O N E

Introduction to Windows 2000 Drivers

CHAPTER OBJECTIVES

Device drivers on any operating system necessarily interact intimately with the underlying system code. This is especially true for Windows 2000. Before jumping into the world of Windows 2000 device drivers, this chapter presents the design philosophy and overall architecture of Windows 2000.

Overall System Architecture

Windows 2000 presents arguably the most aggressive attempt at operating system control in the history of computers. This section tours the Windows 2000 architecture, highlighting the features of significant interest to a device driver author.

Design Goals for Windows 2000

The original goals for Microsoft's NT ("New Technology") operating system took form in early 1989. Interestingly, the original concept for NT did not

even include a Windows operating environment. While the NT OS has indeed come a long way since 1989, the five fundamental goals remain intact.

- **Compatibility.** The OS should support as much existing software and hardware as possible.
- **Robustness and reliability.** The OS should be resilient to inadvertent or intentional misuse. A user's application should not be able to crash the system.
- **Portability.** The OS should run on as many present and future platforms as possible.
- **Extendibility.** Since the market requirements will change (grow) over time, the OS must make it easy to add new features and support new hardware with minimal impact on existing code.
- **Performance.** The OS should provide good performance for a given capability of the hardware platform which hosts it.

Of course, goals are not reality, and over time, serious compromise of one goal may be necessary to achieve another. NT is an operating system and, as such, is subject to the same sorts of compromises that affect all systems. The remainder of this section describes the delicate balance of solutions that Microsoft OS designers chose to implement their goals.

Hardware Privilege Levels in Windows 2000

To achieve the robustness and reliability goal, the designers of NT chose a *client-server architecture* for its core implementation. A user application runs as a client of OS services.

The user application runs in a special mode of the hardware known generically as *user mode*. Within this mode, code is restricted to nonharmful operations. For example, through the magic of virtual memory mapping, code cannot touch the memory of other applications (except by mutual agreement with another application). Hardware I/O instructions cannot be executed. Indeed, an entire class of CPU instructions (designated *privileged*), such as a CPU Halt, cannot be executed. Should the application require the use of any of these prohibited operations, it must make a request of the operating system kernel. A hardware-provided *trap* mechanism is used to make these requests.

Operating system code runs in a mode of the hardware known as *kernel mode*. Kernel-mode code can perform any valid CPU instruction, notably including I/O operations. Memory from any application is exposed to kernel-mode code, providing, of course, that the application memory has not been *paged out* to disk.

All modern processors implement some form of *privileged* vs. *nonprivileged* modes. Kernel-mode code executes in this privileged context, while user-mode code executes in the nonprivileged environment. Since different processors and platforms implement privileged modes differently, and to help achieve the goal of portability, the OS designers provided an abstraction for user

and kernel modes. OS code always uses the abstraction to perform privileged context switches, and thus only the abstraction code itself need be ported to a new platform. On an Intel platform, user mode is implemented using Ring 3 of the instruction set, while kernel mode is implemented using Ring 0.

This discussion is relevant to device driver writers in that kernel-mode drivers execute in a privileged context. As such, poorly written device driver code can and does compromise the integrity of the Windows 2000 operating system. Driver writers must take extra care in handling all boundary conditions to ensure that the code does not bring down the entire OS.

Portability

To achieve the portability goal, NT designers chose a layered architecture for the software, as shown in Figure 1.1.

The Hardware Abstraction Layer (HAL) isolates processor and platform dependencies from the OS and device driver code. In general, when device driver code is ported to a new platform, only a recompile is necessary. How can this work since device driver code is inherently device-, processor-, and platform-specific? Clearly, the device driver code must rely on code (macros) within the HAL to reference hardware registers and buses. In some cases, the device driver code must rely on abstraction code provided in the I/O Manager (and elsewhere) to manipulate shared hardware resources (e.g., DMA channels). Subsequent chapters in this book will explain the proper use of the HAL and other OS services so that device driver code can be platform-independent.

Extendibility

Figure 1.1 also shows an important design concept of Windows 2000—the kernel is separate from a layer known as the *Executive.*

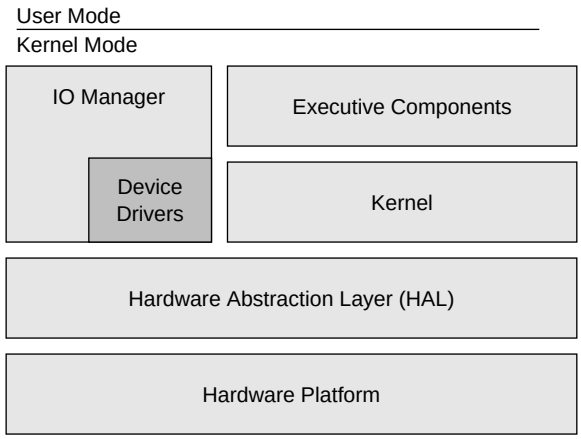

Figure 1.1 *The layers of the Windows 2000 operating system*

The Windows 2000 kernel is primarily responsible for the scheduling of all thread activity. A thread is simply an independent path of execution through code. To remain independent of other thread activity, a unique thread *context* must be preserved for each thread. The thread context consists of the CPU register state (including a separate stack and Program Counter), an ID (sometimes called a Thread ID or TID, internally known as a Client ID), a priority value, storage locations local to the thread (Thread Local Storage), and other thread-relevant information.

The scheduler's responsibility is to manage which thread should execute at any given time. In a single processor environment, of course, only one thread may actually gain control of the processor at a time. In a multiprocessor environment, different threads may be executing on the different available processors, offering true parallel execution of code. The scheduler assigns a processor to a thread for, at most, a fixed period of time known as the *thread time quantum*. Processors are assigned to threads primarily based on the thread's priority value. Higher priority threads that become ready to run will preempt a running thread.

Since the kernel's prime role is to schedule thread activity, other OS components perform the necessary work of memory, process, security, and I/O management. These components are collectively known as the *Executive*. The Executive components have been designed (though the I/O Manager itself is a significant exception) as modular software. Over the years, Microsoft has added, deleted, merged, and separated these components as improvements and compromises deemed necessary. A good example would be the addition of the Active Directory Services, which is relatively new to Windows 2000.

The notion of keeping the kernel itself small and clean, coupled with the modularization of Executive components, provides the basis for NT's claim to extendibility. The OS has now survived about ten years of revisions, maintenance, and significant feature improvement (a.k.a., *creeping elegance*).

Performance

While the layered approach to software design is often characterized by lackluster performance, attention to fast layer interaction has been a continual effort with the NT design group. First, it should be noted that all the layers described so far execute within the same hardware mode, kernel mode. Therefore, interlayer calls often involve nothing more than a processor CALL instruction. Indeed, HAL usage is often implemented with macros, thus achieving inline performance.

Second, there has been a concentrated effort to parallelize as many tasks as possible by allocating threads to different units of work. The Executive components are all multithreaded. Helper routines seldom *block* or *busy-wait* while performing their work. This minimizes true idle time on the processor.

The performance goals of Windows 2000 impact device driver writers. As user and system threads request service from a device, it is vital that the driver code not block execution. If the request cannot be handled immediately, perhaps because the device is busy or slow, the request must be queued for subsequent handling. Fortunately, I/O Manager routines facilitate this process.

Executive Components

Since the Executive components provide the base services for the Windows 2000 operating system (other than thread scheduling), their needs and responsibilities are fairly clear. These components are explained in the following sections.

SYSTEM SERVICE INTERFACE

This component provides the entry point from user mode to kernel mode. This allows user-mode code to cleanly and safely invoke services of the OS. Depending on the platform, the transition from user mode to kernel mode may be a simple CPU instruction or an elaborate Save and Restore context switch.

OBJECT MANAGER

Almost all services offered by the OS are modeled with an object. For example, a user-mode program that needs thread-to-thread synchronization might request a *mutex* service from the OS. The OS presents the mutex in the form of an OS-based object, referenced from user mode only through a *handle*. Files, processes, Threads, Events, Memory Sections, and even Registry Keys are modeled with OS-based objects. All objects are created and destroyed by a centralized Object Manager. This allows for uniform access, life spans, and security with all objects.

CONFIGURATION MANAGER

The Configuration Manager of Windows 2000 models the hardware and installed software of the machine. A database called the Registry is used to store this model. Device drivers utilize information in the Registry to discover many aspects of the environment in which they are executed. With the introduction of Plug and Play into Windows 2000, the role of the Registry for device drivers has been significantly reduced.

PROCESS MANAGER

A process is the environment in which threads execute in Windows 2000. Each process maintains a private address space and security identity. In Windows 2000, it is important to note that processes do not *run*; instead, threads

are the unit of execution and the process is a unit of ownership. A process owns one or more threads.

The Windows 2000 Process Manager is the Executive component that manages the process model and exposes the environment in which process threads run. The Process Manager relies heavily on other Executive components (e.g., the Object Manager and Virtual Memory Manager) to perform its work. As such, it could be said that the Process Manager simply exposes a higher level of abstraction for other lower-level system services.

Device drivers seldom interact with the Process Manager directly. Instead, drivers rely on other services of the OS to touch the process environment. For example, a driver must ensure that a buffer residing with the private address space of a process remains "locked down" during an I/O transfer. Routines within the OS allow a driver to perform this locking activity.

VIRTUAL MEMORY MANAGER

Under Windows 2000, the address space of a process is a flat 4 gigabytes (4 GB) (2^{32}). Only the lower 2 GB is accessible in user mode. A program's code and data must reside in this lower half of the address space. If the program relies on shared library code (dynamic-link libraries or DLLs), the library code also must reside in the first 2 GB of address space.

The upper 2 GB of address space of every process contains code and data accessible only in kernel-mode. The upper 2 GB of address space is shared from process to process by kernel-mode code. Indeed, device driver code is mapped into address space above 2 GB.

The Virtual Memory Manager (VMM) performs memory management on behalf of the entire system. For normal user-mode programs, this means allocating and managing address space and physical memory below the 2 GB boundary. If the needed memory for a given process is not physically available, the VMM provides an illusion of memory by *virtualizing* the request. Needed memory is *paged* onto a disk file and retrieved into RAM when accessed by a process. In effect, RAM becomes a shared resource of all processes, with memory moving between files on the disk and the limited RAM available on a given system.

The VMM also acts a memory allocator in that it maintains heap areas for kernel-mode code. Device drivers can request the VMM to assign dedicated areas of pagable or nonpagable memory for its use. Further, devices that operate using DMA (direct memory access) can assign nonpagable memory as needed to perform data transfers between RAM and a device. Of course, these topics are covered in more detail in subsequent chapters.

LOCAL PROCEDURE CALL FACILITY

A Local Procedure Call (LPC) is a call mechanism between processes of a single machine. Since this *interprocess* call must pass between different address

spaces, a kernel-mode Executive component is provided to make the action efficient (and possible). Device driver code has no need for the LPC facility.

I/O MANAGER

The I/O Manager is an Executive component that is implemented with a series of kernel-mode routines that present a uniform abstraction to user-mode processes for I/O operations. One goal of the I/O Manager is to make all I/O access from user mode device-independent. It should not matter (much) to a user process whether it is accessing a keyboard, a communication port, or a disk file.

The I/O Manager presents requests from user-mode processes to device drivers in the form of an I/O Request Packet (IRP). The IRP represents a work order, usually synthesized by the I/O Manager, that is presented to a device driver. It is the job of device drivers to carry out the requested work of an IRP. Much of the remainder of this book is devoted to the proper care and processing of IRPs by device driver code.

In effect, the I/O Manager serves as an interface layer between user-mode code and device drivers. It is therefore the most important block of code that a device driver must interact with during operation.

ACTIVE DIRECTORY SERVICE

The Active Directory Service is somewhat new to Windows 2000. It provides a network-wide namespace for system resources. Previously, the internal names used to identify system resources (disk drive names, printer names, user names, file names) were managed within a restricted space of the OS. It was the responsibility of other OS components (e.g., the networking services) to *export* names across different protocols.

The Active Directory is now a uniform, secure, and standard way to identify system resources. It is based on a hierarchical scheme (strictly defined by a schema) whereby entities are categorized into organization units (OUs), trees, and forests.

EXTENSIONS TO THE BASE OPERATING SYSTEM

Although the Executive components of Windows 2000 define and implement core services of the OS, it might be interesting to note that these services are not directly exposed to user-mode applications. Instead, Microsoft defines several Application Programming Interfaces (APIs) that user-mode code treats as abstractions of OS services. These APIs form different *environmental subsystems* that application code live within. Currently, the following environmental subsystems are included with Windows 2000.

- The Win32 subsystem is the native-mode API of Windows 2000. All other environmental subsystems rely upon this subsystem to perform their work. All new Windows 2000 applications (and indeed, most

ported ones as well) rely on the Win32 subsystem for their environment. Because of its importance (and interesting implementation), this subsystem is described in more detail in the next section.

- The Virtual DOS Machine (VDM) subsystem provides a 16-bit MSDOS environment for old-style DOS applications. Despite its promise of compatibility, many existing 16-bit DOS programs do not operate properly. This is due to Microsoft's conservative and safe approach that *emulates* device (and other system resources) access. Attempts to directly access these resources results in intervention from the OS that provides safe, but not always faithful, results.
- The Windows on Windows (WOW) subsystem supports an environment for old-style 16-bit Windows applications (i.e., Windows 3.X programs). Interestingly, each 16-bit program runs as a separate thread within the address space of a single WOW process. Multiple WOWs can be spawned, but 16-bit Windows applications are then prohibited from sharing resources.
- The POSIX subsystem provides API support for Unix-style applications that conform to the POSIX 1003.1 source code standard. Unfortunately, this subsystem has not proved workable for hosting the ports of many (most) Unix-style applications. As such, most Unix applications are ported by rewriting for the Win32 environment.
- The OS/2 subsystem creates the execution environment for 16-bit OS/2 applications—at least those that do not rely on the Presentation Manager (PM) services of OS/2. This subsystem is available only for the Intel (x86) version of Windows 2000.

A given application is tightly coupled to exactly one environmental subsystem. Applications cannot make API calls to other environments. Also, only the Win32 subsystem is native—other subsystems emulate their environments and therefore experience various degrees of performance degradation compared to native Win32. Their purpose is compatibility, not speed.

Environmental subsystems are generally implemented as separate user-mode processes. They launch as needed to support and host user-mode processes. The environmental subsystem becomes the *server* for the user-mode *client*. Each request from a client is passed, using the Local Procedure Call Executive component, to the appropriate server process. The server process (i.e., the environmental subsystem) either performs the work to fulfill the request directly or it, in turn, makes a request of the appropriate Executive component.

THE WIN32 SUBSYSTEM

As the native API for Windows 2000, the Win32 environmental subsystem is responsible for

- The Graphical User Interface (GUI) seen by users of the system. It implements and exposes viewable windows, dialogs, controls, and an overall style for the system.
- Console I/O including keyboard, mouse, and display for the entire system, including other subsystems.
- Implementation of the Win32 API, which is what applications and other subsystems use to interact with the Executive.

Because the Win32 Subsystem holds special status within the system and because of its inherent requirement for high performance, this subsystem is implemented differently from any of the other subsystems. In particular, the Win32 subsystem is split into some components that execute in user mode and some that execute in kernel mode. In general, the Win32 function can be divided into three categories.

- USER functions that manage windows, menus, dialogs, and controls.
- GDI functions that perform drawing operations on physical devices (e.g., screens and printers).
- KERNEL functions, which manage non-GUI resources such as processes, threads, files, and synchronization services. KERNEL functions map closely to system services of the Executive.

Since NT 4.0, USER and GDI functions have been moved to kernel mode. User processes that request GUI services are therefore sent directly to kernel-mode using the System Service Interface, an efficient process. Kernel-mode code that implements USER and GDI functions resides in a module called WIN32K.SYS. The USER and GDI kernel components are illustrated in Figure 1.2.

Conversely, KERNEL functions rely on a standard server process, CSRSS.exe (Client-Server Runtime Subsystem), to respond to user process requests. In turn, CSRSS traps into Executive code to complete the request for such functions.

INTEGRAL SUBSYSTEMS

In addition to the Environmental Subsystems, there are also key system components that are implemented as user mode processes. These include

- The Security Subsystem, which manages local and remote security using a variety of processes and dynamic libraries. Part of the Active Directory work also resides within this logical subsystem.
- The Service Control Manager (affectionately called the *scum*, or SCM) manages services (daemon processes) and device drivers.
- The RPC Locator and Service processes give support to applications distributed across the network. Through the use of remote procedure calls, an application can distribute its workload across several networked machines.

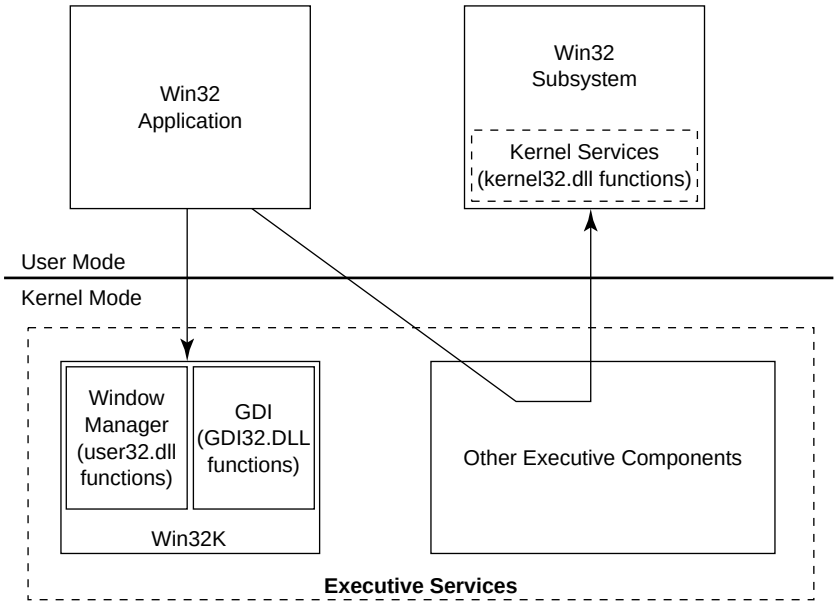

Figure 1.2 *USER and GDI kernel components.*

Kernel-Mode I/O Components

The purpose of this section is to describe the goals and architecture of the Windows 2000 I/O subsystem. Since different kinds of drivers perform wildly different kinds of service, the I/O Manager's categorization of drivers is also discussed.

Design Goals for the I/O Subsystem

The I/O subsystem of Windows 2000 added to the overall design goals of the operating system by including

- Portability, platform to platform.
- Configurability in terms of both hardware and software. For Windows 2000 drivers, this would include full support for Plug and Play buses and devices.
- Preemptable and interruptable. I/O code should never block and should always be written thread-safe.
- Multiprocessor-safe. The same I/O code should run on both uniprocessor and multiprocessor configurations.
- Object-based. The services provided by I/O code should be offered in encapsulated data structures with well-defined allowable operations.

- Packet-driven. Requests made of the I/O subsystem should be submitted and tracked using a distinct "work order" format, known as an *I/O Request Packet* (IRP).
- Asynchronous I/O support. Requests made of the I/O subsystem should be allowed to complete in parallel with the requestor's execution. When the request ultimately completes, a mechanism must exist to notify the caller of completion.

Besides these published goals, there is also strong emphasis placed on code reusability. This translates to heavy structuring of I/O code (including drivers) into logical layers. For example, bus-driving code should be layered separately from specific device code to allow for reuse of the bus code across multiple devices. In many cases, different vendors supply code for different layers. Only through careful *modularization* can this goal be achieved.

Kinds of Drivers in Windows 2000

There once was a time when a device driver author could understand the intricacies of the new hardware, learn the OS device driver interface, scope the work, and "just write the code." For better or worse, the days of monolithic device driver code have passed. Today, an author must understand the *architectures* of both complex hardware buses and heavily layered I/O subsystems just to scope the work statement. Deciding what *kind* of driver to write for Windows 2000 is itself an interesting challenge. Deciding whether to implement or to reuse a layer is yet another challenge. The purpose of this section is to describe where different kinds of drivers fit within the hardware world and the OS.

At the highest level, Windows 2000 supports two kinds of drivers, user-mode and kernel-mode. User-mode drivers, as the name implies, is system-level code running in user mode. Examples include a simulated, or *virtualized*, driver for imaginary hardware or perhaps a new environmental subsystem. Since Windows 2000 user mode does not allow direct access to hardware, a virtualized driver necessarily relies upon real driver code running in kernel mode. This book does not describe user-mode drivers. The purpose of this book is to describe *real* drivers, which in Windows 2000 are known as *kernel-mode drivers.*

Kernel-mode drivers consist of system-level code running in kernel mode. Since kernel mode allows direct hardware access, such drivers are used to control hardware directly. Of course, nothing prevents a kernel-mode driver from virtualizing real hardware—the choice between user and kernel mode is largely an implementer's choice. Again, however, the purpose of this book is to present the strategies for implementing true kernel-mode drivers for real hardware.

Moving down a level, kernel-mode drivers can be further decomposed into two general categories, legacy and Windows Driver Model (WDM). Legacy drivers were fully described in the first edition of this book. The tech-

niques needed to discover hardware and interface with the I/O subsystem are well documented. Thankfully, most of the knowledge gained by understanding legacy Windows NT drivers is transportable to the Windows 2000 (and Windows 98) WDM world.

WDM drivers are Plug and Play compliant. They support power management, autoconfiguration, and hot plugability. A correctly written WDM driver is usable on both Windows 2000 and Windows 98, though at present, Microsoft does not guarantee binary compatibility. At most, a rebuild of the driver source is necessary using the Windows 98 DDK (Device Driver Kit).

Moving down yet another level, legacy and WDM drivers can be further decomposed into three categories, high-level, intermediate, and low-level. As the names imply, a high-level driver depends on intermediate and low-level drivers to complete its work. An intermediate driver depends on a low-level driver to complete its work.

High-level drivers include file system drivers (FSDs). These drivers present a nonphysical abstraction to requestors that, in turn, is translated into specific device requests. The need to write a high-level driver is apparent when the underlying hardware services are already provided by lower levels—only a new abstraction is required for presentation to requestors.

Microsoft supplies an Installable File System (IFS) kit, sold separately from MSDN or any other product. The IFS kit requires the DDK (and other products) for successful file system development. There are numerous restrictions on the types of file systems that can be developed using this kit. For pricing and ordering information, you can visit the HWDEV virtual site of Microsoft's Internet site. This book does not address file system development.

Intermediate drivers include such drivers as *disk mirrors*, *class drivers*, *mini drivers*, and *filter drivers*. These drivers insert themselves between the higher-level abstractions and the lower-level physical support. For example, a disk mirror receiving the request from the high-level FSD to write to a file translates such a request into two requests of two different low-level disk drivers. Neither the higher nor lower levels need to be aware that mirroring is, in fact, occurring.

Class drivers are an elegant attempt at code reuse within the driver model. Since many drivers of a particular type have much in common, the common code can be placed in a generic *class* driver separate from the physical, device-specific code. For example, all IDE disk drivers share considerable similarity. It is possible to write the common code once, placing it in a generic class driver that loads as an intermediate driver. Vendor and device specific IDE drivers would then be written as *mini drivers* that interact with the generic class driver.

Filter drivers are intermediate drivers that intercept requests to an existing driver. They are given the opportunity to modify requests before presentation to the existing driver.

Finally, within the WDM world, intermediate drivers can also consist of *Functional Drivers*. These drivers can be either class or mini drivers, but they always act as an interface between an abstract I/O request and the low-level

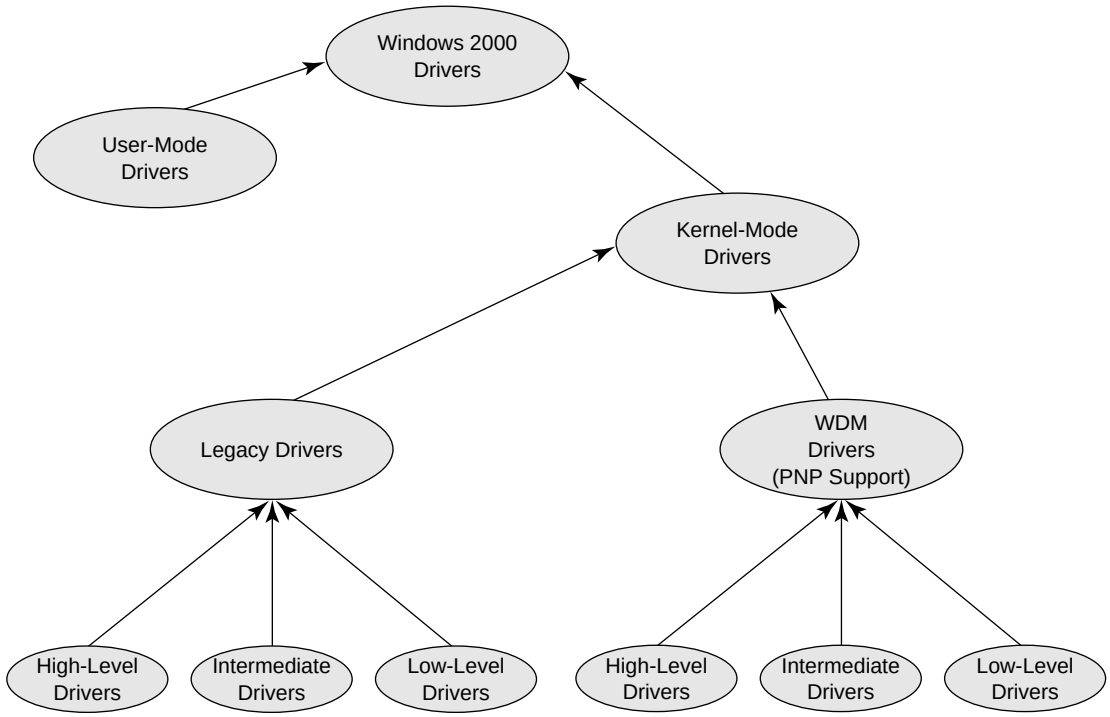

Figure 1.3 *Driver classifications in Windows 2000.*

physical driver code. Within the DDK documentation, the term *Functional Driver* is sometimes interchanged with Class or Mini Driver. The context determines the meaning.

Low-level drivers include controllers for the hardware buses. For example, the SCSI Host Bus Adapter is one such low-level driver. Such drivers interact with Windows 2000 HAL layer and/or the hardware directly. In the WDM world, low-level drivers include the notion of a *Physical Driver*. These Physical Drivers interact with one or more Functional Drivers.

Figure 1.3 shows the driver classifications in Windows 2000.

Special Driver Architectures

Building upon the intermediate driver strategy described in the last section, Microsoft supplies driver architectures for several types or classes of devices.

- Video drivers
- Printer drivers

- Multimedia drivers
- Network drivers

These architectures conform to the spirit, if not to the letter, of the classifications of the last section. Each architecture is described in more detail in the following sections.

Video Drivers

Video drivers in Windows 2000 present special requirements to the I/O subsystem. Because the graphical interface is constantly exposed to users, the apparent overall speed of the system is judged (often incorrectly) by the performance of this component. Competition among video adaptor hardware vendors has forced aggressive graphic accelerators to be included on high-end boards. The video driver architecture must exploit such hardware when it exists, yet provide full compatibility and capability when it does not. Finally, video drivers have evolved since the 16-bit world of Windows. There is a need to provide as much compatibility as possible with legacy drivers.

The video driver architecture of Windows 2000 is shown in Figure 1.4. The shaded components are provided with Windows 2000. Vendors of specific display adaptors supply the display driver. Since many display adaptors are designed using common chip sets, the chip set manufacturer supplies the video miniport *class driver* for its adaptor-manufacturing customers. For example, an ET4000 Miniport driver exists for all adaptors that utilize the ET4000 chip set. The extra hardware surrounding the chip set is driven with adaptor-specific display driver code.

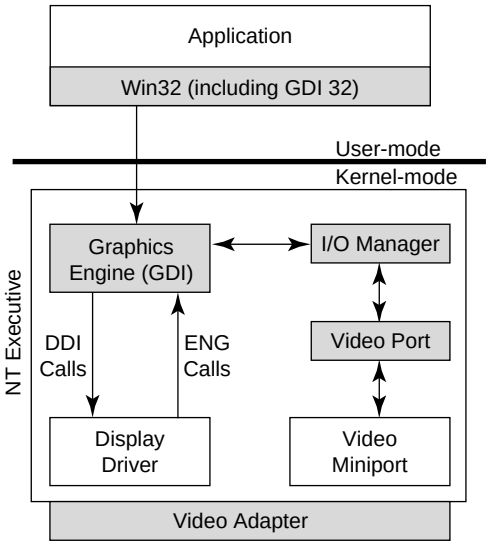

Figure 1.4 *Video driver architecture.*

Fundamentally, the video driver architecture differs from the standard I/O architecture in that user applications do not communicate directly with the I/O Manager when requesting drawing services. Instead, user-mode code interacts with a *Graphics Device Interface* (GDI) component of the kernel.

The GDI implements functions that allow the drawing of lines, shapes, and text in selected fonts. The GDI, therefore, is similar to a high-level driver. In turn, the GDI relies upon the services of the display driver and the I/O Manager to complete its work. Communication between the GDI and display driver is bidirectional. Where speed is paramount, the GDI can invoke functions in the display driver directly, bypassing the I/O Manager altogether. The display driver implements an interface known as the *Device Driver Interface* (DDI), which consists of functions prefixed with the **Drv** string. Conversely, the display driver relies on common graphics library routines implemented within the GDI. These GDI routines are known as *Graphics Engine calls* and are prefixed within the **Eng** string.

For less time-critical services, the GDI relies upon the traditional layered approach of the Windows 2000 I/O subsystem. The GDI uses the I/O Manager to invoke support routines of the video port and miniport intermediate drivers. An example of a function that would be implemented with port and miniport drivers would be a mode-switch command. Requests made by the I/O Manager of the video port driver are in the standard IRP format. The video port driver converts these IRPs into *Video Request Packets* (VRPs), which are then sent to and processed by the video miniport driver.

Printer Drivers

Printer drivers differ from standard Windows 2000 drivers in several ways. First, a print job may be directed to a spooling mechanism before being sent to a physical device. Second, the physical device is often connected to a remote machine, thus burdening the spool process with the use of RPC calls. Finally, the different printer stream protocols used by different printer devices (e.g., Postscript and HPCL) burden printer driver authors with yet another layer of integration.

The spooler components are shown in Figure 1.5. If spooling is enabled, an application's print job is first directed to a file by the spooler. The spooler then dequeues jobs as a printer (perhaps in a logical queue) becomes available. Data is passed to a print provider, which then directs the output to a local or remote printer.

The shaded components in Figure 1.5 are supplied with Windows 2000, as are several print providers. The client-side spooler component, *winspool.drv* or *Win32spl.DLL* (when remote printing), is simply an RPC-based client stub. It connects with a server-side RPC stub, *spoolsv.exe*, to implement a spooler API.

The server-side stub code relies on a routing service, *spoolss.dll*, which connects a specific print provider based on the target printer name. Finally,

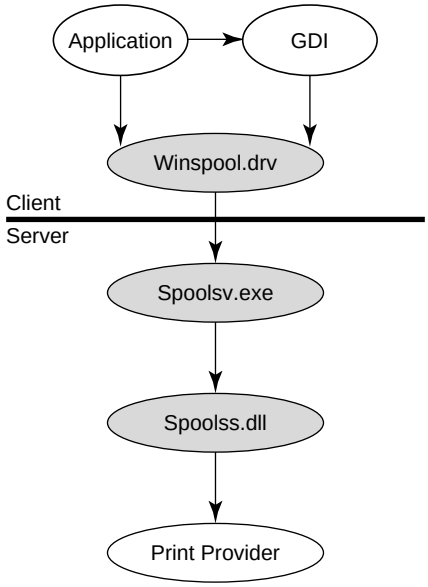

Figure 1.5 *Printer spooler components.*

the print provider acts as an abstract front-end for a specific print server or device. A specific device's job queue is created and managed by the print provider. A single print provider can serve the needs of an entire class of printers. Thus, the print providers supplied with Windows 2000 are usually sufficient to serve the needs of most applications and printers. A local, remote, and Internet print provider are included with Windows 2000.

Different print device characteristics, or perhaps different network protocols, sometimes require that a unique print provider be supplied. For example, Novell supplies a print provider for Windows 2000 that directs output to a NetWare print server.

Whether or not the spooling process is involved in a print job, it is the GDI which must ultimately render an application's drawing command into a printer-specific format. The GDI relies upon the services of the Printer Driver. The Printer Driver components consist of a Printer Graphics DLL and a Printer Interface DLL.

The printer graphics DLL is the component responsible for rendering data for a specific device. Depending on device capabilities, the work involved can range from *intense* for a bit-oriented device to *high-level* for a device supporting a full graphics engine (such as Postscript.) NT 4 required that this DLL reside in kernel mode, but the performance advantages of placing this code in the trusted category were not significant enough to outweigh the many disadvantages. In Windows 2000, the printer graphics DLL can reside in either user mode or kernel mode. Flexible configuration and higher system reliability result from placement of this DLL in user mode.

Each function exported by the printer graphics DLL is prefixed with the string **Drv.** The functions defined are invoked by the GDI when rendering is performed.

The printer interface DLL is responsible for configuration of device-specific parameters by providing a user interface for each device option. For example, a printer with multiple paper trays needs a way for a user to specify a default tray and paper size. The printer interface DLL provides user interfaces by building one or more property sheets. These sheets are a kind of Windows dialog, with standard Windows controls allowing selection of various options.

Multimedia Drivers

To support multimedia devices such as sound cards and TV tuners, Windows 2000 supports a new model known as *Kernel Streaming* (KS). Kernel Streaming consists of both Functional and Filter Drivers. Applications interact with a KS driver using Methods, Properties, and Events, familiar terms from the COM (Component Object Model) world. These mechanisms apply to four different kinds of KS objects exposed to an application: Filters, Pins, Clocks, and Allocators. Each KS object is exposed to an application as a standard I/O file object.

A filter object (which should not be confused with a filter driver) is the top-level entity exposed to an application performing multimedia operations. For example, an application might open a microphone filter on a sound card.

A pin object is a subobject of a filter. It represents a node (input or output) on a device. For example, the microphone filter might expose a single pin for input. An output pin could then be used to read, or acquire, the digitized signal.

A clock object exposes the real-time clock of a multimedia device, if equipped. The clock object may signal an application with an event when the clock timer expires or ticks.

Allocator objects represent the direct memory interface to a multimedia card. Memory on the card can be allocated or freed using this object.

Windows 2000 includes a generic class driver for streaming devices, *Stream.sys*. In most cases, it is only necessary to write a mini driver to support a specific device such as an audio card or video camera. The class driver implements the Windows 2000 Kernel Streaming abstraction, while the vendor-supplied mini driver utilizes the class driver's services, including buffer and DMA support, to support device-specific actions.

Network Drivers

Network drivers in Windows 2000 follow the Open Systems Interconnection (OSI) standard of ISO. This is a seven-layer model, with the top layer being the application software and the bottom layer being the physical hardware

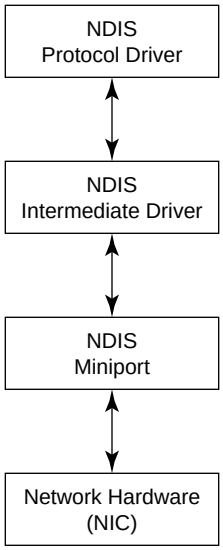

Figure 1.6 *Network driver interface layers.*

connection and topology of the network. Network Interface Cards (NIC) provide the hardware interface on most platforms for a given network. NIC drivers are written to support a specific device and Windows 2000 ships with popular NIC driver types. Separately, the OSI transport layer (the fourth layer up within the model) is provided by *protocol* drivers. It is possible to bind different protocols to the same physical NIC driver.

A Network Driver Interface Specification (NDIS) provides library support for NIC drivers that usually permits a NIC vendor to supply only an NDIS miniport driver to manage hardware specifics. Higher layers of NDIS, the NDIS intermediate driver and the NDIS protocol driver, provide media translations, filtering, and media-specific actions when required. The layering of NDIS is shown in Figure 1.6.

Windows 2000 includes a layer of kernel-mode software known as the Transport Driver Interface (TDI). This layer interfaces between the NDIS layer and higher-level software abstractions such as sockets and NetBIOS. The TDI layer makes the construction of Windows 2000 components such as the NetBIOS redirector and server easier and more portable.

Summary

Windows 2000 provides a rich architecture for applications to exploit. Of course, this richness comes at a price that device driver authors (among others) will have to pay. The I/O processing scheme of Windows 2000 is com-

plex and the need to keep a view of the "big picture" is difficult, but necessary.

At this point, there should be a good understanding of what kinds of drivers exist (and where they fit) in Windows 2000. The next chapter begins a discussion of the hardware that drivers must control. An overview of the various buses that are supported by Windows 2000 is presented.