advanced tab Startup and Recovery section.

Some of the debugging techniques that are useful to use are:

- Use an incremental development. This means get each stage working and tested before moving onto the next stage.
- Always generate a checked version of your driver. The checked version is the unoptimized test debug build. When you are done with the driver you might produce the optimized retail release version, which is called the free build.
- While debugging the driver it's better to debug in a checked build of Windows. This version of windows includes the entire symbol debugging information for the kernel modules and might come in handy some times.

The most powerful tool is a debugger. Standard user-mode debuggers like Visual Studio do not work within drivers. The Microsoft DDKs come with WinDbg debugger for Windows NT and Windows 2000 systems. To use this you must have two PCs – one running a checked build of NT/2000 and another running the free build – connected together using a serial cable.

8. Conclusion

Developing device drivers is not an easy thing to do and it involves many repetitive coding. You would find your self spending a lot of time just designing the architecture of your driver in order to make it compatible with the I/O Manager. To master WDM developing, you should refer back to the Microsoft DDK, which includes many useful information and examples about developing device drivers to many different kind of devices.

References

1. MSDN Library and DDK (Device driver kit)

2. NuMega Driver Suite v1.01

3. Programming the Microsoft Windows Driver Model, Walter Oney, R&D Books.

4. Writing Windows WDM Device Drivers, Chris Cant, Microsoft Press.