8

How the Host Communicates

Under Windows, an application that wants to access a USB peripheral must communicate with a device driver that knows how to manage communications with the system's USB drivers. This chapter explains how Windows manages USB communications and explores the options for device drivers.

Device Driver Basics

A device driver is a software component that enables applications to access a hardware device. The hardware device may be a printer, modem, keyboard, video display, data-acquisition unit, or just about anything controlled by circuits that the CPU can access. Some devices, such as internal disk drives, are inside the box with the CPU. Others, including just about all USB devices, are external devices that connect to the system via cables (or wireless links). As Chapter 7 explained, some device drivers are class drivers that handle communications with a variety of devices that perform similar functions.

Insulating Applications from the Details

Applications are the programs that users run, including everything from word processors and databases to special-purpose applications that access custom hardware. A device driver insulates applications from having to know details about the physical connections, signals, and protocols required to communicate with a device.

A device driver can enable application code to access a peripheral when the application knows only the peripheral's name (such as HP LaserJet 2300) or the device's function (joystick, drive, scanner). The application doesn't have to know the physical address of the port the peripheral attaches to or monitor and control handshaking signals. Applications don't even have to know whether a device uses USB or another interface. The application code can be the same for devices that perform similar functions but have different interfaces, with the hardware-specific details handled at a lower level.

The device driver translates between application-level and hardware-specific code. Applications communicate with device drivers using functions supported by the operating system. The hardware-specific code handles the protocols needed to access the peripheral's circuits, including detecting the states of status signals and toggling control signals at appropriate times.

The Windows drivers for USB devices use a layered driver model where each driver in a series performs a portion of the communication. The top layer contains a client device driver (or client driver for short) that manages communications between applications and lower-level bus drivers. Another term for client driver is function driver. The bottom layer contains bus drivers that manage communications between the client driver and the hardware. One or more filter drivers may supplement the client and bus drivers.

The layered driver model simplifies the job of writing drivers. Devices can share code for tasks they have in common. The drivers that handle communications with the system's USB hardware are included with Windows, so writers of client drivers don't have to handle these details. Note also that the layered driver model means that applications *can't* access USB ports directly. Windows doesn't allow it. All application communications must be with a driver assigned to a device.

Options for USB Devices

There are several approaches to obtaining a driver for a USB device that you're developing. Many devices can use a driver that's included with Windows or provided by a chip vendor or other source. Other devices may require custom drivers. When a custom driver is necessary, toolkits are available to simplify and speed up the task of driver writing. Sometimes more than one way will work, and the choice depends on what's easier, cheaper, or offers better performance.

As Chapter 7 showed, many peripherals fit into standard classes such as disk drives, printers, modems, keyboards, and mice. Each of these devices is available with a choice of interfaces, including USB. For example, a disk drive may use ATA/ATAPI, SCSI, IEEE-1394, or USB. When the devices in a class may have any of multiple hardware interfaces, supplemental drivers can support different interfaces. If a device has features or capabilities beyond what a class driver supports, a device-specific filter driver may be able to support these as needed.

User and Kernel Modes

To understand what the device driver has to do, you need to understand where the driver fits in the communications path of a data transfer. Even if you don't need to write a driver for your device, understanding the driver's role will help in understanding the application-level code that you do write.

In the most general sense, a device driver is any code that handles communication details for a hardware device that interfaces to a CPU. Even a short subroutine in an application can be considered a device driver. Under Windows, the code for most drivers, including USB drivers, differs from application code because the operating system allows the driver code a greater level of privilege than applications are allowed.

Program code in a Windows system runs in one of two modes: user or kernel. Each allows a different level of privilege in accessing memory and other system resources. Applications must run in user mode. Most drivers, including all USB drivers, run in kernel mode, though a driver for a USB device

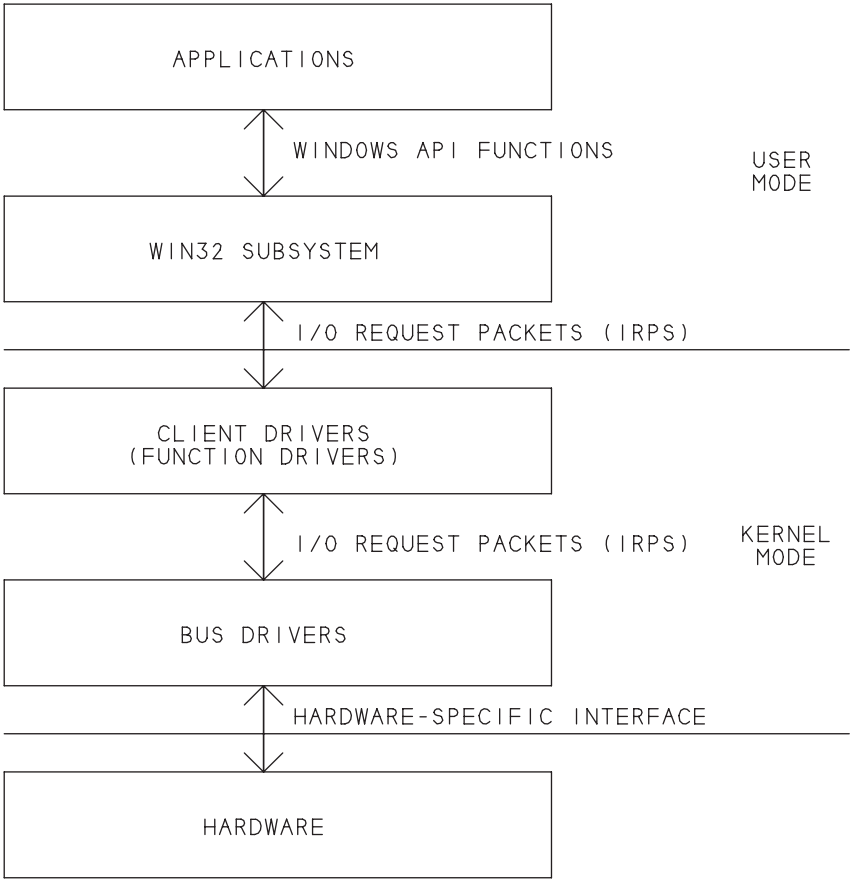

Figure 8-1: USB uses a layered driver model under Windows, with separate drivers for devices and the buses they connect to.

may have a supplementary user-mode driver. Figure 8-1 shows the major components of user and kernel modes in a USB communication.

In user mode, Windows limits access to memory and other system resources. Windows doesn't allow an application to access memory that the operating system has designated as protected. Managing memory in this way enables a PC to run multiple applications at the same time. If an application crashes, other applications shouldn't be affected. On x86 processors, user mode corresponds to the CPU's Ring 3 mode.

In kernel mode, the code has unrestricted access to system resources, including the ability to execute memory-management instructions and control access to I/O ports. A kernel-mode driver can permit or deny an application access to a device. For example, a joystick driver can allow any application to use a device, or the driver can allow an application to reserve the device for exclusive use. Other abilities that Windows reserves for kernel-mode drivers include DMA transfers and responding to hardware interrupts. On x86 processors, kernel mode corresponds to the CPU's Ring 0 mode.

Applications communicate with client drivers using Windows API functions or other components that call API functions internally but shield application programmers from the details. The Windows WIN32 subsystem manages communications between applications and client drivers. To communicate with a USB device, an application typically doesn't have to know anything about the USB protocol, or even if a device uses USB at all.

Drivers communicate with each other using structures called I/O request packets (IRPs). Windows defines a set of IRPs that drivers can use. Each IRP requests a single input or output action. A client driver for a USB device uses IRPs to communicate with the bus drivers that handle USB communications. The bus drivers are included with Windows and require no programming by applications programmers or device-driver writers.

WDM Drivers

USB device drivers for Windows are kernel-mode drivers that must conform to the Windows Driver Model defined by Microsoft for use under Windows 98 and later. These drivers are known as WDM drivers and have the extension *.sys.* (Other driver types may also use the *.sys* extension.)

Application programmers have a choice of programming languages, including Visual Basic, Delphi, and C and its derivatives. But to write a driver for a USB device, you need a tool that is capable of compiling a WDM driver. The Windows DDK includes a C compiler for this purpose. An exception is driver toolkits that provide a generic driver and either require no program-

ming at all or permit you to use other compilers to customize a generic driver with a user-mode component.

Layered Drivers

In the layered driver model used in USB communications, each layer handles a piece of the communication process. Dividing communications into layers is efficient because devices that have tasks in common can use the same driver for those tasks. For example, all kinds of devices may use USB, so it makes sense to have one set of drivers to handle the USB-specific tasks that are common to all. Including these drivers with Windows means that device vendors don't have to provide drivers. The alternative would be to have each device driver handle communicating directly with the USB hardware, with much duplication of effort.

Figure 8-2 shows the layers involved with USB communications under Windows XP.

Client Drivers

A client driver can consist of one or more files. The main client driver can be a class driver provided with Windows or a vendor-provided driver. The client driver manages communications that are specific to a device or a class of devices. A class driver may also communicate with a miniclass driver that manages communications with a subset of devices in a class. For example, the HID USB miniclass driver manages USB-specific communications with HID-class devices that have USB interfaces. Other HID miniclass drivers could manage bus-specific communications with HIDs that have other hardware interfaces.

A client driver or miniclass driver may also have one or more upper and lower filter drivers (Figure 8-3). An upper-level filter driver can monitor and modify communications between applications and a client driver. A lower-level filter driver can monitor and modify communications between a client driver and the bus drivers.

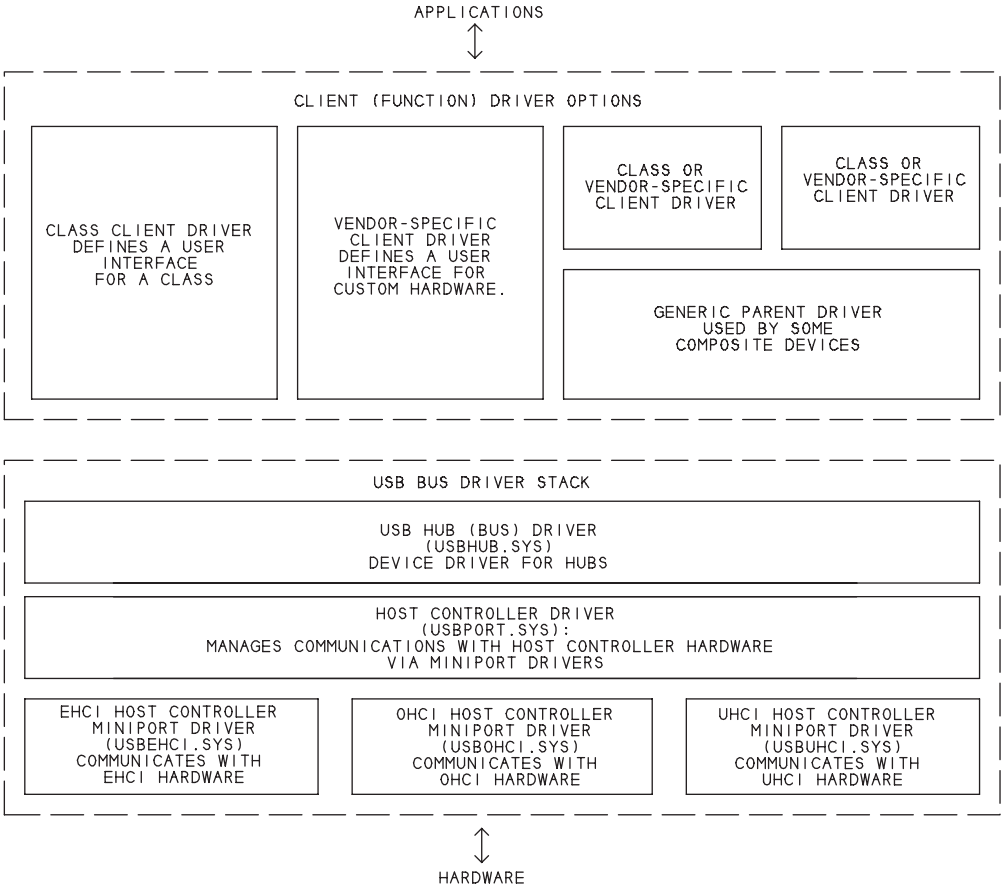

Figure 8-2: USB communications under Windows XP involve the USB bus driver stack and one or more client drivers.

For some composite devices, Windows XP loads a USB common-class generic parent driver between the bus drivers and the client drivers for the device's interfaces. The generic parent driver handles synchronization, Plug-and-Play, and power-management functions for the device as a whole and manages communications between the bus drivers and the client drivers for the composite device's interfaces.

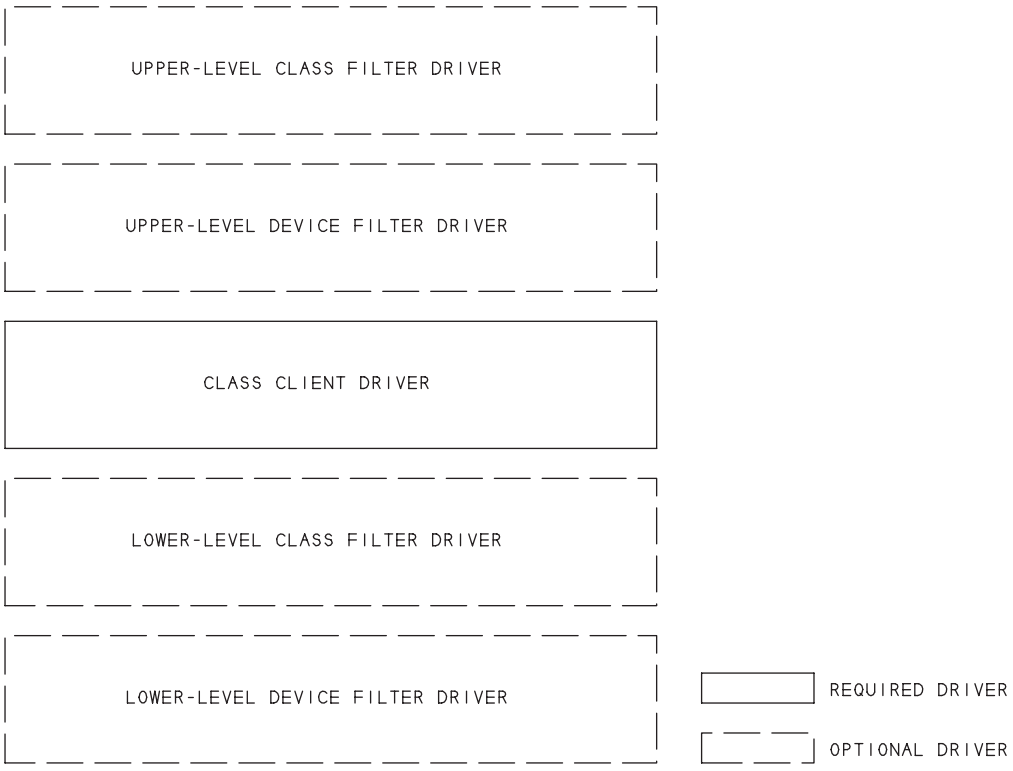

Figure 8-3: A client driver can have one or more filter drivers that monitor or modify communications with devices.

USB Drivers

Under Windows XP, the USB bus drivers consist of the host-controller driver, one or more miniport drivers, and the hub driver. The host-controller driver, sometimes called the port driver, manages tasks that are common to all host controllers. The host controller driver consists of a port driver (*usbport.sys*) and one or more miniport drivers that each manage communications with one of the three host-controller types. The hub driver manages communications with the system's hubs. In Windows XP, the hub driver is *usbhub.sys*.

The bus drivers are included with Windows, and application and device-driver writers don't have to know the details about how they work.

Perhaps because of this, Microsoft provides little documentation for these drivers. If you want to know more about how low-level communications work, one source of information is the source code and other documentation from the Linux USB Project.

Host Controller Types

There are three types of host controllers. Two support low- and full-speed communications and one supports high-speed communications. The low- and full-speed controller types are the Open Host Controller Interface (OHCI) and the Universal Host Controller Interface (UHCI). High-speed host controllers implement the Enhanced Host Controller Interface (EHCI). The USB-IF's Web site has links to the specifications.

Windows' Device Manager enables you to view information about the host controllers in a PC. To view the driver type, right-click the host controller name, select Properties, then Driver and Driver Details. One of the drivers should have *ohci*, *uhci*, or *ehci* in the name. Chapter 9 has more about using the Device Manager.

OHCI and UHCI Differences

In Windows XP, controllers that conform to the OHCI standard use the driver *usbohci.sys*, and controllers that conform to the UHCI standard use the driver *usbuhci.sys*. In other Windows editions, the driver names can vary but will contain *ohci* or *uhci*. Both drivers provide a way for the USB hardware to communicate with the bus-class driver. The two drivers take different approaches to implementing the host-controller's functions. UHCI places more of the communications burden on software and allows using of simpler, cheaper hardware. OHCI places more of the burden on the hardware and allows simpler software control. UHCI was developed by Intel and OHCI was developed by Compaq, Microsoft, and National Semiconductor.

The differences should be transparent to driver developers and application programmers. Both controllers comply fully with the USB specification. Their performance can differ, however. Developers shouldn't assume their device works fine based on tests with one host-controller type.

An OHCI controller is capable of scheduling more than one stage of a control transfer in a single frame, while a UHCI controller always schedules each stage in a different frame. For bulk endpoints with a maximum packet size less than 64 bytes, a UHCI driver attempts no more than one transaction per frame, while an OHCI driver may schedule additional transactions in a frame. An OHCI controller will poll an interrupt endpoint at least once every 32 milliseconds, even if the endpoint descriptor requests a maximum latency of 255 milliseconds, while UHCI controllers can, but don't have to, support less-frequent polling.

Developers who use UHCI hosts are sometimes surprised when their devices fail when connected to an OHCI host, usually because the device isn't expecting to see multiple transaction attempts per frame for a single transfer. Every device should work with both controller types. Test your device on both!

Supporting All Speeds

An EHCI controller handles high-speed communications only. The EHCI specification says that a host that supports EHCI must also support low and full speeds except for the unusual situation where every port has a permanently attached high-speed device. To support low and full speeds, the host must have a companion OHCI or UHCI host controller or a USB 2.0-compliant hub, which performs the function of a host controller for low- and full-speed devices. Just about every PC with an EHCI controller has a companion OHCI or UHCI controller. An EHCI controller and a companion OHCI or UHCI controller can share a bus.

Users and application programmers don't have to know or care which host controller is communicating with a device, though Windows will warn if the system has high-speed-capable ports and a user attaches a high-speed-capable device to a 1.x hub. The driver for EHCI controllers is *usbechi.sys*.

Communication Flow

One way to better understand what happens during a USB transfer is to look at an example. The following are the steps in a USB transfer with a data-acquisition device that uses a vendor-specific client device driver.

Preliminary Requirements

Before an application can communicate with the device, several things must happen. When a device is attached, Windows manages enumeration, as described in Chapter 4. To identify which driver to use on first enumeration, Windows compares the retrieved descriptors with the information in the system's INF files. Chapter 9 has more about INF files. When a device supports multiple configurations, the driver selects a configuration. The application that will access the device can then obtain a handle that identifies the device and enables communications with it.

Initiating Data Transfers

To read data from a data-acquisition device, a user might click a button in a data-acquisition application. Or a user might select an option that causes the application to request a reading once per minute. Or periodic data acquisitions might start automatically when the device's driver is loaded or when the user runs the application.

The Application's Role

Windows includes API functions that enable applications to communicate with client drivers. Applications written in Visual Basic, C/C++/C#, Delphi, and other languages can call API functions. The available functions vary with the driver, but applications typically can open communications with CreateFile, exchange data using a combination of ReadFile/ReadFileEx, WriteFile/WriteFileEx, and DeviceIoControl, and close communications with CloseHandle.

To make programming simpler and safer, many languages support alternate ways to access devices of various types. Microsoft's .NET platform includes classes and methods that eliminate the need to call many API functions directly. Instead, applications communicate via intermediate layers with a

Common Language Runtime (CLR) component that in turn calls the API functions. For example, in Visual Basic .NET, the PrintDocument class includes methods that enable applications to send text and images to a printer.

Communications with any device may require calling API functions at times, however. For example, .NET doesn't provide methods for detecting device attachment and removal via WM_DEVICECHANGE messages.

Each call to an API function includes the request, other required information such as the data to write or amount of data to read, and a handle for accessing the device. Microsoft's Platform Software Development Kit (SDK) documents the functions.

Although the names suggest that the functions are used only with files, ReadFile and WriteFile (as well as ReadFileEx and WriteFileEx) can transfer data to and from any driver that supports handle-based operations. The data being read or data to be written is stored in a buffer specified by the function call. A call to ReadFile doesn't always cause the driver to retrieve data from a device. The function may instead return data that a driver has already requested and stored in a buffer. The details vary with the driver.

DeviceIoControl is another way to transfer data to and from buffers. Included in each DeviceIoControl request is a control code that identifies a specific request. Unlike ReadFile and WriteFile, a single DeviceIoControl call can transfer data in both directions. The driver specifies what data, if any, to pass in each direction for each control code. Some control codes are commands that don't need to pass additional data.

Windows drivers define control codes used by drives and other common devices. For example, IOCTL_STORAGE_CHECK_VERIFY determines if media is present and readable on removable media and IOCTL_STORAGE_GET_MEDIA_TYPES returns the types of media supported by a drive.

A vendor-specific driver can also define control codes. Because the codes are sent only to a specific driver, it doesn't matter if other drivers use the same codes. For example, Cypress Semiconductor's general-purpose driver

CyUsb.sys defines a series of DeviceIoControl codes for transferring data, configuring a device, and requesting status and configuration information.

A driver may also define additional functions that applications can use. For example, the HID driver defines the functions Hid_GetFeature and HidD_SetFeature for retrieving and sending Feature reports. These functions use DeviceIoControl internally, but expose driver-specific functions for application programmers.

The Client Driver's Role

When an application calls an API function that reads or writes to a USB device, Windows passes the call to the appropriate client driver. The driver passes the request on in a format the USB bus-class driver can understand.

As mentioned earlier, drivers communicate with each other using IRPs. For USB communications, the IRPs contain structures called USB Request Blocks (URBs), which enable a driver to configure devices and transfer data. For example, a driver requests a descriptor by submitting an IRP that contains this URB:

```
URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE
```

The Windows DDK documents the URBs.

A client driver requests a transfer by creating an URB and submitting it in an IRP to a lower-level driver. The bus and host-controller drivers handle the details of scheduling transactions on the bus. For interrupt and isochronous transfers, if there is no outstanding IRP for an endpoint when its scheduled time comes up, the host controller skips the transaction.

For transfers that require multiple transactions, the client driver submits a single IRP for the entire transfer. All of the transfer's transactions are then scheduled without requiring further communications with the client driver.

If you're using an existing client driver (rather than writing your own), you need to understand how to access the driver's application-level interface, but you don't have to concern yourself with IRPs and URBs. If you're writing a client driver, you need to provide the IRPs that communicate with the system's USB drivers.

The Hub Driver's Role

The USB hub driver, also called the bus driver, is the device driver for the hubs on the bus. The bus driver requires no programming by device developers.

The Host-controller Driver's Role

The host-controller driver passes data provided by the client driver to the host-controller hardware, which in turn connects to the bus. The host-controller driver requires no programming by device developers.

The Device's Role

Data that leaves the host's port may pass through additional hubs. Eventually the data reaches the hub that connects to the device, and the hub passes the data on to the device. The device recognizes its address, reads the incoming data, and takes appropriate action.

The Response

Most communications require a response, which may include data sent in response to the request or a packet with a status code. This information travels back to the host in reverse order: through the device's hub, onto the bus, and to the PC's hardware and software. A client driver may pass a response on to an application, which may display the result or take other action.

Ending Communications

When communications are complete, an application can use the API function CloseHandle to free system resources.

More Examples

Communications with other USB devices follow a similar path, but there can be differences in how the transfer initiates and in how the client driver handles communications.

Other examples of a user initiating a transfer are clicking on a USB drive's icon to view a disk's folders or clicking Print in an application to send a file

to a USB printer. In each of these cases, no data transfers until the application requests a communication and the device driver fills a buffer with data to send or makes a buffer available for received data.

A driver can also cause the host to continuously request data from a device whether or not an application has requested it. For example, a keyboard driver causes the host to request keypress data at frequent intervals because there is no way to predict when a user will press a key.

The host also sends requests to enumerate devices on system power-up or device attachment. The device's hub causes the host to initiate these requests when the hub notifies the host of the presence of a device. A suspended device can use USB's remote-wakeup feature to initiate a transfer by signaling its hub, and in turn the host, to request resuming communications.

Creating a Custom Driver

Creating a WDM driver is not a trivial task. Writing a driver requires expertise in C programming and a fair amount of knowledge about how Windows communicates with hardware and applications. However, several products can help to simplify and speed up the process.

Writing a Driver from Scratch

The minimum requirement for writing a device driver from scratch is the Windows Driver Development Kit (DDK), which includes what you need to create a driver: a C compiler, a linker, build utilities, and documentation. Also included is example source code for filter drivers and drivers that request bulk and isochronous transfers. The example drivers are a useful starting point for developing a custom driver.

How to write a USB driver from the ground up is a much bigger topic than this book has room for. An excellent book in this topic is *Programming the Microsoft Windows Driver Model* by Walter Oney.

Using a Driver Toolkit

A driver toolkit provides a way to jump start driver development by doing as much of the work for you as possible. Toolkits that support creating USB drivers are available from Jungo Ltd. and Compuware NuMega.

There are two general categories of toolkits. One provides a generic driver that handles USB communications and generates a device-specific user-mode driver and INF file for use with the driver. This approach is very fast and requires no programming at all to create the driver but can't handle every situation. Other toolkits provide libraries and other tools that assist in writing a custom driver for a device. This approach is more flexible but requires programming expertise.

Automated Driver Generation

All USB communications follow the protocols defined in the USB specification, so it makes sense that a single generic driver should be able to communicate with just about any device. A full-featured generic USB driver should support all four transfer types, including vendor-defined control requests. The driver should also support the power-management and Plug-and-Play capabilities required of all WDM drivers. Additional functions such as the ability to retrieve descriptors or select a configuration or interface are useful as well.

Jungo's WinDriver USB Device toolkit requires no driver programming at all. A DriverWizard generates files that you can compile to create a custom user-mode driver in an *.exe* file. The user-mode driver communicates with the provided kernel-mode driver. You can compile the files generated by the Wizard using Visual C++, C++ Builder, or Delphi. The DriverWizard also creates an INF file for the device.

From the DriverWizard, you can select your device from the detected devices and test communications by reading and writing to the device's endpoints. You can then request the DriverWizard to create the driver files. When the driver has been installed, applications can communicate with the device using device-specific functions such as MyDevice_Open and MyDevice_GetDeviceInfo.

For faster performance, you can move portions of your code from the user-mode driver to a kernel-mode driver called a Kernel PlugIn, which you compile with Visual C++. For debugging, the included Debug Monitor application enables you to monitor communications handled by the driver. Different editions of WinDriver USB support Windows, Windows CE .NET, and Linux.

Toolkits that Provide Libraries for Creating Custom Drivers

The completely automated toolkits aren't suitable for every device. They can't create filter drivers, and you may want a completely custom driver for the best possible performance. Two products for creating custom drivers are CompuWare's DriverWorks in the DriverStudio suite and Jungo's KernelDriver USB.

Each of these products has Wizards and code libraries that do much of the work for you. You need to fill in the provided skeleton code and compile the driver. The driver's performance can be as fast as if you had written the driver from scratch. DriverWorks is capable of generating driver code for devices that use other buses besides USB. Jungo has a separate KernelDriver product for non-USB devices.

Using GUIDs

A Globally Unique Identifier (GUID) is a 128-bit value that uniquely identifies a class or other entity. Windows uses GUIDs in identifying two types of device classes: device setup classes and device interface classes. A device setup GUID identifies a device setup class, which encompasses devices that Windows installs in the same way. A device interface GUID identifies a device interface class. The device interface GUID provides a mechanism for applications to communicate with a driver assigned to devices in the class. In many cases, devices that belong to a particular device setup class also belong to the same device interface class. Some SetupDi_ API functions accept either type of GUID. But each type of GUIDs provides access to different types of information used for different purposes.

The conventional format for expressing GUIDs divides the GUID into five sets of hex characters, with the sets separated by hyphens.

This is the GUID for the HIDCLASS device setup class:

745a17a0-74d3-11d0-b6fe-00a0c90f57da

This is the GUID for the HID device interface class:

4d1e55b2-f16f-11cf-88cb-001111000030

Driver writers who need to provide a custom GUID can generate one using the *guidgen* utility included with Visual C++. The utility uses an algorithm that makes it extremely unlikely that someone else will create an identical GUID.

Device Setup GUIDs

A device setup GUID identifies devices that Windows sets up and configures in the same way, using the same class installer and co-installers. The system file *devguid.h* defines device setup GUIDs for a variety of classes. The file is included in the Windows DDK.

Table 8-1 shows some device setup classes that might apply to USB devices. Most peripherals should use a device setup class that corresponds to the device's function, such as printer or disk drive. Several of the class names describe functions that obviously match one of the defined USB classes. A single device can belong to multiple setup classes, such as HID and Mouse. The USB class is appropriate for USB hosts and hubs, as well as any device that has unique installation and configuration requirements or capabilities that don't fit another class. A vendor-specific class is another option for such devices, but Microsoft discourages adding vendor-specific classes.

Each device setup GUID corresponds to a Class key in the system registry. Each Class key has a subkey for each instance of a device in the class. Chapter 9 has more about Class keys.

Applications can use device setup GUIDs to retrieve information and perform various installation functions on devices. The *devcon* example in the Windows DDK shows how to use device setup GUIDs in detecting and

Table 8-1: A selection of the device setup classes supported by Windows and the USB device classes that encompass devices in the setup class.

Device Setup Class	USB Class
Battery Devices	HID
CD-ROM Drives	Mass storage
Disk Drives	Mass storage
Human Interface Devices (HID)	HID
Imaging Device (still image)	Still image capture
Keyboard	HID
Modem	Communications
Mouse	HID
Printers	Printer
Smart Card Readers	Chip/smart card interface
Tape Drives	Mass storage
USB	Host controllers and hubs, vendor-specific functions

retrieving information about devices and performing functions such as enabling, disabling, restarting, updating drivers for, and removing devices. Users can perform these same functions via Windows' Device Manager.

Device Interface GUIDs

A class or device driver can register one or more device interface classes to enable applications to learn about and communicate with devices that use the driver. Each device interface class has a device interface GUID.

Using a device interface GUID and SetupDi_ functions, an application can find all attached devices in a device interface class. On detecting a device, the application can obtain a device path name to pass to the CreateFile function. CreateFile returns a handle that the application can use to read and write to the device. Applications can also use device interface GUIDs to request to be notified when a device is attached or removed. Chapter 10 has more about using GUIDs for this purpose.

Unlike the device setup GUIDs, device interface GUIDs aren't stored in one file. A driver package may include a C header file or Visual-Basic declaration that contains a device interface GUID. For the HID class, applications can retrieve the GUID with the function HidD_GetHidGuid.

Not all devices require using device interface GUIDs. For example, applications can use Windows' file system to access files on mass-storage devices and printing functions to access printers. A custom driver can define its own API to enable applications to access devices without having to provide a GUID.

Some older drivers define a symbolic link for each device they control. For example, the first device attached might be \\.*mydevice0*, followed by \\.*mydevice1,* \\.*mydevice2*, and so on up as needed. Applications access these devices using the symbolic links instead of device interface GUIDs.