The following material is excerpted from

USB Complete

Everything You Need to Develop Custom USB Peripherals

by Jan Axelson

For more information about the book, code examples, and links to other USB information and tools, visit Lakeview Research on the web:

www.lvr.com

Lakeview Research
2209 Winnebago St.
Madison, WI 53704
USA

Phone: 608-241-5824
Fax: 608-241-5848
Email: info@lvr.com
Web: http://www.lvr.com

ISBN 0-9650819-3-1

10

How the Host Communicates

A USB peripheral is of no use if its host PC doesn't know how to communicate with it. Under Windows, any communication with a USB peripheral must pass through a device driver that knows how to communicate with the system's USB drivers and the applications that access the device.

This chapter explains how Windows applications communicate with USB devices 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, or just about any collection of electrical circuits connected to the computer. The device may be inside the computer's enclosure (an internal disk drive, for example) or it may use a cable to connect to the computer

(as with the keyboard and mouse). The device can be a standard peripheral type or not, including one-of-a-kind, custom devices.

Insulating Applications from the Details

A device driver insulates applications from having to know details about the physical connections, signals, and protocols required to communicate with a device. Applications are the programs the users run, including special-purpose applications to support custom hardware. A device driver can enable application code to access a peripheral using only the peripheral's name (such as HP LaserJet) or a port designation (such as COM1 or LPT2). The application doesn't have to know the physical address of the port the peripheral attaches to (such as 378h), and it doesn't have to explicitly monitor and control the handshaking signals that the peripheral requires (Busy, Strobe, etc.).

A device driver accomplishes its mission by translating between application-level and hardware-specific code. The application-level code typically uses a set of functions supported by the operating system. The hardware-specific code handles the protocols necessary to access the peripheral's circuits, including detecting the states of status signals and toggling control signals at appropriate times.

Windows includes application programmer's interface (API) functions that enable device drivers and applications to communicate with each other. These functions are part of a set of thousands of functions that enable applications to control the display, handle messages, access memory, read and write to disks and other devices, and more. API functions used to read and write to USB devices are ReadFile, WriteFile, and DeviceIOControl. Applications written in Visual Basic, C/C++, and Delphi can call API functions.

Although API functions simplify the process of communicating with hardware, some programming tasks remain challenging. API functions tend to have specific and rigid requirements for the values they pass and return, and it's not unusual for a mistake to result in an application or system crash.

To make programming simpler and safer, Visual Basic has its own controls for common tasks. For example, applications can use the Printer Object to

send data to printers and the MSComm control to communicate with serial ports. The controls provide a simpler and more bulletproof programming interface for setting parameters and exchanging data. The underlying code within the control will likely use API functions to communicate with device drivers, but using the control insulates application programmers from dealing with the sometimes arcane details of the API calls. However, Visual Basic doesn't include a control for generic access to generic USB devices.

Third-party vendors offer controls for specialized tasks such as communicating with bar-code scanners and other data-acquisition devices. A vendor may offer a control for communicating with a specific USB peripheral, and controls for generic USB communications may appear eventually as well.

On the hardware side of the communication, some device drivers are monolithic drivers that handle everything, from communicating with applications to reading and writing to the ports or memory addresses that connect to the device's hardware. Other drivers, including 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 communicates with applications, the bottom layer communicates with the hardware, and in between there may be one or more additional layers. The layered driver model is more complicated as a whole, but it actually simplifies the job of writing drivers because devices can share code for tasks they have in common. The drivers that handle communications with the system's USB hardware are built into Windows 98, so peripheral vendors don't have to provide these.

Options for USB Devices

There are several approaches to writing or obtaining a driver for a device. Often there is more than one way that will work, and the choice depends on a combination of what's easier, cheaper, and offers better performance.

Standard Device Types

Many peripherals fit into standard classes such as disk drives, printers, modems, keyboards, and mice. For these, Windows includes universal drivers that any device in the class can use. If a device has unique features, a ven-

dor can provide a supplemental driver called a mini-driver that adds capabilities to the universal driver.

Some peripheral types are available with a choice of interfaces, which may include USB. A keyboard may use the original legacy keyboard interface or USB. A disk drive may use an IDE, SCSI, printer-port, or USB interface. In cases like these, a mini-driver can communicate between the universal driver and the interface that the device uses. Windows provides a USB mini-driver for HID-class devices, which include keyboards, mice, and joysticks. For other devices, Windows may not have built-in support for a USB interface, so the product vendor must supply either a mini-driver or a complete, custom driver for the device.

Custom Devices

Some peripherals are custom devices intended for use only with specific applications. One example is the development boards for USB chips, which are designed for use with a vendor's monitor application. Other examples are data-acquisition units, motor controllers, and test instruments. Windows has no knowledge of these specialized devices, so it has no built-in drivers for them.

However, just because Windows doesn't know about a device doesn't mean that applications can't access it. There are several options for communicating with custom devices, not all requiring a custom driver.

These are the options for drivers for any device under Windows 98:

- A generic driver suitable for communicating with a variety of devices. Example device: a test instrument whose host uses the *bulkusb.sys* driver included in the Windows 98 DDK to receive data in bulk transfers.
- A class driver and a mini-driver that supports the device's USB interface. Example device: an HID-class mouse, which uses Windows' HID-class driver along with the HID mini-driver that enables the class driver to communicate with the system's USB drivers.
- A custom driver written specifically for the device. Example device: a data-acquisition unit with a driver that defines a series of control requests

that applications can use to configure the unit and read data from it. The custom driver may be adapted from a generic driver.

How Applications Communicate with Devices

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.

What Is a Device Driver?

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. But 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.

User and Kernel Modes

Code that runs under Windows 98 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 run in kernel mode.

In user mode, Windows limits access to memory and other system resources. Windows won't allow an application to access an area of memory that the operating system has designated as protected. This enables a PC to run multiple applications at the same time, with none of the applications interfering with each other. In theory, even if an application crashes, other applications are unaffected. (Of course in reality it doesn't always work that way, but that's the theory.) Under Windows 98, applications can access I/O ports directly, unless a low-level driver has reserved the port, preventing access. On Pentiums and other x86 processors, user mode corresponds to the CPU's Ring 3 mode.

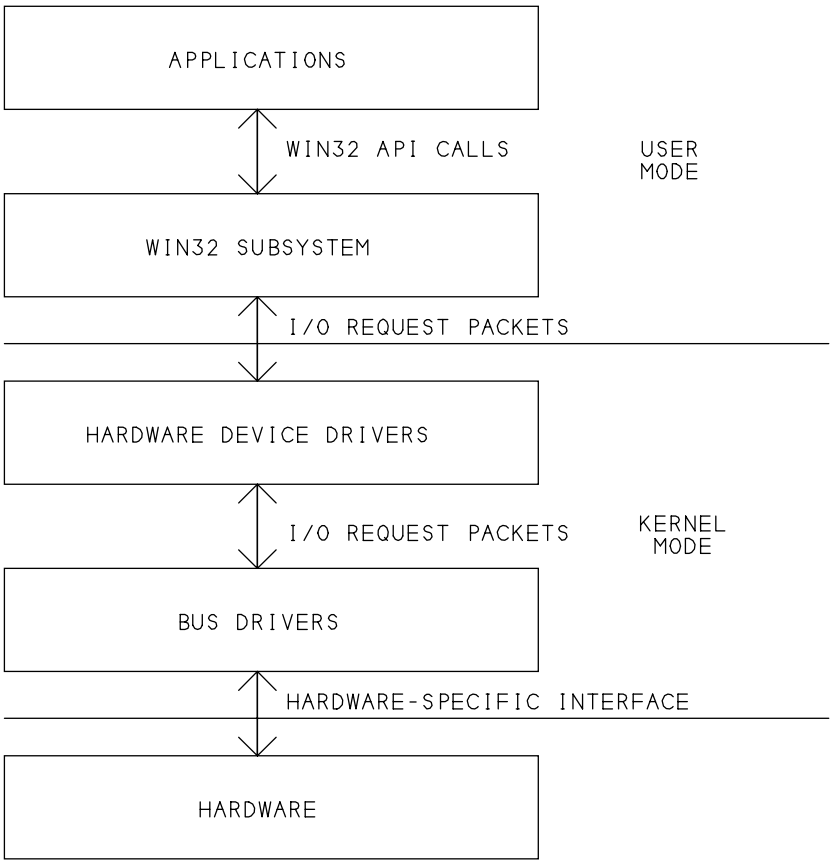

Figure 10-1: Windows 98 uses a layered driver model, with separate drivers for devices and the buses they connect to.

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. On Pentiums and other x86 processors, kernel mode corresponds to the CPU's Ring 0 mode. Figure 10-1 shows the major components of user and kernel modes in a USB communication.

Applications and drivers each use their own language to communicate with the operating system. Applications use Win32 API functions. 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 or carries out a single input or output action. A device driver for a USB device uses IRPs to pass communications to and from the bus drivers that handle USB communications. The bus drivers in turn use IRPs to pass communications to and from drivers that manage aspects of communications closer to the bus. The final bus driver in the series communicates directly with the hardware. The bus drivers are included with Windows and require no programming by applications programmers or device-driver writers.

The Win32 Driver Model

USB device drivers for Windows must conform to the Win32 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 file types may also use the *.sys* extension.)

Like other low-level drivers, a WDM driver has abilities not available to applications because it communicates with the operating system at a more privileged level. A WDM driver can permit or deny an application access to a device. For example, a joystick driver can allow any application to use a joystick, or it can allow one application to reserve the joystick for its exclusive use. Other abilities that Windows reserves for WDM and other low-level drivers include DMA transfers and responding to hardware interrupts.

Driver Models for Different Windows Flavors

The Win32 Driver Model was designed to provide a common driver model for use by any device under Windows 98 and later, including Windows 2000 (the successor to Windows NT 4).

Earlier versions of Windows used different models for device drivers. Windows 95 used VxDs (virtual device drivers). NT 4 used a type of driver called kernel-mode drivers. Developers who wanted to support both Windows 95 and Windows NT had to develop a driver for each. But one WDM driver will work under both Windows 98 and Windows 2000.

The USB bus drivers included with Windows 98 are WDM drivers. Although Windows 98 continues to support VxDs (as well as drivers contained in DLLs), USB devices must use WDM device drivers because their drivers must communicate with the system's bus drivers.

The Win32 Driver Model isn't completely new; it's mostly a combination of what was available in Windows 95 and NT. A WDM driver is an NT kernel-mode driver with the addition of Windows 95's Plug-and-Play and power-management features. The final editions of Windows 95 (versions OSR 2.1 and higher) had some support for WDM drivers as well. These editions weren't available to retail customers, but were available only to vendors who installed the software on the computers they sold. In Windows 98, the WDM support was much expanded and improved.

How can two different operating systems, which previously required very different drivers, now use the same drivers? Windows 98 includes the driver *ntkern.vxd*, which tricks WDM drivers into thinking they're communicating with an NT-like operating system. All WDM drivers running on Windows 98 require this driver, which is included with Windows 98.

Programming Languages

Application programmers have a choice in programming languages, including Visual Basic, Delphi, and Visual C++. To write a USB device driver, however, you need a tool that is capable of compiling a WDM driver, and this means using Visual C++.

Layered Drivers

USB communications use a layered driver model, where each layer handles a piece of the communication process. Dividing communications into layers is efficient because it enables different devices that have some tasks in common to use the same driver for those tasks. For example, all kinds of devices may connect to the USB, so it makes sense to have one set of drivers, accessible to all and included in the operating system, to handle the USB-specific communications. The alternative would be to have each device driver communicate directly with the USB hardware, with much duplication of effort.

USB Driver Layers

The portion of Windows that manages communications with devices is the I/O subsystem. The subsystem has several layers, with each layer containing one or more drivers that handle a set of related tasks. Requests pass in sequence from one layer to the next. Within the I/O subsystem, the I/O manager is in charge of communications. One element within the I/O subsystem is the USB subsystem, which includes the drivers that handle USB-specific communications for all devices.

The set of protocols used by the drivers is called a stack. (This is different from the CPU stack introduced in Chapter 8.) You can think of the layers as being stacked one above the next, with communications passing in sequence up and down the stack.

Device and Bus Drivers

Under Windows, USB communications use two types of drivers: device drivers and bus drivers. A device driver handles communications that are specific to a single device or to a class of devices. A single USB device may use one or more device drivers. Below the device driver are three drivers that handle aspects of bus communications: the hub driver, the bus-class driver, and the host-controller driver. Figure 10-2 shows how these work together in USB communications.

The Device Driver

A device driver enables applications to talk to USB devices using API functions. The API functions are part of Windows' Win32 subsystem, which is also in charge of user functions such as running applications, managing user input via the keyboard and mouse input, and displaying output on the screen. To communicate with a USB device, an application doesn't have to know anything about the USB protocol, or even if the device uses USB at all.

A device may use a class driver or a custom driver. A class driver handles functions that are common to a set of similar devices. Windows includes

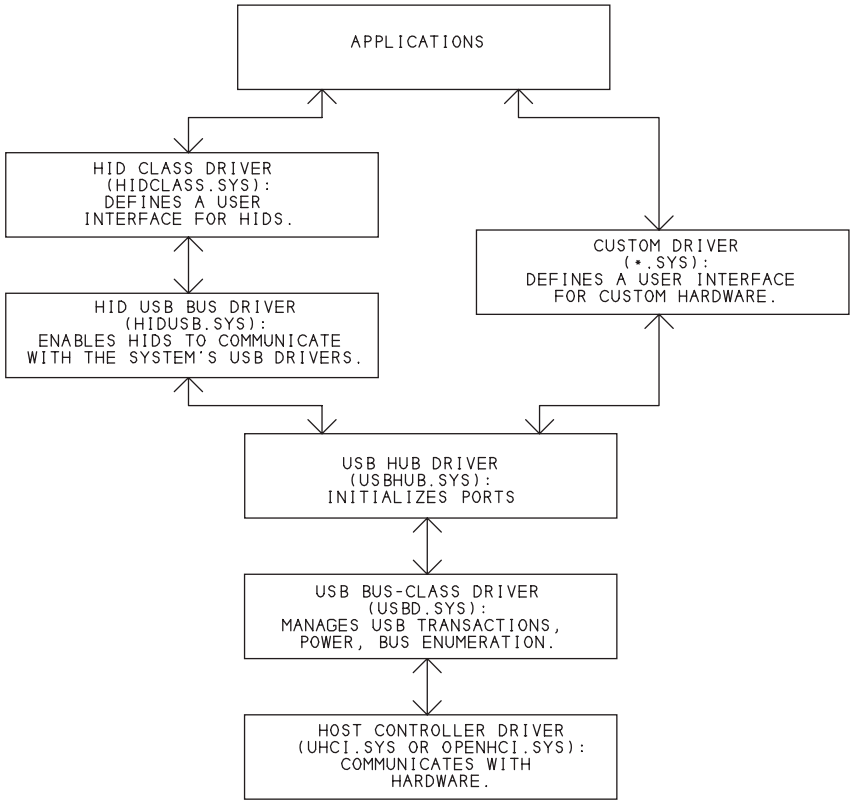

Figure 10-2: USB communications use a host controller driver, class driver, hub driver, and a device driver that may consist of one or more files.

class drivers for HIDs and other common peripherals. (These driver classes are distinct from the USB's device classes, although HIDs are considered a class in both.) The class driver handles functions that are common to all peripherals in the class.

When a device or subclass has requirements beyond what the class driver handles, a mini-driver can add the needed capabilities. For example, not all HIDs have a USB interface, so Windows provides a separate mini-driver that enables the HID driver to communicate with the USB subsystem when needed.

The Bus Drivers

The USB's bus drivers consist of the root-hub driver, the bus-class driver, and the host-controller driver. The root-hub driver manages the initializing of ports and in general manages communications between device drivers and the bus-class driver. The bus-class driver in turn manages bus power, enumeration, USB transactions, and communications between the root-hub driver and the host-controller driver. The host-controller driver knows how to talk to the host controller's hardware, which connects to the bus. The host-controller driver is separate from the bus-class driver because Windows 98 supports two types of host controllers, each with its own driver. The bus-class driver can communicate with either controller type.

The bus drivers are part of Windows, and application and device-driver writers don't have to know the details about how they work. Perhaps because of this, Microsoft provides very little in the way of documentation for them.

The Communications 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 custom driver.

Preliminary Requirements

Before an application can communicate with a device, several things must happen. The device must be attached to the bus. Windows must enumerate the device and identify the driver for the device. And the application that will access the device must obtain a handle that identifies the device and enables communications with it.

Plugging in the cable attaches the device. Windows handles enumeration automatically when it's notified of the device's attachment, as described in Chapter 5. To identify which driver to use, Windows compares the retrieved descriptors with the information in its INF files, as described in Chapter 11.

The handle is a unique identifier that Windows assigns to the device. An application gets the handle by calling the CreateFile API function with a symbolic link that identifies the device.

Some drivers explicitly define a symbolic link for each device they control. For example, Cypress' *ezusb.sys* driver identifies the first EZ-USB chip as ezusb-0. If there are additional EZ-USBs, the driver identifies them as ezusb-1, ezusb-2, and so on up.

Other drivers use a newer method supported by Windows, where the symbolic link contains a globally unique identifier (GUID). The GUID is a 128-bit number that uniquely identifies an object, which may be any system class, interface, or other entity that the software treats as an object.

Windows defines GUIDs for standard objects such as the HID class. For unique devices, developers can obtain a GUID using the *guidgen.exe* program included with Visual C++. The GUID is then included in the driver code.

The *guidgen* program uses a complex algorithm that takes into account a machine identifier, the date and time, and other factors that make it extremely unlikely that another device will end up with an identical GUID. The algorithm was originally defined by the Open Software Foundation.

The standard format for expressing GUIDs divides the GUID into five sets of hex characters, separated by hyphens. This is the GUID for the HID class:

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

Applications can use API calls to retrieve class and device GUIDs from the operating system.

The User's Role

When a device is attached and ready to transfer data, the host initiates a transfer by requesting it. To read data from a data-acquisition unit, the user might click a button in a data-acquisition application to cause the application to request data from the device. Or the 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 user runs the application.

The Application's Role

After the user requests a transfer, the application begins communications with the device. The Windows API offers two ways of accessing devices: using the ReadFile/WriteFile pair or DeviceIOControl. A driver may use either or both. Each call includes the request, other required information such as the data to write or amount of data to read, and the device's handle. The Platform SDK section in the MSDN library documents the calls.

Although the names suggest that they're used only with files, ReadFile and WriteFile are actually general-purpose functions that can transfer data to and from any driver that supports them. The data read or to be written is stored in a buffer specified by the call. Chapter 14 has more on how to use ReadFile and WriteFile.

DeviceIOControl is another way to transfer data to and from buffers. Included in each DeviceIOControl request is a 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 code. Some codes are commands that don't need to pass additional data.

Windows defines control codes used by disk drives and other common devices. These are examples:

> IOCTL_STORAGE_CHECK_VERIFY determines if media is present and readable on removable media.
>
> IOCTL_STORAGE_LOAD_MEDIA loads media on a device.
>
> IOCTL_STORAGE_GET_MEDIA_TYPES returns the types of media supported by a drive.

A driver may also define its own control codes. Because the codes are sent only to a specific driver, it doesn't matter if other drivers use the same codes. The driver for Cypress' thermometer application for the CY7C63001 defines codes to get the temperature and button state, set LED brightness, and read and write to the controller's RAM and ports. This is a Visual-Basic declaration for DeviceIOControl:

```
Declare Function DeviceIoControl Lib "kernel32" _
    (ByVal hDevice As Long, _
```

```
        ByVal dwIoControlCode As Long, _
        lpInBuffer As Any, _
        ByVal nInBufferSize As Long, _
        lpOutBuffer As Any, _
        ByVal nOutBufferSize As Long, _
        lpBytesReturned As Long, _
        lpOverlapped As OVERLAPPED) _
    As Long
```

This is a call that uses the control code 04h, which requests the thermometer to send the temperature:

```
ltemp = DeviceIoControl _
    (hgDrvrHnd, _
    4&, _
    lIn, _
    lInSize, _
    lOut, _
    lOutSize,_
    lSize, _
    gOverlapped)
```

The Device Driver's Role

When an application makes an API call, Windows passes the call to the appropriate device driver. The driver converts the request to a format the USB bus-class driver can understand.

As mentioned earlier, drivers communicate with each other using structures called I/O Request Packets (IRPs). For USB communications, the IRPs contain structures called USB Request Blocks (URBs) that specify protocols for the tasks of configuring devices and transferring data. The URBs are documented in the Windows 98 DDK.

If you're using an existing device 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 device driver, you need to provide the IRPs that communicate with the system's USB drivers.

The Hub Driver's Role

The host's hub driver resides between a device-specific or USB-class driver and the USB bus-class driver. The hub driver handles the initializing of the root hub's ports and any devices downstream of the ports. This driver requires no programming by device developers. Windows 98 includes the hub driver *usbhub.sys*.

The Bus-class Driver's Role

The USB bus-class driver translates communications requests between the hub driver and the host-controller driver. It handles bus enumeration, power management, and some aspects of USB transactions. These communications require no programming by device developers. Windows 98 includes the bus-class driver *usbd.sys*.

The Host-controller Driver's Role

The host-controller driver communicates with the host-controller hardware, which in turn connects to the bus. The host-controller driver requires no programming by device developers.

Windows supports two varieties of host controllers. Controllers that conform to the Open Host Controller Interface standard use the driver *openhci.sys*, and controllers that conform to the Universal Host Controller Interface standard use the driver *uhci.sys*. Both drivers provide a way for the USB hardware to communicate with the USB's bus-class driver. Although they differ in how they do so, any differences should be transparent to driver developers and application programmers.

The two drivers take different approaches to implementing the host-controller's functions. The UHCI places more of the communications burden on software and allows the use of simpler, cheaper hardware. The OHCI places more of the burden on the hardware and allows simpler software control. UHCI was developed by Intel and the specification is available on Intel's website. OHCI was developed by Compaq, Microsoft, and National Semiconductor and the specification is available on Compaq's website. The USB Implementers Forum's website has links to both.

The Device's Role

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

The Response

Many communications will require a response, which may contain data sent in response to the request or just a packet with a handshake 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 device driver may pass a response on to an application, which may display the result or take other action.

Ending Communications

When an application closes or otherwise decides that it no longer needs to access the device, it uses the API function CloseHandle to free system resources.

More Examples

Communications with other USB devices follow a similar pattern, though there can be differences in how the transfer initiates and in how the device driver handles communications.

In the above example, the host ignores the device until the user takes an action. Other examples of a user initiating a transfer are clicking on a USB drive's icon to view a disk's folders in Windows Explorer's My Computer or clicking Print in an application to send a file to a USB printer. In each of these examples, nothing happens 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.

In some cases, the host continuously sends requests to the device whether or not an application has requested them. For example, a keyboard driver

causes the host to make periodic requests for keypress data, whether or not the user has pressed any keys.

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 device can use the USB's remote-wakeup feature to initiate a transfer by signaling its hub, and in turn the host, to request it to resume communications.

As mentioned earlier, some devices use class drivers instead of, or in addition to, a driver for the specific device. For an HID-class device, applications may communicate with the system's HID-class driver, which handles communications for all HIDs. Not all HIDs have a USB interface, but those that do also use a mini-driver to handle the USB-specific communications.

Some USB devices may use yet another type of driver, called a legacy virtualization driver. To communicate with the keyboard, mouse, and joystick, Windows 98 uses the virtual device drivers (VxDs) inherited from Windows 95. When one of these peripherals has a USB interface, a legacy virtualization driver translates between the device's HID interface and the VxD. The legacy virtualization driver is a VxD that knows how to talk to the HID driver.

Choosing a Driver Type

How do you decide whether to use an existing driver, a custom driver, or a combination? Sometimes the choice is limited by what's available for the device. From there it depends on a combination of the performance you need, cost, and speed of development.

Drivers Included with Windows

When it's feasible, a simple approach to accessing a USB device is to use what's available in Windows. This way, there are no drivers to write or install and any computer running Windows 98 or later can access the device.

It would be nice if Windows included generic drivers that enabled any application to request control, interrupt, bulk, and isochronous transfers. Maybe

someday Windows will have this. But as of Windows 98, the included USB drivers are limited.

HID Drivers

The first class of USB peripherals suitable for a variety of applications and fully supported by Windows is the human interface device (HID) class. Applications can use API calls to identify a device, read and write generic values, and set and read the states of buttons on the device. The data is formatted in reports. The HID class has defined report formats for mice, keyboards, and joysticks, or you can define your own format.

But an HID doesn't have to be a standard peripheral type, and it doesn't even have to have a human interface. The only requirement is that the descriptors stored in the device must conform to the requirements of HID-class descriptors, and the device must send and receive data using interrupt or control transfers as defined in the HID specification.

The main limitation to HID communications is the available transfer types. For device-to-host data transfers, HIDs can use interrupt or control transfers. For host-to-device transfers, Windows 98 SE (or any host that complies with the HID 1.1 or later specification) will use interrupt transfers if an OUT interrupt pipe is available, or control transfers if not. The original release of Windows 98 complies only with the HID 1.0 specification and will use control transfers for all host-to-device transfers.

As Chapter 3 explained, interrupt transfers aren't the fastest transfer type, and they don't have the guaranteed transfer rate of isochronous transfers (though they do have guaranteed maximum latency). Control transfers have no guaranteed rate or latency. But even with these limitations, the simplicity of using the HID functions makes it attractive when the limits are acceptable.

An alternative to using API functions for accessing HIDs is to use Microsoft's DirectX components. DirectX enables control of system hardware, including HIDs. DirectX originated as a tool for game programmers, but has since expanded to enable accessing any HID. The advantage of using DirectX is that it provides faster access. Instead of having to poll an

input with ReadFile, you can configure the DirectX software components to notify an application when data is available to read.

The DirectInput and DirectInput2 components of DirectX enable communications with HIDs. For Visual-Basic programmers, DirectX version 7 includes support that enables Visual-Basic programmers to use DirectX.

Point-of-Sale Driver

The Point-of-Sale USB driver is another driver that was originally intended for a specific category of devices, but may be useful to devices outside the original category. Point-of-sale (POS) devices include bar-code scanners, displays, receipt printers, and other devices used in sales transactions.

In July 1999, Microsoft released the POSUSB point-of-sale driver for USB devices. The driver enables host applications to communicate with the device as if it were connected to a conventional COM port, using CreateFile, ReadFile, and WriteFile. For bidirectional communications, a device needs one IN endpoint and one OUT endpoint. A device that requires only one-way communications needs just one IN or OUT endpoint. The endpoints may be configured for bulk or interrupt transfers.

The supporting documentation includes manuals for host application and firmware programmers. The driver and documentation is available from Microsoft's website, and is scheduled to ship with future releases of Windows.

Vendor-supplied Drivers

Another way to communicate with a device is to use a driver supplied by the chip's vendor. The ideal is a ready-to-install, general-purpose driver, along with complete, commented source code in case you want to adapt it for use with a particular device. The driver should also include documentation that shows how to use API calls to open a handle to the device and read and write to it in application code.

But the usefulness of vendor-supplied drivers varies. A driver is less useful if it turns out to be buggy, doesn't include the features you need, or has sketchy documentation that makes it hard to understand and use.

Cypress Semiconductor's *ezusb.sys* is an example of a driver that you can use without modification to communicate with Cypress' EZ-USB chip using any transfer type. The driver defines a set of DeviceIOControl codes for making standard requests in control transfers, exchanging data in interrupt, bulk, and isochronous transfers, and performing other USB-related and EZ-USB-specific functions. The driver also handles the EZ-USB's unique method of having the host load the chip's firmware on power-up or attachment.

Other Generic Drivers

If the chip's vendor doesn't supply a driver, in some cases you can use a generic driver that has no device-specific features or requirements. For bulk transfers, Windows 98's DDK includes source and compiled code, documentation, and an example application for the *bulkusb.sys* driver. The driver was written for Intel's 8x930 chip, but is designed to work with just about any USB chip that supports bulk transfers. Applications use ReadFile and WriteFile for data transfers.

In a similar way, the Windows 98 DDK also includes the *isousb.sys* driver for handling isochronous transfers. This driver was also written for the 8x930 chip.

If you decide to use either of these, check the Implementers Forum's webboard for tips and fixes.

Custom Drivers

The final option is to use a custom driver. Sometimes there is no generic or vendor driver that includes the transfer types you want to use. Or you may want to define custom DeviceIOControl codes.

An example of a custom driver is Cypress' driver for its thermometer application in the Starter Kit for the CY7C63001. This driver is written to support a specific application, rather than for general use. The driver defines DeviceIOControl codes to get the temperature and button state, set LED brightness, and read and write to the controller's RAM and ports. This driver was written before the release of Windows 98. If the device doesn't

need to work with hosts running Windows 95, you could instead use the HID drivers to transfer the information, without having to write a driver.

The EZ-USB is a custom driver as well. The driver supports the vendor-specific request AnchorLoad, which causes the device to store received firmware and simulate detaching and reattaching to the bus.

If a chip's vendor provides source code for a driver, even if you don't use the driver as-is, you can modify the code or use portions of it, rather than starting from scratch. Ideally, the source code will include liberal commenting to help you understand it. It should include instructions that explain how to compile and install the driver. Even making minor changes or additions to an existing driver can be difficult if the code isn't well documented.

Writing a Custom Driver

If you don't have experience writing device drivers, creating a WDM driver is not a trivial task. It requires an investment in tools, expertise in C programming, and a fair amount of knowledge about how Windows communicates with hardware and applications. However, there are resources that can simplify and speed up the process.

Requirements

The minimum requirement for writing a device driver is Microsoft's Visual C++, which is capable of compiling WDM drivers. The compiler also includes a programming environment and a debugger to help during development.

Beyond this basic requirement, other tools can help to varying degrees, including the Windows 98 Device Developer's Kit (DDK), a subscription to Microsoft's Developer's Network (MSDN), driver toolkits, and advanced debuggers.

The Windows 98 DDK includes example code and developer-level documentation for Windows 98. The USB-related documentation includes tutorials on WDM drivers and HIDs and source code for several USB drivers, including a bulk-transfer driver, an isochronous-transfer driver, a filter

driver, and the *usbview* utility. The examples can be a useful starting point in developing your own drivers. You can download the Windows 98 DDK from Microsoft's website.

MSDN is Microsoft's subscription service to massive quantities of documentation, examples, and developer's tools for Microsoft products. The topics covered include WDM driver development, USB, and DirectX, with quarterly updates. There are several levels of subscription that enable you to get the documentation alone or with varying amounts of Microsoft applications and development tools. Much of the information and other tools are also downloadable from Microsoft's website.

A driver toolkit provides a way to jumpstart driver development by doing as much of the work for you as possible. Toolkits that support USB drivers are available from Bluewater Systems and DriverWorks from Compuware NuMega.

Using a Driver Toolkit

To give an idea of what's involved in creating a device driver, I'll show how it's done with BlueWater Systems' WinDK development library. In addition to code libraries, the product includes a driver Wizard, sample drivers, and excellent documentation and tutorials on related topics. WinDK creates drivers for Windows 98. An optional USB Extension enables you to use the same source code to create a driver that will run on Windows NT 4.

The Libraries

There are two libraries, depending on whether you prefer to program in C or C++.

C++ programmers can use a library of classes for carrying out the tasks common to many drivers. For example, the functions of the CUsb class handle USB-related activities, including finding a device on a system, learning its capabilities, and transferring data using control, bulk, interrupt, and isochronous transfers. The CPnPDevice class handles Plug-and-Play functions, and the CRegistry class handles saving and retrieving information from the system registry.

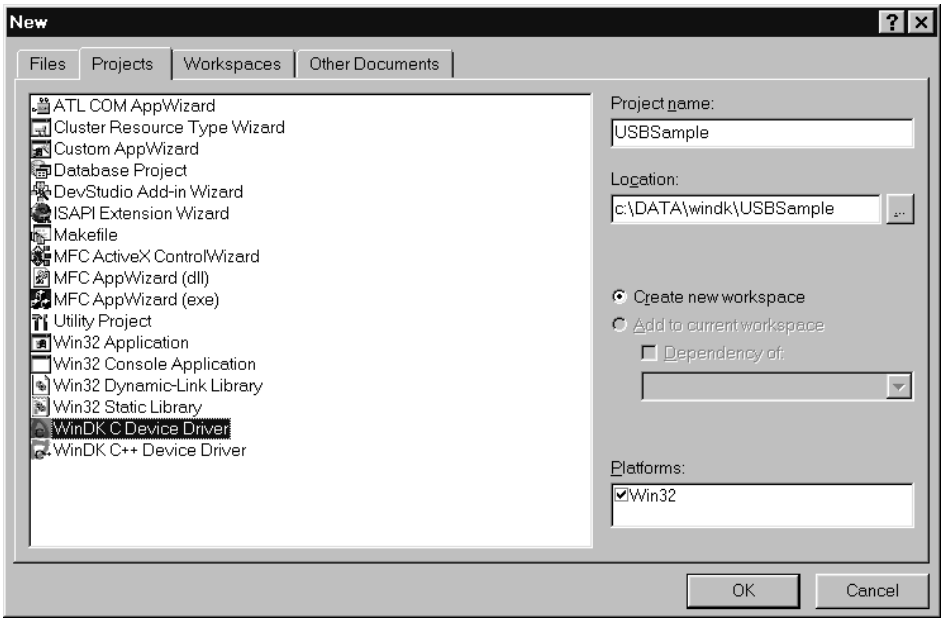

Figure 10-3: To create a driver with WinDK's Wizard, select one of the WinDK options from Visual C++'s *New* menu.

C programmers can accomplish the same things using the library of C functions.

Using the Wizard

The WinDK Wizard gets you started by creating a driver with as much code as possible filled in for you. The toolkit integrates into the Visual C++ programming environment. You start the Wizard from within Visual C++ by selecting File > New. As Figure 10-3 shows, the list of options includes WinDK C Device Driver and WinDK C++ Device Driver.

Figure 10-4 shows a few of the screens that the Wizard uses to ask you about your driver.

For a USB driver, you provide the following information to the Wizard:

- The project name and location.

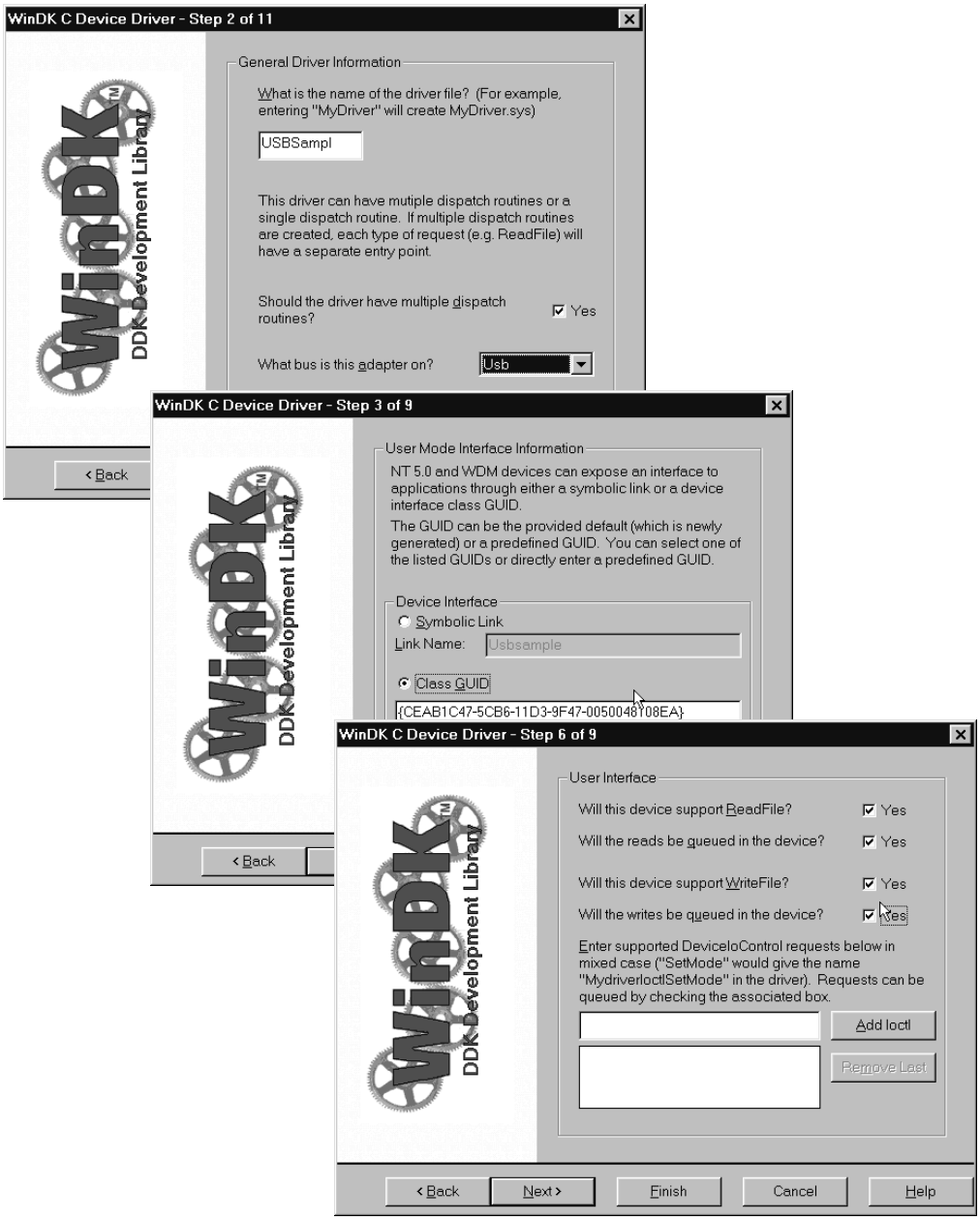

Figure 10-4: WinDK's Wizard queries you about your driver, then does as much of the work for you as possible.

- Whether to generate C or C++ code.
- Installation directories.
- Device class name.
- Driver type (WDM hardware driver).
- Bus type (USB).
- Vendor and Product IDs.
- Device Interface. This may be a GUID generated by the Wizard, an existing class GUID, or a symbolic link you specify.
- Optional registry keys for device-specific parameters and initial values for these keys.
- Whether the device will support ReadFile and WriteFile or DeviceIOControl requests, and the names of any DeviceIOControl requests.

For USB drivers, there's no need to specify port or memory addresses or interrupts.

With this information, the Wizard creates a driver, an INF file, a test application, and a documentation file.

The driver includes default routines for power management, Plug and Play, system control, and creating and closing the device. For USB devices, these routines normally don't need modifying. After the driver's device is enumerated, the driver retrieves information about the device's configuration and endpoints.

If you specified DeviceIOControl functions for your driver, the Wizard adds skeleton code for each of these. You need to add the device-specific code that tells the driver what to do when an application calls a DeviceIOControl function. In a similar way, if the driver uses ReadFile and WriteFile, you add the device-specific code to the skeleton provided.

The INF file has the driver's filename, vendor and product IDs, and other information in the required format.

The Wizard also creates a command-line test application that opens the device with CreateFile and enables testing of the driver.

The documentation file lists each file created by the Wizard and explains its purpose.

The process is similar if you're using Compuware NuMega's DriverWorks. There's a Wizard that walks you through the process of defining your driver, then produces code to match what you've specified.

Another tool that most driver developers find essential is a debugger that enables testing of the driver code using breakpoints, single stepping, and other standard techniques. The Professional and higher levels of MSDN subscriptions include the WinDbg debugger. Compuware NuMega's Soft-ICE debugger enables debugging of WDM drivers using a single Windows 98 system.